Nitpick: A Counterexample Generator for Isabelle/HOL Based on the Relational Model Finder Kodkod (System Description)*

Jasmin Christian Blanchette
Technische Universität München
Garching, Germany
blanchette@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. 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.

For several years, Isabelle/HOL [11] has provided two such tools: Quickcheck [3] generates functional code for the higher-order logic (HOL) formula and evaluates it for random values of the free variables, and Refute [16] searches for finite countermodels of a formula through a reduction to SAT. 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 [9], a testing tool for first-order relational logic (FORL), has enjoyed considerable success lately. Alloy's backend, the relational model finder Kodkod [14], 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 [7]. 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; however, common HOL idioms such as (co)inductive datatypes and (co)inductive predicates necessitate a translation scheme tailored for SAT solving [5]. In addition, Nitpick benefits from many novel optimizations that greatly improve its performance, especially in the presence of higher-order constructs.

Our evaluation indicates that Nitpick falsifies more formulas than Quickcheck and Refute, to a large extent because it imposes no syntactic restrictions on the formulas to falsify. Nitpick is integrated with the TPTP benchmark suite [13] and exposed three bugs in the higher-order provers TPS [1] and LEO-II [2].

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

2 The Basic Translation

Nitpick employs Kodkod to find a finite model of $\neg P$, where P is the conjecture. The translation from HOL to FORL is parameterized by the cardinalities of the atomic types occurring in it. Nitpick enumerates the possible cardinalities for each atomic type, exploiting monotonicity to prune the search space [6]. If a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas. Whenever practicable, HOL constants are mapped to their FORL equivalents, rather than expanded to their definitions. For example, HOL defines the transitive closure r^+ as the least fixed point of $\lambda R\,(x, y).\ (\exists a\,b.\ x = a \wedge y = b \wedge r\ (a, b)) \vee (\exists a\,b\,c.\ x = a \wedge y = c \wedge R\ (a, b) \wedge r\ (b, c))$; in FORL it is built-in.

As a rule, HOL scalars are mapped to FORL singletons and functions are mapped to FORL relations accompanied by a constraint. For example, assuming the cardinalities $|\alpha| = 2$ and $|\beta| = 3$, the conjecture $\forall x :: \alpha.\ \exists y :: \beta.\ f\,x = y$ corresponds to the (negated) Kodkod problem

$$\begin{array}{l}\textbf{var } \emptyset \subseteq f \subseteq \{\mathsf{a}_1, \mathsf{a}_2\} \times \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\} \\ \textbf{solve } (\forall x \in \{\mathsf{a}_1, \mathsf{a}_2\}.\ \mathsf{one}\ f(x)) \wedge \neg\,(\forall x \in \{\mathsf{a}_1, \mathsf{a}_2\}.\ \exists y \in \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}.\ f(x) = y)\end{array}$$

The **var** declaration declares the free relation f with lower and upper bounds. 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 *bool*, the function is more efficiently coded as an unconstrained n-ary relation. This allows formulas such as $A^+ \cup B^+ = (A \cup B)^+$ 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 $\forall g :: \beta \to \alpha.\ g\,x \neq y$ into something like

$$\forall g \subseteq \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\} \times \{\mathsf{a}_1, \mathsf{a}_2\}.\ (\forall x \in \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}.\ \mathsf{one}\ g(x)) \longrightarrow g(x) \neq y,$$

but the $\subseteq$ symbol is not allowed at the binding site; only $\in$ is. Skolemization solves half of the problem, 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. The n-tuple simply encodes g's function table. For the formula above, this gives

$$\forall G \in \{\mathsf{a}_1, \mathsf{a}_2\} \times \{\mathsf{a}_1, \mathsf{a}_2\} \times \{\mathsf{a}_1, \mathsf{a}_2\}.\ \big(\overbrace{\{\mathsf{a}_3\} \times \pi_1(G) \cup \{\mathsf{a}_4\} \times \pi_2(G) \cup \{\mathsf{a}_5\} \times \pi_3(G)}^{g}\big)(x) \neq y,$$

where G is the triple corresponding to g and $\pi_i(G)$ is its ith component (i.e., the ith entry in the function table). 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 [7], make this rarely necessary.

3 Refinements to the Basic Translation

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 ($\star$), coded as the empty set. Formulas of the form $\forall n :: nat.\ P(n)$ are treated as $(\forall n < K.\ P(n)) \wedge P(\star)$, which usually evaluates to either *False* (if $P(i)$ gives *False* for some $i < K$) or $\star$, but not to *True*, since we do not know whether $P(K)$, $P(K+1)$, ..., collectively represented by $P(\star)$, are true. Partiality leads to a Kleene three-valued logic, which is soundly expressed in terms of Kodkod's two-valued logic.

Encoding of (Co)inductive Predicates. Isabelle lets users specify (co)inductive predicates p by their introduction rules and synthesizes a fixed point definition $p = \mathit{lfp}\ F$ or $p = \mathit{gfp}\ F$ behind the scenes. For performance reasons, Nitpick avoids expanding *lfp* and *gfp* to their definitions and translates (co)inductive predicates directly, using appropriate FORL concepts.

An inductive predicate p is a fixed point, so we can use the equation $p = F\ p$ 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:

- If the recursion in F is well-founded [8], the fixed point equation $p = F\ p$ admits exactly one solution and we can safely use it as p's specification.
- If p occurs negatively in the formula, we can replace these occurrences by a fresh constant q satisfying the axiom $q = F\ q$; this transformation preserves satisfiability.

For the remaining positive occurrences of p, we unroll the predicate a given number of times, as in bounded model checking [4]. The situation is mirrored for coinductive predicates: Positive occurrences are coded using the fixed-point equation, and negative occurrences are unrolled.

Encoding of (Co)inductive Datatypes. 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 [10]. 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 $\alpha\ \mathit{list}$ generated from $\mathit{Nil} :: \alpha\ \mathit{list}$ and $\mathit{Cons} :: \alpha \to \alpha\ \mathit{list} \to \alpha\ \mathit{list}$. The FORL axiomatization is done in terms of the discriminators *nilp* and *consp* and the selectors *hd* and *tl*, which give access to a nonempty list's head and tail. *Nil* and *Cons x xs* are translated as *nilp* and $\mathit{hd}^{-1}(x) \cap \mathit{tl}^{-1}(\mathit{xs})$.

The FORL axiomatization specifies a subterm-closed finite universe of lists. Examples of subterm-closed list substructures using traditional notation are $\{[], [0], [1]\}$ and $\{[], [1], [2,1], [0,2,1]\}$. In contrast, the set $L = \{[], [1,1]\}$ is not subterm-closed, because $\mathit{tl}([1,1]) = [1] \notin L$. Given cardinalities for the list type and the item type, Kodkod enumerates all corresponding subterm-closed list substructures.

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.

4 Example: A Security Type System

Assuming a partition of program variables into public and private ones, Volpano, Smith, and Irvine [15] provide typing rules guaranteeing that private variables stay private. They define two types, *High* (private) and *Low* (public). An expression is *High* if it involves private variables and *Low* otherwise. A command is *High* if it modifies private variables only; commands that could alter public variables are *Low*.

We consider a fragment of the formal soundness proof by Snelting and Wasserrab [12]. 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 $\Gamma, \sigma \vdash c$:

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

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

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

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

It asserts that if executing the *High* command c in state s terminates in s', 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:

$$\Gamma = [\mathtt{v}_1 \mapsto \mathit{Low}] \qquad s = [\mathtt{v}_1 \mapsto \mathit{false}]$$
$$c = \mathtt{skip\,;\ v_1\ :=}\ (\mathit{Var}\ \mathtt{v}_1\ \mathtt{==}\ \mathit{Var}\ \mathtt{v}_1) \qquad s' = [\mathtt{v}_1 \mapsto \mathit{true}].$$

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.

5 Evaluation

To assess Nitpick, we used a database of mutated formulas consisting mostly of non-theorems, as was done for Quickcheck [3]. The table below summarizes the results of running Nitpick, Refute, and Quickcheck on 2400 random mutants from 12 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	3	**141**	*ArrowGS*	0	0	**139**
List	78	3	**117**	*CoreC++*	7	3	**29**
MacLaurin	**43**	0	26	*MiniML*	14	0	**79**
Map	19	103	**157**	*Ordinal*	0	10	**12**
Relation	0	144	**150**	*POPLmark*	56	4	**103**
Set	17	149	**151**	*Topology*	0	124	**139**

Nitpick also competes against Refute in the higher-order model finding division of the TPTP [13]. 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 been proved by TPS [1] or LEO-II [2], revealing two bugs in the former and one bug in the latter. In exchange, LEO-II exposed one bug in Nitpick.

6 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 [14] to solve these.

The translation to FORL is designed to exploit Kodkod's strengths. Datatypes are encoded following an Alloy idiom [10] 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 and polarity, and to devise optimizations that dramatically increase scalability in practical applications [6, 7].

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. Nitpick would have been impossible without the help and support of Tobias Nipkow and the members of the Isabelle group in Munich. Alexander Krauss contributed to the monotonicity inference calculi used to prune the search space. Emina Torlak developed the Kodkod model finder that serves as Nitpick's backend. Geoff Sutcliffe runs Nitpick regularly against Refute, Satallax, Paradox, and other model finders on his computer farm. Andreas Lochbihler, Denis Lohner, and Daniel Wasserrab were among the first users of the system and provided much helpful feedback. 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] 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, editors, *TPHOLs: Emerging Trends*. C.S. Dept., University of Kaiserslautern, Internal Report 364/07, 2007.

[3] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *SEFM 2004*, pages 230–239. IEEE C.S., 2004.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *TACAS '99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[5] J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. In G. Fraser and A. Gargantini, editors, *TAP 2010*, volume 6143 of *LNCS*, pages 117–134. Springer, 2010.

[6] J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 91–106. Springer, 2010.

[7] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *ITP-10*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[8] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 38–53. Springer, 2007.

[9] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[10] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In H. C. Gall, editor, *ESEC/FSE 2005*, 2005.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[12] G. Snelting and D. Wasserrab. A correctness proof for the Volpano/Smith security typing system. In G. Klein, T. Nipkow, and L. C. Paulson, editors, *AFP*. Sept. 2008.

[13] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving. `http://www.cs.miami.edu/~tptp/`.

[14] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[15] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Comp. Sec.*, 4(3):167–187, Dec. 1996.

[16] T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2008.