Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*

Jasmin Christian Blanchette and Tobias Nipkow

Institut für Informatik, Technische Universität München, Germany
{blanchette,nipkow}@in.tum.de

Abstract. Nitpick is a counterexample generator for Isabelle/HOL that builds on Kodkod, a SAT-based first-order relational model finder. Nitpick supports unbounded quantification, (co)inductive predicates and datatypes, and (co)recursive functions. Fundamentally a finite model finder, it approximates infinite types by finite subsets. As case studies, we consider a security type system and a hotel key card system. Our experimental results on Isabelle theories and the TPTP library indicate that Nitpick generates more counterexamples than other model finders for higher-order logic, without restrictions on the form of the formulas to falsify.

1 Introduction

Anecdotal evidence suggests that most "theorems" initially given to an interactive theorem prover do not hold, typically because of a typo or a missing assumption, but sometimes because of a fundamental flaw. Modern proof assistants often include counterexample generators that can be run on putative theorems or on specific subgoals in a proof to spare users the Sisyphean task of trying to prove non-theorems.

Isabelle/HOL [17] includes two such tools: Quickcheck [4] generates functional code for the higher-order logic (HOL) formula and evaluates it for random values of the free variables, and Refute [23] searches for finite countermodels of a formula through a reduction to SAT (Boolean satisfiability). Their areas of applicability are almost disjoint: Quickcheck excels at inductive datatypes but is restricted to the executable fragment of HOL (which excludes unbounded quantifiers) and may loop endlessly on inductive predicates. Refute copes well with logical symbols, but inductive datatypes and predicates are mostly out of reach due to combinatorial explosion.

In the first-order world, the Alloy Analyzer [13], a testing tool for first-order relational logic (FORL), has enjoyed considerable success lately. Alloy's backend, the relational model finder Kodkod [21], is available as a stand-alone Java library and is used in many projects.

Alloy's success inspired us to develop a new counterexample generator for Isabelle, called Nitpick.[1] It uses Kodkod as its backend, thereby benefiting from Kodkod's optimizations (notably its symmetry breaking) and its rich relational logic. The basic translation from HOL to FORL is conceptually simple (Section 3); however, common HOL

* This work is supported by the DFG grant Ni 491/11-1.

[1] The name Nitpick is appropriated from Alloy's venerable precursor.

idioms such as (co)inductive datatypes and (co)inductive predicates necessitate a translation scheme tailored for SAT solving (Section 4). In addition, Nitpick benefits from many novel optimizations that greatly improve its performance, especially in the presence of higher-order constructs (Section 5).

As case studies, we consider the Isabelle formalizations of a hotel key card system and a security type system (Section 6), both of which are currently beyond the reach of Quickcheck and Refute. Our evaluation indicates that Nitpick falsifies more formulas than Quickcheck and Refute (Section 7), to a large extent because it imposes no syntactic restrictions on the formulas to falsify. Nitpick is integrated with the TPTP benchmark suite [20] and exposed three bugs in the higher-order provers TPS [1] and LEO-II [3].

2 Background

2.1 Higher-Order Logic (HOL)

The types and terms of HOL [12] are that of the simply typed λ-calculus extended with type constructors and constants:

Types:		*Terms:*	
$\sigma ::= \alpha$	(type variable)	$t ::= x^\sigma$	(variable)
$\mid (\sigma, \ldots, \sigma)\, \kappa$	(type construction)	$\mid c^\sigma$	(constant)
		$\mid t\, t$	(application)
		$\mid \lambda x^\sigma.\, t$	(abstraction)

We write κ for $()\, \kappa$, $\sigma\, \kappa$ for $(\sigma)\, \kappa$, and $\sigma\, \kappa\, \tau$ for $(\sigma, \tau)\, \kappa$. HOL's standard semantics interprets the Boolean type o and the function space $\sigma \to \tau$. Other types are defined, notably the product type $\sigma \times \tau$. The function arrow associates to the right, reflecting the left-associativity of application. We assume throughout that terms are well-typed using the standard typing rules and write x and c instead of x^σ and c^σ when the type σ is irrelevant or can be inferred from the context. A formula is a term of type o.

Type variables occurring in the type of a constant can be instantiated, offering a restricted form of polymorphism. Standard models interpret the constant $\simeq^{\alpha \to \alpha \to o}$ as equality on α for any instance of α. Logical connectives and quantifiers can be defined in terms of $\simeq$; for example, $\mathit{True}^o = (\lambda x^o.\, x) \simeq (\lambda x.\, x)$ and $\forall^{(\alpha \to o) \to o} = (\lambda P^{\alpha \to o}.\, P \simeq (\lambda x.\, \mathit{True}))$. The traditional binder notation $Q x.\, t$ abbreviates $Q\, (\lambda x.\, t)$.

2.2 First-Order Relational Logic (FORL)

Kodkod's idiosyncratic logic, FORL, combines elements from first-order logic and relational calculus, to which it adds the transitive closure operator [21]. Its formulas involve variables and terms ranging over relations (sets of tuples drawn from a universe of uninterpreted atoms) of arbitrary arities. The logic is unsorted, but each term denotes a relation of a fixed arity that can be inferred from the arities of its variables. Our translation relies on the following FORL fragment.

Formulas:		*Terms:*	
$\varphi ::= \mathsf{false}$	(falsity)	$r ::= \mathsf{none}$	(empty set)
$\mid \mathsf{true}$	(truth)	$\mid \mathsf{iden}$	(identity relation)
$\mid m\ r$	(multiplicity constraint)	$\mid \mathsf{a}_n$	(atom)
$\mid r \simeq r$	(equality)	$\mid x$	(variable)
$\mid r \subseteq r$	(inclusion)	$\mid \{\langle d,\ldots,d\rangle \mid \varphi\}$	(comprehension)
$\mid \neg\varphi$	(negation)	$\mid \pi^n_n(r)$	(projection)
$\mid \varphi \wedge \varphi$	(conjunction)	$\mid r^+$	(transitive closure)
$\mid \forall d\colon \varphi$	(universal quantification)	$\mid r.r$	(dot-join)
		$\mid r \times r$	(Cartesian product)
$d ::= x \in r$		$\mid r \cup r$	(union)
$m ::= \mathsf{no} \mid \mathsf{lone} \mid \mathsf{one}$		$\mid r - r$	(difference)
$n ::= 1 \mid 2 \mid \cdots$		$\mid \mathsf{if}\ \varphi\ \mathsf{then}\ r\ \mathsf{else}\ r$	(conditional)

FORL syntactically distinguishes between terms and formulas. The universe of discourse is $\mathscr{A} = \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}$, where each a_i is an uninterpreted atom. Atoms and n-tuples are identified with singleton sets and singleton n-ary relations, respectively. Bound variables in quantifications and comprehensions range over the tuples in a relation; thus, $\forall x \in (\mathsf{a}_1 \cup \mathsf{a}_2) \times \mathsf{a}_3\colon \varphi(x)$ is equivalent to $\varphi(\mathsf{a}_1 \times \mathsf{a}_3) \wedge \varphi(\mathsf{a}_2 \times \mathsf{a}_3)$.

Although they are not listed above, we will sometimes make use of $\vee$, $\longrightarrow$, $\exists$, *, and $\cap$ in examples. The constraint $\mathsf{no}\ r$ expresses that r is the empty relation, $\mathsf{one}\ r$ expresses that r is a singleton, and $\mathsf{lone}\ r \Longleftrightarrow \mathsf{no}\ r \vee \mathsf{one}\ r$. The projection and dot-join operators are unconventional; their semantics is given by the equations

$$[\![\pi^k_i(r)]\!] = \{(r_i,\ldots,r_{i+k-1}) \mid (r_1,\ldots,r_m) \in [\![r]\!]\}$$
$$[\![r.s]\!] = \{(r_1,\ldots,r_{m-1},s_2,\ldots,s_n) \mid \exists t.\ (r_1,\ldots,r_{m-1},t) \in [\![r]\!] \wedge (t,s_2,\ldots,s_n) \in [\![s]\!]\}.$$

The dot-join operator admits three important special cases. Let s be unary and r, r' be binary relations. The expression $s.r$ gives the direct image of the set s under r; if s is a singleton and r a function, it coincides with the function application $r(s)$. Analogously, $r.s$ gives the inverse image of s under r. Finally, $r.r'$ expresses relational composition.

To pass an n-tuple s to a function r, we write $\langle s\rangle.r$, which stands for the n-fold dot-join $\pi_n(s).(\ldots.(\pi_1(s).r)\ldots)$. We write $\pi_i(r)$ for $\pi^1_i(r)$.

The relational operators often make it possible to express first-order problems concisely. The following Kodkod specification attempts to fit 30 pigeons in 29 holes:

vars $pigeons = \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\}$, $holes = \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}$
var $\emptyset \subseteq nest \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\} \times \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}$
solve $(\forall p \in pigeons\colon \mathsf{one}\ p.nest) \wedge (\forall h \in holes\colon \mathsf{lone}\ nest.h)$

The example declares three free variables: *pigeons* and *holes* are given fixed values, whereas *nest* is specified with a lower and an upper bound. Variable declarations are an extralogical way of specifying sort constraints and partial solutions.

The constraint $\mathsf{one}\ p.nest$ states that pigeon p is in relation with exactly one hole, and $\mathsf{lone}\ nest.h$ that hole h is in relation with at most one pigeon. Taken as a whole, the formula states that *nest* is a one-to-one function. It is, of course, not satisfiable, a fact that Kodkod can establish in less than a second.

When reducing FORL to SAT, each n-ary relational variable y is in principle translated to an $|\mathscr{A}|^n$ array of propositional variables $V[i_1,\ldots,i_n]$, with $V[i_1,\ldots,i_n] \iff \langle \mathsf{a}_{i_1},\ldots,\mathsf{a}_{i_n}\rangle \in y$. Most relational operations can be coded efficiently; for example, $\cup$ is simply $\vee$. The quantified formula $\forall r \in s\colon \varphi(r)$ is treated as $\bigwedge_{j=1}^{n} t_j \subseteq s \longrightarrow \varphi(t_j)$, where the t_j's are the tuples that may belong to s. Transitive closure is unrolled to saturation.

3 The Basic Translation

Nitpick employs Kodkod to find a finite model (a satisfying assignment to the free variables and constants) of $\neg P$, where P is the formula to refute. The translation of a formula from HOL to FORL is parameterized by the cardinalities of the types occurring in it, provided as a function $|\sigma|$ from types to positive integers obeying

$$|\sigma| \geq 1 \qquad |o| = 2 \qquad |\sigma \to \tau| = |\tau|^{|\sigma|} \qquad |\sigma \times \tau| = |\sigma| \cdot |\tau|\,.$$

Following Jackson [13], we call such a function a *scope*. Like other SAT-based model finders, Nitpick enumerates the possible scopes for each basic type, so that if a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

The basic translation presented in this section handles the following HOL constants:

False^{o}	(falsity)	$\mathit{insert}^{\alpha\to(\alpha\to o)\to\alpha\to o}$	(element insertion)
True^{o}	(truth)	$\mathit{UNIV}^{\alpha\to o}$	(universal set)
$\simeq^{\alpha\to\alpha\to o}$	(equality)	$\cup^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$	(union)
$\subseteq^{(\alpha\to o)\to(\alpha\to o)\to o}$	(subset)	$-^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$	(set difference)
$\neg^{o\to o}$	(negation)	$\mathit{Pair}^{\alpha\to\beta\to\alpha\times\beta}$	(pair constructor)
$\wedge^{o\to o\to o}$	(conjunction)	$\mathit{fst}^{\alpha\times\beta\to\alpha}$	(first projection)
$\forall^{(\alpha\to o)\to o}$	(universal quantifier)	$\mathit{snd}^{\alpha\times\beta\to\beta}$	(second projection)
$\emptyset^{\alpha\to o}$	(empty set)	$(\ )^{+\,(\alpha\times\alpha\to o)\to\alpha\times\alpha\to o}$	(transitive closure)

SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas. Whenever practicable, HOL constants should be mapped to their FORL equivalents, rather than expanded to their definitions. This is especially true for the transitive closure r^+, which is defined as the least fixed point of $\lambda R\,(x, y).\ (\exists a\,b.\ x \simeq a \wedge y \simeq b \wedge r\ (a, b)) \vee (\exists a\,b\,c.\ x \simeq a \wedge y \simeq c \wedge R\ (a, b) \wedge r\ (b, c))$.

As a rule, HOL functions should be mapped to FORL relations accompanied by a constraint. For example, assuming the scope $|\alpha| = 2$ and $|\beta| = 3$, the presumptive theorem $\forall x^{\alpha}.\ \exists y^{\beta}.\ f\,x \simeq y$ corresponds to the Kodkod problem

var $\emptyset \subseteq f \subseteq \{\mathsf{a}_1, \mathsf{a}_2\} \times \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}$
solve $(\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \mathsf{one}\ x.f) \wedge \neg\,(\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \exists y \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon x.f \simeq y)$

The first conjunct ensures that f is a function, and the second conjunct is the negation of the HOL formula translated to FORL.

An n-ary first-order function (curried or not) can be coded as an $(n+1)$-ary relation accompanied by a constraint. However, if the return type is o, the function is more efficiently coded as an unconstrained n-ary relation. This allows formulas such as $A^+ \cup B^+ \simeq (A \cup B)^+$ to be translated without taking a detour through ternary relations.

Higher-order quantification and functions bring complications of their own. For example, we would like to translate $\forall g^{\beta\to\alpha}.\ g\,x \not\simeq y$ into something like

$$\forall g \subseteq (\mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5) \times (\mathsf{a}_1 \cup \mathsf{a}_2)\colon (\forall x \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon \mathsf{one}\ x.g) \longrightarrow x.g \not\simeq y,$$

but the $\subseteq$ symbol is not allowed at the binding site; only $\in$ is. Skolemization solves half of the problem (Section 5.1), but for the remaining quantifiers we are forced to adopt an unwieldy n-tuple singleton representation of functions, where n is the cardinality of the domain. For the formula above, this gives

$$\forall G \in (\mathsf{a}_1 \cup \mathsf{a}_2)\times(\mathsf{a}_1 \cup \mathsf{a}_2)\times(\mathsf{a}_1 \cup \mathsf{a}_2)\colon x.\overbrace{\big(\mathsf{a}_3\times\pi_1(G)\ \cup\ \mathsf{a}_4\times\pi_2(G)\ \cup\ \mathsf{a}_5\times\pi_3(G)\big)}^{g} \not\simeq y,$$

where G is the triple corresponding to g. In the body, we convert the singleton G to the relational representation, then we apply x on it using dot-join. The singleton encoding is also used for passing functions to functions; fortunately, two optimizations, function specialization and boxing (Section 5.1), make this rarely necessary.

We are now ready to look at the basic translation in more detail. The translation distinguishes between formulas (F), singletons (S), and relations (R). We start by mapping HOL types to sets of FORL atom tuples. For each type σ, we provide two codings, a singleton representation $\mathsf{S}\llbracket\sigma\rrbracket$ and a relational representation $\mathsf{R}\llbracket\sigma\rrbracket$:[2]

$$\begin{aligned}
\mathsf{S}\llbracket\sigma\to\tau\rrbracket &= \mathsf{S}\llbracket\tau\rrbracket^{|\sigma|} & \mathsf{R}\llbracket\sigma\to o\rrbracket &= \mathsf{S}\llbracket\sigma\rrbracket\\
\mathsf{S}\llbracket\sigma\times\tau\rrbracket &= \mathsf{S}\llbracket\sigma\rrbracket\times\mathsf{S}\llbracket\tau\rrbracket & \mathsf{R}\llbracket\sigma\to\tau\rrbracket &= \mathsf{S}\llbracket\sigma\rrbracket\times\mathsf{R}\llbracket\tau\rrbracket\\
\mathsf{S}\llbracket\sigma\rrbracket &= \{\mathsf{a}_1,\ldots,\mathsf{a}_{|\sigma|}\} & \mathsf{R}\llbracket\sigma\rrbracket &= \mathsf{S}\llbracket\sigma\rrbracket.
\end{aligned}$$

In the S representation, an element of type σ is mapped to a single tuple $\in \mathsf{S}\llbracket\sigma\rrbracket$. In the R representation, an element of type $\sigma\to o$ is mapped to a subset of $\mathsf{S}\llbracket\sigma\rrbracket$ consisting of the points at which the predicate is *True*; an element of $\sigma\to\tau$ (where $\tau\neq o$) is mapped to a relation $\subseteq \mathsf{S}\llbracket\sigma\rrbracket\times\mathsf{R}\llbracket\tau\rrbracket$; any other element is coded as a singleton. For simplicity, we reuse the same atoms for distinct types. Doing so is sound for well-typed terms.

For each free variable y^σ, we generate the declaration **var** $\emptyset \subseteq y \subseteq \mathsf{R}\llbracket\sigma\rrbracket$ as well as a constraint $\Phi^\sigma(y)$ to ensure that functions are functions and single values are singletons:

$$\Phi^{\sigma_1\to\cdots\to\sigma_n\to o}(r) = \mathsf{true} \qquad \Phi^{\sigma\to\tau}(r) = \forall b_\mathrm{f}\in\mathsf{S}\llbracket\sigma\rrbracket\colon \Phi^\tau\big(\langle b_\mathrm{f}\rangle.r\big) \qquad \Phi^\sigma(r) = \mathsf{one}\ r.$$

We assume that free and bound variables are syntactically distinguishable, and use the letter y for the former and b for the latter. The symbol b_f denotes a fresh bound variable.

We assume a total order on n-tuples of atoms and let $\mathsf{S}_i\llbracket\sigma\rrbracket$ denote the ith tuple from $\mathsf{S}\llbracket\sigma\rrbracket$ according to that order. Furthermore, we define $s(\sigma)$ and $r(\sigma)$ as the arity of the tuples in $\mathsf{S}\llbracket\sigma\rrbracket$ and $\mathsf{R}\llbracket\sigma\rrbracket$, respectively. The translation of terms requires the following rather technical conversions between singletons (S), relations (R), and formulas (F):

$$\begin{aligned}
&s2r^{\sigma\to o}(r) = \textstyle\bigcup_{i=1}^{|\sigma|}\pi_i(r).(\mathsf{a}_2\times\mathsf{S}_i\llbracket\sigma\rrbracket) && f2s(\varphi) = \mathsf{if}\ \varphi\ \mathsf{then}\ \mathsf{a}_2\ \mathsf{else}\ \mathsf{a}_1\\
&s2r^{\sigma\to\tau}(r) = \textstyle\bigcup_{i=1}^{|\sigma|}\mathsf{S}_i\llbracket\sigma\rrbracket\times s2r^\tau\Big(\pi^{s(\tau)}_{(i-1)\cdot s(\tau)+1}(r)\Big) && s2f(r) = r\simeq\mathsf{a}_2\\
&r2s^{\sigma\to\tau}(r) = \{\langle b_\mathrm{f}\in\mathsf{S}\llbracket\sigma\to\tau\rrbracket\rangle \mid s2r^{\sigma\to\tau}(b_\mathrm{f})\simeq r\} && s2r^\sigma(r) = r \qquad r2s^\sigma(r) = r.
\end{aligned}$$

The Boolean values false and true are arbitrarily coded as a_1 and a_2, respectively.

[2] Metatheoretic functions here and elsewhere are defined using sequential pattern matching, eliminating the need for side conditions such as "if $\tau\neq o$" and "otherwise."

The translation of HOL terms is performed by three functions, $\mathsf{F}\langle\!\langle t \rangle\!\rangle$, $\mathsf{S}\langle\!\langle t \rangle\!\rangle$, and $\mathsf{R}\langle\!\langle t \rangle\!\rangle$. Their defining equations are to be matched modulo η-equivalence:

$$\begin{array}{rcl}
\mathsf{F}\langle\!\langle y \rangle\!\rangle = s2f(y) & \mathsf{F}\langle\!\langle t \subseteq u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle u \rangle\!\rangle & \mathsf{S}\langle\!\langle b \rangle\!\rangle = b \\
\mathsf{F}\langle\!\langle b \rangle\!\rangle = s2f(b) & \mathsf{F}\langle\!\langle \neg t \rangle\!\rangle = \neg \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle t \rangle\!\rangle \times \mathsf{S}\langle\!\langle u \rangle\!\rangle \\
\mathsf{F}\langle\!\langle \mathit{False} \rangle\!\rangle = \mathsf{false} & \mathsf{F}\langle\!\langle t \wedge u \rangle\!\rangle = \mathsf{F}\langle\!\langle t \rangle\!\rangle \wedge \mathsf{F}\langle\!\langle u \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{fst}\ t^{\sigma\times\tau} \rangle\!\rangle = \pi_1^{s(\sigma)}(\mathsf{S}\langle\!\langle t \rangle\!\rangle) \\
\mathsf{F}\langle\!\langle \mathit{True} \rangle\!\rangle = \mathsf{true} & \mathsf{F}\langle\!\langle \forall b^{\sigma}.\ t \rangle\!\rangle = \forall b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle\colon \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{snd}\ t^{\sigma\times\tau} \rangle\!\rangle = \pi_{s(\sigma)+1}^{s(\tau)}(\mathsf{S}\langle\!\langle t \rangle\!\rangle) \\
\mathsf{F}\langle\!\langle t \simeq u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{R}\langle\!\langle u \rangle\!\rangle & \mathsf{F}\langle\!\langle t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle u \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle t \rangle\!\rangle = r2s^{\sigma}\big(\mathsf{R}\langle\!\langle t \rangle\!\rangle\big)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{R}\langle\!\langle c^{o} \rangle\!\rangle = f2s(c) & \mathsf{R}\langle\!\langle \mathit{insert}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle t \rangle\!\rangle \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle y \rangle\!\rangle = y & \mathsf{R}\langle\!\langle t \cup u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle b^{\sigma} \rangle\!\rangle = s2r^{\sigma}(b) & \mathsf{R}\langle\!\langle t - u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle - \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle & \mathsf{R}\langle\!\langle (t^{\sigma\times\sigma\to o})^{+} \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle^{+} \quad \text{if } r(\sigma) = 1 \\
\mathsf{R}\langle\!\langle \mathit{fst}\ t^{\sigma\times\tau} \rangle\!\rangle = s2r^{\sigma}\big(\mathsf{S}\langle\!\langle \mathit{fst}\ t \rangle\!\rangle\big) & \mathsf{R}\langle\!\langle t^{\sigma\to o}\ u \rangle\!\rangle = f2s\big(\mathsf{F}\langle\!\langle t\ u \rangle\!\rangle\big) \\
\mathsf{R}\langle\!\langle \mathit{snd}\ t^{\sigma\times\tau} \rangle\!\rangle = s2r^{\tau}\big(\mathsf{S}\langle\!\langle \mathit{snd}\ t \rangle\!\rangle\big) & \mathsf{R}\langle\!\langle t\ u \rangle\!\rangle = \langle \mathsf{S}\langle\!\langle u \rangle\!\rangle \rangle.\mathsf{R}\langle\!\langle t \rangle\!\rangle \\
\mathsf{R}\langle\!\langle \emptyset^{\sigma} \rangle\!\rangle = \mathsf{none}^{r(\sigma)} & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{o} \rangle\!\rangle = \{\langle b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle \rangle \mid \mathsf{F}\langle\!\langle t \rangle\!\rangle\} \\
\mathsf{R}\langle\!\langle \mathit{UNIV}^{\sigma} \rangle\!\rangle = \mathsf{R}\langle\!\langle \sigma \rangle\!\rangle & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{\tau} \rangle\!\rangle = \{\langle b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle, b_{\mathrm{f}} \in \mathsf{R}\langle\!\langle \tau \rangle\!\rangle \rangle \mid b_{\mathrm{f}} \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle\}.
\end{array}$$

Annoyingly, the translation of transitive closure is defined only if $r(\sigma) = 1$. We will see ways to lift this restriction in the next two sections.

Theorem 1 (Soundness). *Given a putative theorem P with free variables $y_1^{\sigma_1}, \ldots, y_n^{\sigma_n}$ within our HOL fragment and a scope S, P admits a counterexample if there exists a valuation V with $V(y_j) \subseteq \mathsf{R}\langle\!\langle \sigma_j \rangle\!\rangle$ that satisfies the FORL formula $\mathsf{F}\langle\!\langle \neg P \rangle\!\rangle \wedge \bigwedge_{j=1}^{n} \Phi^{\sigma_j}(y_j)$.*

Proof sketch. Let $[\![t]\!]_A$ denote the set-theoretic semantics of the HOL term t w.r.t. a variable assignment A and the scope S. Let $[\![\rho]\!]_V$ denote the semantics of the FORL term or formula ρ w.r.t. a variable valuation V and the scope S. Furthermore, let $\lfloor v \rfloor_X$ denote the X-encoded FORL value corresponding to the HOL value v, for $X \in \{\mathsf{F}, \mathsf{S}, \mathsf{R}\}$. Using recursion induction, it is straightforward to prove that $[\![X\langle\!\langle t \rangle\!\rangle]\!]_V = \lfloor [\![t]\!]_A \rfloor_X$ if $V(y_i) = \lfloor A(y_i) \rfloor_{\mathsf{R}}$ for all free variables y_i and $V(b_i) = \lfloor A(b_i) \rfloor_{\mathsf{S}}$ for all locally free bound variables b_i occurring in t. Moreover, from the satisfying valuation V of the free variables y_i, we can construct a type-correct HOL assignment A such that $\lfloor A(y_i) \rfloor_{\mathsf{R}} = V(y_i)$; the $\Phi^{\sigma_j}(y_j)$ constraints and the variable bounds $V(y_j) \subseteq \mathsf{R}\langle\!\langle \sigma_j \rangle\!\rangle$ ensure that such an assignment exists. Hence, $[\![\mathsf{F}\langle\!\langle \neg P \rangle\!\rangle]\!]_V = \mathsf{true} = \lfloor [\![\neg P]\!]_A \rfloor_{\mathsf{F}}$, which shows that A falsifies P.

A very thorough soundness proof of a translation from HOL to SAT can be found in Tjark Weber's Ph.D. thesis [23].

4 Refinements to the Basic Translation

4.1 Approximation of Infinite Types and Partiality

Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers finite subsets $\{0, 1, \ldots, K-1\}$

of *nat* and maps numbers $\geq K$ to the undefined value ($\bot$), coded as none, the empty set. Formulas of the form $\forall n^{nat}.\ P(n)$ are treated in essence as $(\forall n < K.\ P(n)) \wedge P(\bot)$, which usually evaluates to either *False* (if $P(i)$ gives *False* for some $i < K$) or $\bot$, but not to *True*, since we do not know whether $P(K)$, $P(K+1)$, ... (collectively represented by $P(\bot)$) are true. In view of this, Nitpick generally cannot soundly disprove conjectures that contain an infinite existential quantifier in their conclusion or an infinite universal quantifier in their assumptions. As a fallback, the tool enters an unsound mode in which the quantifiers are artificially bounded. Counterexamples obtained under these conditions are marked as "potential."

Functions from *nat* to α are abstracted by relations $\subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_K\} \times \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\}$ constrained to be partial functions. Partiality makes it possible to encode the successor function *Suc* as the relation $S = (\mathsf{a}_1 \times \mathsf{a}_2) \cup \cdots \cup (\mathsf{a}_{K-1} \times \mathsf{a}_K)$, which associates no value with a_K. Conveniently, the dot-join $\mathsf{a}_K.S$ yields none, and so does none.S. This is desirable because *Suc* $(K-1)$ is unrepresentable and *Suc* $\bot$ is unknown.

Partiality leads to a Kleene three-valued logic, which is expressed in terms of Kodkod's two-valued logic as follows. At the outermost level, we let the FORL truth value false stand for both *False* (no counterexample) and $\bot$ (potential counterexample), and reserve true for *True* (genuine counterexample). The same convention is obeyed in other positive contexts within the formula. In negative contexts, false codes *False* and true codes *True* or $\bot$. Finally, in unpolarized contexts (for example, as argument to a function), the atom a_1 codes *False*, a_2 codes *True*, and none codes $\bot$. Unlike similar approximation approaches [18, p. 164; 23], Nitpick's logic is sound, although the tool also has an unsound mode as noted above.

4.2 Nonuniform Representation of HOL Terms

FORL gives Nitpick a lot of flexibility when encoding terms. A value of type $\alpha \times \beta$, for example, can be translated as before to a pair $\in \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\} \times \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\beta|}\}$, but it can also be mapped to a single atom $\in \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha \times \beta|}\}$. Predicates on α (or functions from α to σ with $|\sigma| = 2$) can be coded as single atoms, sets of atoms, relations from atoms to $\{\mathsf{a}_1, \mathsf{a}_2\}$, or $|\alpha|$-tuples over $\{\mathsf{a}_1, \mathsf{a}_2\}$.

Nitpick uses FORL's flexibility to a larger extent than was hinted at in Section 3. For example, it ensures that the operand of transitive closure is always a binary relation (a set of pairs), no matter what the HOL type is, lifting an annoying limitation in the basic translation described earlier. It also keeps track of whether a term can evaluate to $\bot$, which makes many optimizations possible in FORL. For example, because free variables never yield $\bot$, we encode $x \simeq y$ as $x \subseteq y$, which is more efficient.

The current representation selection scheme proceeds in a straightforward bottom-up fashion, inserting conversions as appropriate. More sophisticated schemes that would minimize the number of conversions have yet to be tried.

4.3 Encoding of (Co)inductive Predicates

Isabelle lets users specify (co)inductive predicates p by their introduction rules and synthesizes a fixed point definition $p \simeq \mathit{lfp}\ F$ or $p \simeq \mathit{gfp}\ F$. For performance reasons,

Nitpick avoids expanding *lfp* and *gfp* to their definitions and translates (co)inductive predicates directly, using appropriate FORL concepts.

A first intuition is that an inductive predicate p is a fixed point, so we could use the equation $p \simeq F\ p$ as the axiomatic specification of p. In general, this is unsound since it underspecifies p, but there are two important cases for which this method is sound. First, if the recursion in F is well-founded, the fixed point equation $p \simeq F\ p$ admits exactly one solution and we can safely use it as p's specification. Second, if p occurs negatively in the formula, we can replace these occurrences by a fresh constant q satisfying the axiom $q \simeq F\ q$; this transformation preserves equisatisfiability.

To deal with positive occurrences of p, we adapt a technique from bounded model checking [5]: We replace p by a fresh predicate r_k defined by

$$r_0 \simeq (\lambda \bar{x}.\ \bot) \qquad\qquad r_{n+1} \simeq F\ r_n,$$

which corresponds to p unrolled k times. For unpolarized occurrences, we use $q \cap r_k$. In essence, we have made p well-founded by adding a counter that decreases by one with each recursive call. This unrolling comes at a price: The search space and the size of the propositional formula for r_k is k times that of q. Hence, it makes sense to look for a counterexample with a small value of k first and increment it gradually if needed.

The situation is mirrored for coinductive predicates: Negative occurrences of p become r_k, positive occurrences become q, and unpolarized occurrences become $q \cup r_k$.

To determine whether a predicate is well-founded, Nitpick generates a wellfoundedness goal and invokes Isabelle's termination prover [7] with a time limit. Given introduction rules of the form

$$\frac{p\ \bar{t}_{i1} \quad \cdots \quad p\ \bar{t}_{in_i} \quad Q_i}{p\ \bar{u}_i}$$

for $i \in \{1,\ldots,m\}$, the termination prover must exhibit a well-founded relation R such that $\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n_i} Q_i \longrightarrow \langle \bar{t}_{ij}, \bar{u}_i \rangle \in R$ holds.

In our experience, about half of the inductive predicates occurring in practice are well-founded—this includes most type systems and other compositional formalisms, but generally excludes state transition systems.

4.4 Encoding of (Co)inductive Datatypes and (Co)recursive Functions

In contrast to Isabelle's constructor-oriented treatment of inductive datatypes, Nitpick's FORL axiomatization revolves around selectors and discriminators, inspired by Kuncak and Jackson's modeling of lists and trees in Alloy [14]. The selector and discriminator view is usually more efficient than the constructor view because it breaks high-arity constructors into several low-arity selectors.

Consider the type α *list* generated from $Nil^{\alpha\ list}$ and $Cons^{\alpha \to \alpha\ list \to \alpha\ list}$. The FORL axiomatization is done in terms of the discriminators $isNil^{\alpha\ list \to o}$ and $isCons^{\alpha\ list \to o}$ and the selectors $get1Cons^{\alpha\ list \to \alpha}$ and $get2Cons^{\alpha\ list \to \alpha\ list}$, which give access to a nonempty list's head and tail. Following Dunets et al. [10], *Nil* and *Cons x xs* are translated as *isNil* and $get1Cons.x \cap get2Cons.xs$, respectively.

The following axioms, with $N = 1, 2$, specify a subterm-closed finite universe of lists using the atoms $\mathscr{A}_{\alpha\ list}$:

DISJ: no $isNil \cap isCons$

EXH: $isNil \cup isCons \simeq \mathscr{A}_{\alpha\ list}$

SEL$_N$: $\forall xs \in \mathscr{A}_{\alpha\ list}$: if $xs \subseteq isCons$ then one $xs.getNCons$ else no $xs.getNCons$

UNIQ: lone $isNil \wedge (\forall x \in \mathscr{A}_\alpha, xs \in \mathscr{A}_{\alpha\ list}$: lone $get1Cons.x \cap get2Cons.xs)$

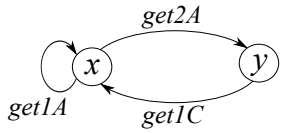
ACYCL: no $get2Cons^+ \cap$ iden.

Examples of subterm-closed list universes using traditional list notation are $\{[], [a_1], [a_2], [a_3]\}$ and $\{[], [a_2], [a_3, a_2], [a_1, a_3, a_2]\}$. For recursive functions, Nitpick ignores the construction synthesized by Isabelle and relies instead on the user-specified equations.

The approach can be generalized to mutually recursive datatypes. To generate the ACYCL axioms for the mutually recursive datatypes x with constructors $A^{x \to y \to x}$ and B^x and y with constructor $C^{x \to y}$, we compute their datatype dependency graph, in which vertices are labeled with datatypes and arcs are labeled with selectors. Then we compute for each datatype a regular expression capturing the nontrivial paths from the datatype to itself, with $.$ standing for concatenation, $\cup$ for alternative, and * and $^+$ for repetition. We require the paths to be disjoint from identity:

no $(get1A \cup get2A.get1C)^+ \cap$ iden

no $(get1C.get1A^*.get2A)^+ \cap$ iden.



Nitpick supports coinductive datatypes, even though Isabelle does not provide a high-level mechanism for defining them. Users can define custom coinductive datatypes from first principles and tell Nitpick to substitute its efficient FORL axiomatization for their definitions. Nitpick also knows about Isabelle's coinductive "lazy list" datatype, *α llist*, with the constructors $LNil^{\alpha\ llist}$ and $LCons^{\alpha \to \alpha\ llist \to \alpha\ llist}$. The FORL axiomatization is similar to that used for *α list*, but the ACYCL axiom is omitted to allow cyclic (ω-regular) lists. Infinite lists are presented to the user as lassos, with a finite stem and cycle. The following coinductive bisimulation principle is translated along with the HOL formula, to ensure that distinct atoms correspond to observably distinct lists:

$$\frac{}{LNil \simeq LNil}\ \text{BISIM}_1 \qquad \frac{x \simeq y \quad xs \simeq ys}{LCons\ x\ xs \simeq LCons\ y\ ys}\ \text{BISIM}_2.$$

5 Optimization Steps

5.1 HOL Preprocessing

Function Specialization. A function argument is said to be static if it is passed unaltered to all recursive calls. A typical example is f in the definition of *map*:

$$map\ f\ [] \simeq [] \qquad map\ f\ (x \cdot xs) \simeq f\ x \cdot map\ f\ xs.$$

An optimization reminiscent of the static argument transformation or lambda-dropping [9, pp. 148–156] is to specialize the function for each eligible call site, thereby avoiding passing the static argument altogether. At the call site, any term whose free variables are all globally free is eligible for this optimization. Following this scheme, *map Suc ns* would become $map_{Suc}\ ns$, where map_{Suc} is defined as follows:

$$map_{Suc}\ [] \simeq [] \qquad\qquad map_{Suc}\ (x \cdot xs) \simeq Suc\ x \cdot map_{Suc}\ xs.$$

For this example, specialization reduces the number of propositional variables needed to encode the function by a factor of $|nat|^{|nat|}$.

Boxing. Nitpick normally translates function and product types directly to the homologous Kodkod concepts. This is not always desirable; for example, a transition relation on states represented as n-tuples leads to a $2n$-ary relation, which gives rise to a combinatorial explosion and precludes the use of FORL's binary transitive closure.

Our experience suggests that it is almost always advantageous to approximate n-tuples where $n \geq 3$ as well as higher-order arguments. This is achieved by wrapping them in an isomorphic type $\alpha\ box$ with the single constructor $Box^{\alpha \to \alpha\ box}$, inserting constructors and selectors as appropriate. Assuming that specialization is not in use, the second equation for *map* would then become

$$map\ f^{(nat \to nat)\ box}\ (x \cdot xs) \simeq get1Box\ f\ x \cdot map\ f\ xs,$$

with $map\ (Box\ Suc)\ ns$ at the call site. Notice that for function types, boxing is similar to defunctionalization [2], with selectors playing the role of "apply" functions. Further opportunities for boxing are created by uncurrying high-arity constants beforehand.

Quantifier Massaging. (Co)inductive definitions are marred by existential quantifiers, which blow up the size of the resulting propositional formula. The following steps are applied to eliminate quantifiers or reduce their binding range: (1) Replace quantifications of the forms $\forall x.\ x \simeq t \longrightarrow P(x)$ and $\exists x.\ x \simeq t \wedge P(x)$ by $P(t)$ if x does not occur free in t. (2) Skolemize. (3) Distribute quantifiers over congenial connectives ($\forall$ over $\wedge$, $\exists$ over $\vee$ and $\longrightarrow$). (4) For any remaining subformula $Qx_1 \ldots x_n.\ p_1 \otimes \cdots \otimes p_m$, where Q is a quantifier and $\otimes$ is a connective, move the p_i's out of as many quantifiers as possible by rebuilding the formula using $qfy(\{x_1, \ldots, x_n\}, \{p_1, \ldots, p_m\})$, defined as

$$qfy(\emptyset, P) = \bigotimes P \qquad\qquad qfy(x \uplus X, P) = qfy(X, P - P_x \cup \{Qx.\ \bigotimes P_x\}),$$

where $P_x = \{p \in P \mid x \text{ occurs free in } p\}$.

The order in which individual variables x are removed from the first argument is crucial because it affects which p_i's can be moved out. For clusters of up to 7 quantifiers, Nitpick considers all permutations of the bound variables and chooses the one that minimizes the sum $\sum_{i=1}^{m} |\tau_{i1}| \cdot \ldots \cdot |\tau_{ik_i}| \cdot size(p_i)$, where $\tau_{i1}, \ldots, \tau_{ik_i}$ are the types of the variables that have p_i in their binding range, and $size(p_i)$ is a rough syntactic measure of p_i's size; for larger clusters, it falls back on a heuristic inspired by Paradox's clause splitting procedure [8]. Thus, the formula $\exists x^{\alpha}\ y^{\alpha}.\ p\ x \wedge q\ x\ y \wedge r\ y\ (f\ y\ y)$ is transformed into $\exists y^{\alpha}.\ r\ y\ (f\ y\ y) \wedge (\exists x^{\alpha}.\ p\ x \wedge q\ x\ y)$. Processing y before x in *qfy* would instead give $\exists x^{\alpha}.\ p\ x \wedge (\exists y^{\alpha}.\ q\ x\ y \wedge r\ y\ (f\ y\ y))$, which is more expensive because $r\ y\ (f\ y\ y)$, the most complex conjunct, is doubly quantified and hence $|\alpha|^2$ copies of it are needed in the resulting propositional formula.

Constructor Elimination. Since datatype constructors may return $\bot$ in our encoding, we can increase precision by eliminating them. A formula such as $[x,y] \simeq [a,b]$ can easily be rewritten into $x \simeq a \wedge y \simeq b$, which evaluates to either *True* or *False* even if $[x,y]$ or $[a,b]$ would yield $\bot$.

For multiple-argument constructors, eliminating constructors helps reduce the number of nested quantifiers. Consider a datatype of AVL trees with two constructors, $Null^{\alpha\ tree}$ and $Node^{\alpha \to \alpha\ tree \to \alpha\ tree \to nat \to \alpha\ tree}$, and a *data* constant defined by the equations

$$data\ Null \simeq \emptyset \qquad \forall a\ t_1\ t_2\ h.\ data\ (Node\ a\ t_1\ t_2\ h) \simeq \{a\} \cup data\ t_1 \cup data\ t_2.$$

Our target is the constructor application $Node\ a\ t_1\ t_2\ h$ in the second equation's left-hand side. We first pull it out and assign it to a fresh bound variable y:

$$\forall a\ t_1\ t_2\ h\ y.\ y \simeq Node\ a\ t_1\ t_2\ h \longrightarrow data\ y \simeq \{a\} \cup data\ t_1 \cup data\ t_2.$$

Then we express the constructor arguments in terms of selectors in the conclusion, rewrite the assumption to use a discriminator, and omit the obsolete variables:

$$\forall y.\ isNode\ y \longrightarrow data\ y \simeq \{get1Node\ y\} \cup data\ (get2Node\ y) \cup data\ (get3Node\ y).$$

By quantifying over a single variable, we reduce the number of copies of the body from $|\alpha| \cdot |\alpha\ tree|^2 \cdot |nat|$ to $|\alpha\ tree|$ in the SAT problem, without losing counterexamples. This technique is also useful for constructors taking a single higher-order argument, such as those inserted by the boxing optimization described above.

5.2 Monotonicity Inference

Many formulas occurring in practice are monotonic in the sense that if the formula is falsifiable for a given scope, it is also falsifiable for all larger scopes [13, p. 165]. That not all formulas are monotonic will become clear after considering $|UNIV| = 3$.

Monotonicity can be exploited to prune the search space. For a formula involving n uninterpreted types, a model finder must a priori consider k^n scopes to exhaust all models up to the cardinality bound k. With monotonicity, it is sufficient to consider the single scope in which all types have cardinality k.

We developed and implemented two calculi for inferring monotonicity, and proved them sound [6]. The first calculus, on which we focus here, has limited support for sets encoded as predicates. The second, more powerful calculus addresses this problem by annotating function arrows and relying on a SAT solver to ensure consistent annotations.

For simplicity of exposition, the first calculus is defined for a HOL fragment in which the only constants are $\simeq$ and $\longrightarrow$. We let *True* abbreviate $(\lambda x^o.\ x) \simeq (\lambda x.\ x)$ and $\forall x.\ p$ abbreviate $(\lambda x.\ p) \simeq (\lambda x.\ True)$. We assume a distinguished type variable α with respect to which monotonicity is inferred. The calculus is defined below:

$$\mathbf{TV}^s(o) = \emptyset \qquad \mathbf{TV}^+(\beta) = \{\beta\} \qquad \mathbf{TV}^-(\beta) = \emptyset \qquad \mathbf{TV}^s(\sigma \to \tau) = \mathbf{TV}^{-s}(\sigma) \cup \mathbf{TV}^s(\tau)$$

$$\frac{}{\mathbf{K}(x)} \qquad \frac{}{\mathbf{K}(\longrightarrow)} \qquad \frac{\alpha \notin \mathbf{TV}^-(\sigma)}{\mathbf{K}(\simeq^{\sigma \to \sigma \to o})} \qquad \frac{\mathbf{K}(t) \quad \mathbf{K}(u)}{\mathbf{K}(t\ u)} \qquad \frac{\mathbf{K}(t)}{\mathbf{K}(\lambda x.\ t)}$$

$$\frac{\mathbf{K}(t)}{\mathbf{M}^s(t)} \quad \frac{\mathbf{M}^{-s}(t) \quad \mathbf{M}^s(u)}{\mathbf{M}^s(t \longrightarrow u)} \quad \frac{\mathbf{M}^+(t) \quad \alpha \notin \mathbf{TV}^+(\sigma)}{\mathbf{M}^+(\forall x^\sigma.\, t)} \quad \frac{\mathbf{M}^-(t)}{\mathbf{M}^-(\forall x.\, t)} \quad \frac{\mathbf{K}(t) \quad \mathbf{K}(u)}{\mathbf{M}^-(t \simeq u)}.$$

The $\mathbf{TV}^s(\sigma)$ function gives the set of type variables occurring positively (if s is $+$) or negatively (if s is $-$) with respect to $\rightarrow$. The judgment $\mathbf{K}(t)$ expresses that t's value remains essentially the same when α's cardinality is increased, assuming that the free variables also stay the same. A formula P is monotonic if $\mathbf{M}^-(P)$ is derivable.

We evaluated both calculi on the theorems from six highly polymorphic Isabelle theories (*AVL2*, *Fun*, *Huffman*, *List*, *Map*, and *Relation*). We found that the simple calculus inferred monotonicity for 41% to 97% of the theorems depending on the theory, while the more sophisticated calculus achieved 65% to 100% [6].

6 Case Studies

6.1 Volpano–Smith–Irvine Security Type System

Assuming a partition of program variables into public and private ones, Volpano, Smith, and Irvine [22] provide typing rules guaranteeing that the contents of private variables stay private. They define two types, *High* (private) and *Low* (public). An expression is *High* if it involves private variables; otherwise it is *Low*. A command is *High* if it modifies private variables only; commands that could alter public variables are *Low*.

As our first case study, we consider a fragment of the formal soundness proof by Snelting and Wasserrab [19]. Given a variable partition Γ, the inductive predicate $\Gamma \vdash e : \sigma$ tells whether e has type σ, whereas $\Gamma, \sigma \vdash c$ tells whether command c has type σ. Below is a flawed definition of $\Gamma, \sigma \vdash c$:

$$\frac{}{\Gamma, \sigma \vdash \texttt{skip}} \quad \frac{\Gamma\, v \simeq \lfloor \mathit{High} \rfloor}{\Gamma, \sigma \vdash v \texttt{ := } e} \quad \frac{\Gamma \vdash e : \mathit{Low} \quad \Gamma\, v \simeq \lfloor \mathit{Low} \rfloor}{\Gamma, \mathit{Low} \vdash v \texttt{ := } e} \quad \frac{\Gamma, \sigma \vdash c_1}{\Gamma, \sigma \vdash c_1 \texttt{ ; } c_2}$$

$$\frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c_1 \quad \Gamma, \sigma \vdash c_2}{\Gamma, \sigma \vdash \texttt{if (}b\texttt{) } c_1 \texttt{ else } c_2} \quad \frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c}{\Gamma, \sigma \vdash \texttt{while (}b\texttt{) } c} \quad \frac{\Gamma, \mathit{High} \vdash c}{\Gamma, \mathit{Low} \vdash c}.$$

The following theorem constitutes a key step in the soundness proof:

$$\Gamma, \mathit{High} \vdash c \wedge \langle c, s \rangle \rightsquigarrow^* \langle \texttt{skip}, s' \rangle \longrightarrow \forall v.\, \Gamma\, v \simeq \lfloor \mathit{Low} \rfloor \longrightarrow s\, v \simeq s'\, v.$$

Informally, it asserts that if executing the *High* command c in state s terminates in state s', then the public variables of s and s' must agree. This is consistent with our intuition that *High* commands should only modify private variables. However, because we planted a bug in the definition of $\Gamma, \sigma \vdash c$, Nitpick finds a counterexample:

$$\Gamma = [\mathtt{v}_1 \mapsto \mathit{Low}] \qquad s = [\mathtt{v}_1 \mapsto \mathit{false}]$$
$$c = \texttt{skip ; } \mathtt{v}_1 \texttt{ := } (\mathit{Var}\, \mathtt{v}_1 \texttt{ == } \mathit{Var}\, \mathtt{v}_1) \qquad s' = [\mathtt{v}_1 \mapsto \mathit{true}].$$

Even though the command c has type *High*, it assigns *true* to the *Low* variable $\mathtt{v}_1$. The bug is a missing assumption $\Gamma, \sigma \vdash c_2$ in the typing rule for sequential composition.

6.2 Hotel Key Card System

We consider a state-based model of a vulnerable hotel key card system with recordable locks [16], inspired by an Alloy specification due to Jackson [13, pp. 299–306]. The formalization relies on three opaque types, *room*, *guest*, and *key*. A key card, of type $card = key \times key$, combines an old key and a new key. A state is a 7-field record $(\!|\mathsf{owns} :: room \to guest\ option, \mathsf{curr} :: room \to key, \mathsf{issued} :: key \to o, \mathsf{cards} :: guest \to card \to o, \mathsf{roomk} :: room \to key, \mathsf{isin} :: room \to guest \to o, \mathsf{safe} :: room \to o|\!)$. The set *reach* of reachable states is defined inductively by the following rules:

$$\frac{inj\ init}{(\!|\mathsf{owns} = (\lambda r.\ \bot),\ \mathsf{curr} = init,\ \mathsf{issued} = range\ init,\ \mathsf{cards} = (\lambda g.\ \emptyset),\ \mathsf{roomk} = init,\ \mathsf{isin} = (\lambda r.\ \emptyset),\ \mathsf{safe} = (\lambda r.\ True)|\!) \in reach}\ \textsc{Init}$$

$$\frac{s \in reach \quad k \notin issued\ s}{s(\!|\mathsf{curr} := (curr\ s)(r := k),\ \mathsf{issued} := issued\ s \cup k,\ \mathsf{cards} := (cards\ s)(g := cards\ s\ g \cup \langle curr\ s\ r, k\rangle),\ \mathsf{owns} := (owns\ s)(r := \lfloor g \rfloor),\ \mathsf{safe} := (safe\ s)(r := False)|\!) \in reach}\ \textsc{Check-In}$$

$$\frac{s \in reach \quad \langle k, k'\rangle \in cards\ s\ g \quad roomk\ s\ r \in \{k, k'\}}{s(\!|\mathsf{isin} := (isin\ s)(r := isin\ s\ r \cup g),\ \mathsf{roomk} := (roomk\ s)(r := k'),\ \mathsf{safe} := (safe\ s)(r := owns\ s\ r \simeq \lfloor g \rfloor \wedge isin\ s\ r \simeq \emptyset \vee safe\ s\ r)|\!) \in reach}\ \textsc{Entry}$$

$$\frac{s \in reach \quad g \in isin\ s\ r}{s(\!|\mathsf{isin} := (isin\ s)(r := isin\ s\ r - \{g\})|\!) \in reach}\ \textsc{Exit}.$$

A desirable property of the system is that it should prevent unauthorized access:

$$s \in reach \wedge safe\ s\ r \wedge g \in isin\ s\ r \longrightarrow owns\ s\ r \simeq \lfloor g \rfloor.$$

Nitpick needs some help to contain the state space explosion: We restrict the search to one room and two guests. Within seconds, we get the counterexample

$$\begin{aligned} s = (\!|&\mathsf{owns} = (\mathsf{r}_1 := \lfloor \mathsf{g}_1 \rfloor),\ \mathsf{curr} = (\mathsf{r}_1 := \mathsf{k}_1),\ \mathsf{issued} = \{\mathsf{k}_1, \mathsf{k}_2, \mathsf{k}_3, \mathsf{k}_4\},\\ &\mathsf{cards} = (\mathsf{g}_1 := \{\langle \mathsf{k}_3, \mathsf{k}_1\rangle, \langle \mathsf{k}_4, \mathsf{k}_2\rangle\},\ \mathsf{g}_2 := \{\langle \mathsf{k}_2, \mathsf{k}_3\rangle\}),\ \mathsf{roomk} = (\mathsf{r}_1 := \mathsf{k}_3),\\ &\mathsf{isin} = (\mathsf{r}_1 := \{\mathsf{g}_1, \mathsf{g}_2\}),\ \mathsf{safe} = \{\mathsf{r}_1\}|\!) \end{aligned}$$

with $g = \mathsf{g}_2$ and $r = \mathsf{r}_1$.

To retrace the steps from the initial state to the s, we can ask Nitpick to show the interpretation of *reach* at each iteration. This reveals the following "guest in the middle" attack: (1) Guest g_1 checks in and gets a card $\langle \mathsf{k}_4, \mathsf{k}_2\rangle$ for room r_1, whose lock expects k_4. Guest g_1 does not enter the room yet. (2) Guest g_2 checks in, gets a card $\langle \mathsf{k}_2, \mathsf{k}_3\rangle$ for r_1, and waits. (3) Guest g_1 checks in again, gets a card $\langle \mathsf{k}_3, \mathsf{k}_1\rangle$, inadvertently unlocks room r_1 with her previous card, $\langle \mathsf{k}_4, \mathsf{k}_2\rangle$, leaves a diamond on the nightstand, and exits. (4) Guest g_2 enters the room and "borrows" the diamond.

This flaw was already detected by Jackson using the Alloy Analyzer on his original specification and can be fixed by adding $k' \simeq curr\ s\ r$ to the conjunction in ENTRY.

7 Evaluation

An ideal way to assess Nitpick's strength would be to run it against Refute and Quickcheck on a representative database of Isabelle/HOL non-theorems. Lacking such a database, we chose instead to derive formulas from existing theorems by mutation, replacing constants with other constants and swapping arguments, as was done when evaluating Quickcheck [4]. The vast majority of formulas obtained this way are invalid, and those few that are valid do not influence the ranking of the counterexample generators. For executable theorems, we made sure that the generated mutants are also executable to prevent a bias against Quickcheck.

The table below summarizes the results of running the tools on 3200 random mutants from 20 Isabelle theories (200 per theory), with a limit of 10 seconds per formula. Most counterexamples are found within a few seconds; giving the tool more time would have little impact on the results.

THEORY	QUICK.	REF.	NITP.	THEORY	QUICK.	REF.	NITP.
Divides	$134_{/184}$	$3{+}15$	$\mathbf{141}{+}2$	*ArrowGS*	$0_{/0}$	$0{+}126$	$\mathbf{139}{+}2$
Fun	$5_{/9}$	$162{+}1$	$\mathbf{163}{+}0$	*Coinductive*	$4_{/6}$	$16{+}7$	$\mathbf{87}{+}13$
GCD	$119_{/162}$	$1{+}17$	$\mathbf{124}{+}10$	*CoreC++*	$7_{/30}$	$3{+}6$	$\mathbf{29}{+}1$
List	$78_{/130}$	$3{+}113$	$\mathbf{117}{+}9$	*FFT*	$31_{/40}$	$1{+}2$	$\mathbf{47}{+}15$
MacLaurin	$\mathbf{43}_{/62}$	$0{+}0$	$26{+}7$	*Huffman*	$84_{/160}$	$1{+}45$	$\mathbf{119}{+}2$
Map	$19_{/34}$	$103{+}45$	$\mathbf{157}{+}0$	*MiniML*	$14_{/33}$	$0{+}116$	$\mathbf{79}{+}49$
Predicate	$2_{/2}$	$147{+}14$	$\mathbf{161}{+}0$	*NBE*	$41_{/62}$	$0{+}18$	$\mathbf{81}{+}24$
Relation	$0_{/2}$	$144{+}3$	$\mathbf{150}{+}1$	*Ordinal*	$0_{/66}$	$10{+}3$	$\mathbf{12}{+}0$
Set	$17_{/25}$	$149{+}0$	$\mathbf{151}{+}0$	*POPLmark*	$56_{/96}$	$4{+}6$	$\mathbf{103}{+}15$
Wellfounded	$10_{/24}$	$118{+}20$	$\mathbf{141}{+}1$	*Topology*	$0_{/0}$	$124{+}4$	$\mathbf{139}{+}3$

The table's entries have the form $G_{/X}$ for Quickcheck and $G{+}P$ for Refute and Nitpick, where $G =$ number of genuine counterexamples found and reported as such, $P =$ number of potential counterexamples found (in addition to G), and $X =$ number of executable mutants (among 200).

Refute's three-valued logic is unsound, so all counterexamples for formulas that involve an infinite type are potentially spurious and reported as such to the user. Nitpick also has an unsound mode, which contributes some potential counterexamples. Unfortunately, there is no easy way to tell how many of these are actually genuine.

Quickcheck and Nitpick are comparable on executable formulas, but Nitpick also fares well on non-executable ones. Notable exceptions are formalizations involving real arithmetic (*MacLaurin* and *FFT*), complex set-theoretic constructions (*Ordinal*), or a large state space (*CoreC++*).

Independently, Nitpick competes against Refute in the higher-order model finding division of the TPTP [20]. In a preliminary run, it disproved 293 out of 2729 formulas (mostly theorems), compared with 214 for Refute. Much to our surprise, Nitpick exhibited counterexamples for five formulas that had previously been proved by TPS [1] or LEO-II [3], revealing two bugs in the former and one bug in the latter.

8 Related Work

The approaches for testing conjectures can be classified in three broad categories:

- *Random testing.* The formula is evaluated for random values of the free variables. This approach is embodied by Isabelle's Quickcheck [4] and similar tools for other proof assistants. It is restricted to executable formulas.
- *SAT solving.* The formula is translated to propositional logic and handed to a SAT solver. This procedure was pioneered by McCune in his first-order finder MACE [15]. Other first-order MACE-style finders include Paradox [8] and Kodkod [21]. The higher-order finders Refute [23] and Nitpick also belong to this category.
- *Direct search.* The search for a model is performed directly on the formula, without translation to propositional logic. This approach was introduced by SEM [24].

Some proof methods deliver sound or unsound counterexamples upon failure, notably model checking, semantic tableaux, and satisfiability modulo theory (SMT) solving. Also worth of mention is the Dynamite tool [11], which lets users prove Alloy formulas in the interactive theorem prover PVS. Weber [23, pp. 3–4] provides a more detailed discussion of related work.

9 Conclusion

Nitpick is to our knowledge the first higher-order model finder that supports both inductive and coinductive predicates and datatypes. It works by translating higher-order formulas to first-order relational logic (FORL) and invoking the highly-optimized SAT-based Kodkod model finder [21] to solve these. Compared with Quickcheck, which is restricted to executable formulas, Nitpick shines by its generality—the hallmark of SAT-based model finding.

The translation to FORL is designed to exploit Kodkod's strengths. Datatypes are encoded following an Alloy idiom [10, 14] extended to mutually recursive and coinductive datatypes. FORL's relational operators provide a natural encoding of partial application and λ-abstraction, and the transitive closure plays a crucial role in the encoding of inductive datatypes. Our main contributions have been to isolate three ways to translate (co)inductive predicates to FORL, based on wellfoundedness, polarity, and linearity, and to devise optimizations—notably function specialization, boxing, and monotonicity inference—that dramatically increase scalability in practical applications.

Nitpick is included with the latest version of Isabelle and is invoked automatically whenever users enter new formulas to prove, helping to catch errors early, thereby saving time and effort. But Nitpick's real beauty is that it lets users experiment with formal specifications in the playful way championed by Alloy but with Isabelle's higher-order syntax, definition principles, and theories at their fingertips.

Acknowledgment. Stefan Berghofer, Lukas Bulwahn, Marcelo Frias, Florian Haftmann, Alexander Krauss, Mark Summerfield, Emina Torlak, and several anonymous reviewers provided useful comments on drafts of this paper. Alexander Krauss also helped devise the monotonicity inference calculi, and Geoff Sutcliffe ran Nitpick against Refute on the TPTP benchmark suite. We thank them all.

References

1. P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem-proving system for classical type theory. *J. Auto. Reas.*, 16(3):321–353, 1996.
2. J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. *ACM SIGPLAN Notices*, 32(8):25–37, Aug. 1997.
3. C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. Progress report on LEO-II, an automatic theorem prover for higher-order logic. In K. Schneider and J. Brandt, eds., *TPHOLs: Emerging Trends*. C.S. Dept., University of Kaiserslautern, Internal Report 364/07, 2007.
4. S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, eds., *SEFM 2004*, pp. 230–239. IEEE C.S., 2004.
5. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, ed., *TACAS '99*, vol. 1579 of *LNCS*, pp. 193–207. Springer, 1999.
6. J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. In J. Giesl and R. Hähnle, eds., *IJCAR 2010*, LNCS. Springer, 2010. To appear.
7. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, eds., *TPHOLs 2007*, vol. 4732 of *LNCS*, pp. 38–53. Springer, 2007.
8. K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *MODEL*, 2003.
9. A. L. de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. Ph.D. thesis, C.S. Dept., University of Glasgow, 1995.
10. A. Dunets, G. Schellhorn, and W. Reif. Bounded relational analysis of free datatypes. In B. Beckert and R. Hähnle, eds., *TAP 2008*, vol. 4966 of *LNCS*, pp. 99–115. Springer, 2008.
11. M. F. Frias, C. G. L. Pombo, and M. M. Moscato. Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In O. Grumberg and M. Huth, eds., *TACAS 2007*, vol. 4424 of *LNCS*, pp. 587–601. Springer, 2007.
12. M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
13. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
14. V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In H. C. Gall, ed., *ESEC/FSE 2005*, 2005.
15. W. McCune. A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, ANL, 1994.
16. T. Nipkow. Verifying a hotel key card system. In K. Barkaoui, A. Cavalcanti, and A. Cerone, eds., *ICTAC 2006*, vol. 4281 of *LNCS*. Springer, 2006.
17. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
18. J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.
19. G. Snelting and D. Wasserrab. A correctness proof for the Volpano/Smith security typing system. In G. Klein, T. Nipkow, and L. C. Paulson, eds., *AFP*. Sept. 2008.
20. G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving. `http://www.cs.miami.edu/~tptp/`.
21. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, eds., *TACAS 2007*, vol. 4424 of *LNCS*, pp. 632–647. Springer, 2007.
22. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Comp. Sec.*, 4(3):167–187, Dec. 1996.
23. T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2008.
24. J. Zhang and H. Zhang. SEM: A system for enumerating models. In M. Kaufmann, ed., *IJCAI 95*, vol. 1, pp. 298–303, 1995.