# Generating Counterexamples for Structural Inductions by Exploiting Nonstandard Models

Jasmin Christian Blanchette[1]⋆ and Koen Claessen[2]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Dept. of CSE, Chalmers University of Technology, Gothenburg, Sweden

**Abstract.** Induction proofs often fail because the stated theorem is noninductive, in which case the user must strengthen the theorem or prove auxiliary properties before performing the induction step. (Counter)model finders are useful for detecting non-theorems, but they will not find any counterexamples for noninductive theorems. We explain how to apply a well-known concept from first-order logic, nonstandard models, to the detection of noninductive invariants. Our work was done in the context of the proof assistant Isabelle/HOL and the counterexample generator Nitpick.

## 1 Introduction

Much of theorem proving in higher-order logics, whether interactive or automatic, is concerned with induction proofs: rule induction over inductive predicates, structural induction over inductive datatypes (which includes mathematical induction over natural numbers as a special case), and recursion induction over well-founded recursive functions. Inductive properties are difficult to prove because the failure to perform an induction step can mean any of the following:

1. The property is not a theorem.
2. The property is a theorem but is too weak to support the induction step.
3. The property is a theorem and is inductive, although no proof has been found yet.

Depending on which of the above scenarios applies, the prover (human or machine) would take the appropriate course of action:

1. Repair the property's statement or its underlying specification so that it becomes a theorem.
2. Generalize the property and/or prove auxiliary properties.
3. Work harder on a proof.

How can we distinguish these three cases? Counterexample generators can often detect scenario 1, and automatic proof methods sometimes handle scenario 3, but what can we do when we have neither a proof nor a counterexample?

This paper introduces a method for detecting noninductive properties of datatypes (properties that cannot be proved without structural induction) using a model finder, inspired by previous work involving the second author to detect noninductive invariants of

⋆ Research supported by the DFG grant Ni 491/11-1.

state transition systems [8]. The basic idea is to weaken the higher-order axiomatization of datatypes to allow nonstandard models (Section 4). The existence of a nonstandard countermodel for a theorem means that it cannot be proved without structural induction. If the theorem is an induction step, we have identified scenario 2.

A concrete nonstandard model can also help the prover to figure out *how* the induction needs to be strengthened for the proof to go through, in the same way that a standard counterexample helps locate mistakes in a conjecture. Our examples (Sections 3 and 5) illustrate how this might be done.

We have implemented the approach in the counterexample generator Nitpick for the proof assistant Isabelle/HOL, which we employed for the examples and the evaluation (Section 6). When Nitpick finds no standard countermodel to an induction step, it asks permission to run again, this time looking for nonstandard countermodels.

## 2 Background

### 2.1 Isabelle/HOL

Isabelle [13] is a generic theorem prover whose built-in metalogic is an intuitionistic fragment of higher-order logic [7, 9]. The metalogical operators include implication, written $\varphi \Longrightarrow \psi$, and universal quantification, written $\bigwedge x.\ \varphi$. Isabelle's HOL object logic provides a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers ($\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$, $\forall$, $\exists$). Isabelle proofs are usually written in the human-readable Isar format inspired by Mizar [18]. This paper will show some proofs written in Isar. We do not expect readers to understand every detail of the proofs, and will explain any necessary Isar syntax in context.

The term language consists of simply-typed $\lambda$-terms augmented with constants and weak polymorphism. We adopt the convention that italicized Latin letters with optional subscripts denote variables, whereas longer names denote constants. Function application expects no parentheses around the argument list and no commas between the arguments, as in $f\ x\ y$. Syntactic sugar provides an infix syntax for common operators, such as $x = y$ and $x + y$. Variables may range over functions and predicates. Types are usually implicit but can be specified using a constraint $:: \tau$.

HOL's standard semantics interprets the Boolean type *bool* and the function space $\sigma \to \tau$. The function arrow associates to the right, reflecting the left-associativity of application. HOL identifies sets with unary predicates and provides syntactic sugar for set-theoretic notations. Additional types can be declared as uninterpreted types or as isomorphic to a subset of another type. Alternatively, inductive datatypes can be declared by specifying the constructors and the types of their arguments. For example, Isabelle's type $\alpha$ *list* of finite lists over the type variable $\alpha$ is declared as follows:

**datatype** $\alpha$ *list* = *Nil* | *Cons* $\alpha$ ($\alpha$ *list*) (**infixr** "·")

The type is generated freely from *Nil* :: $\alpha$ *list* and *Cons* :: $\alpha \to \alpha$ *list* $\to \alpha$ *list*. The **infixr** tag declares the infix syntax $x \cdot xs$ as an abbreviation for *Cons* $x$ $xs$. Since lists are so common, Isabelle also supports the traditional notation $[x_1, \ldots, x_n]$.

Constants can be introduced axiomatically or definitionally. Isabelle also provides high-level definitional mechanisms for defining inductive sets and predicates as well as recursive functions. For example, the $hd :: \alpha\ list \to \alpha$ and $tl :: \alpha\ list \to \alpha\ list$ functions that return the head and tail of a list are specified by $hd\ (x \cdot xs) = x$ and $tl\ (x \cdot xs) = xs$.

### 2.2 Nitpick

Nitpick [5] is a counterexample generator for Isabelle/HOL based on Kodkod [17], a finite model finder for first-order relational logic that in turn relies on SAT solving. Given a conjecture, Nitpick determines which axioms and definitions are needed and searches for a standard set-theoretic model that satisfies the axioms and falsifies the conjecture.

The basic translation from HOL to Kodkod's relational logic is straightforward, but common HOL idioms such as inductive predicates, inductive datatypes, and recursive functions necessitate a translation scheme tailored for SAT solving. In particular, infinite datatypes are (soundly) approximated by finite subterm-closed subsets [4].

The following example shows Nitpick in action on a conjecture about list reversal:

**theorem** REV_CONS_REV: $rev\ (x \cdot rev\ xs) = x \cdot xs$
**nitpick** [*show_datatypes*]

*Nitpick found a counterexample for* $|\alpha| = 5$*:*

*Free variables:* $x = a_1 \quad xs = [a_2]$
*Datatype:* $\alpha\ list = \{[], [a_1], [a_1, a_2], [a_2], [a_2, a_1], \ldots\}$

(The output is shown in a slanted font to distinguish it from the user's proof text and interactive commands.) We see that it sufficed to consider the subset $\alpha\ list = \{[], [a_1], [a_1, a_2], [a_2], [a_2, a_1], \ldots\}$ of all lists over $\{a_1, \ldots, a_5\}$ to falsify the conjecture.

## 3 Introductory Examples

Our approach is best explained by demonstrating it on a few examples before looking at the technicalities. The first example focuses on rule induction: We define a set inductively and attempt to prove properties about it by following the introduction rules. Although rule induction is not our main topic, the example is instructive in its own right and serves as a stepping stone for the application of our method to structural induction proofs. The second example illustrates a failed structural induction on binary trees.

### 3.1 Rule Induction

Properties about inductively defined sets and predicates can be proved by rule induction. The following specification of the scoring of tennis games will serve as illustration:

**datatype** *player* = *Serv* | *Recv*
**datatype** *score* = *Points nat nat* (**infix** "◇") | *Adv player* | *Game player*

**inductive_set** *legal* :: *score list* $\rightarrow$ *bool* **where**

LOVE_ALL: $[0\diamond 0] \in \mathit{legal}$
SERV_15: $0\diamond n \cdot xs \in \mathit{legal} \Longrightarrow 15\diamond n \cdot 0\diamond n \cdot xs \in \mathit{legal}$
SERV_30: $15\diamond n \cdot xs \in \mathit{legal} \Longrightarrow 30\diamond n \cdot 15\diamond n \cdot xs \in \mathit{legal}$
$\vdots$
RECV_GAME: $n\diamond 40 \cdot xs \in \mathit{legal} \Longrightarrow n \neq 40 \Longrightarrow \mathit{Game\ Recv} \cdot n\diamond 40 \cdot xs \in \mathit{legal}$
DEUCE_ADV: $40\diamond 40 \cdot xs \in \mathit{legal} \Longrightarrow \mathit{Adv}\ p \cdot 40\diamond 40 \cdot xs \in \mathit{legal}$
ADV_DEUCE: $\mathit{Adv}\ p \cdot xs \in \mathit{legal} \Longrightarrow 40\diamond 40 \cdot \mathit{Adv}\ p \cdot xs \in \mathit{legal}$
ADV_GAME: $\mathit{Adv}\ p \cdot xs \in \mathit{legal} \Longrightarrow \mathit{Game}\ p \cdot \mathit{Adv}\ p \cdot xs \in \mathit{legal}$

A game is a trace $[s_n, \ldots, s_1]$ of successive scores listed in reverse order. The inductively defined set *legal* is the set of all legal (complete or incomplete) games, starting from the score $0\diamond 0$. For example, $[15\diamond 15, 15\diamond 0, 0\diamond 0]$ and $[\mathit{Game\ Recv}, 0\diamond 40, 0\diamond 30, 0\diamond 15, 0\diamond 0]$ are legal games, but $[15\diamond 0]$ is not.

By manually inspecting the rules, it is easy to persuade ourselves that no player can reach more than 40 points. Nitpick is also convinced:

**theorem** LE_40: $g \in \mathit{legal} \Longrightarrow a\diamond b \in g \longrightarrow \mathit{max}\ a\ b \leq 40$
**nitpick**

*Nitpick found no counterexample.*

(The symbol '$\in$' is overloaded to denote list membership as well as set membership.) Let us try to prove the above property by rule induction:

**proof** (*induct set*: *legal*)
**case** LOVE_ALL **thus** *?case* **by** *simp*

The first line of the proof script tells Isabelle that we want to perform a proof by rule induction over the set *legal*. The second line selects the proof obligation associated with the LOVE_ALL rule from *legal*'s definition and discharges it using the *simp* method, which performs equational reasoning.

**case** (SERV_15 *n xs*) **thus** *?case*

The next line selects the proof obligation associated with the SERV_15 rule. At this point, the induction hypothesis is

$$a\diamond b \in 0\diamond n \cdot xs \longrightarrow \mathit{max}\ a\ b \leq 40, \qquad (\mathcal{IH})$$

and we must prove

$$a\diamond b \in 15\diamond n \cdot 0\diamond n \cdot xs \longrightarrow \mathit{max}\ a\ b \leq 40. \qquad (\mathcal{G})$$

We may also assume

$$0\diamond n \cdot xs \in \mathit{legal}, \qquad (\mathcal{R})$$

since it occurs as a hypothesis in SERV_15. Observe that the hypothesis $\mathcal{R}$ involves *legal*, which is precisely the set on which we are performing rule induction. If we stated our theorem "strongly enough," it should be sufficient to use the induction hypothesis $\mathcal{IH}$ to prove the goal $\mathcal{G}$, without reasoning about *legal* directly. (There is certainly nothing wrong with reasoning about *legal*, but this would mean performing a nested induction proof or invoking a lemma that we would then prove by induction. We want to avoid this if we can.)

Can we prove $\mathcal{IH} \Longrightarrow \mathcal{G}$ without $\mathcal{R}$? None of Isabelle's automatic tactics appear to work, and if we tell Nitpick to ignore $\mathcal{R}$, it finds the following counterexample:

*Free variables:* $a = 15 \quad b = 41 \quad n = 41 \quad xs = []$

Indeed, the induction hypothesis is not applicable, because $15 \diamond 41 \notin [0 \diamond 41]$; and the goal is falsifiable, because $15 \diamond 41 \in [15 \diamond 41]$ and $max\ 15\ 41 \not\leq 40$. The counterexample disappears if we reintroduce $\mathcal{R}$, since $[0 \diamond 41]$ is not a legal game. This suggests that the stated theorem is correct, but that it is not general enough to support the induction step.

The countermodel tells us additional information that we can use to guide our search for a proof. First, notice that the counterexample falsifies $a \diamond b \in 0 \diamond n \cdot xs$, and so the induction hypothesis is useless. On closer inspection, instantiating $a$ with 15 in the induction hypothesis is odd; it would make more sense to let $a$ be 0. Then $\mathcal{IH}$ becomes $0 \diamond 41 \in [0 \diamond 41] \longrightarrow max\ 0\ 41 \leq 40$, which is false—and the countermodel disappears because $\mathcal{IH} \Longrightarrow \mathcal{G}$ is true. If we can eradicate all countermodels, it is likely that the formula will become provable.

This instantiation of $a$ would have been possible if $a$ had been universally quantified in $\mathcal{IH}$. This can be achieved by explicitly quantifying over $a$ in the statement of the theorem. We do the same for $b$. The proof is now within the *auto* tactic's reach:

**theorem** LE_40: $g \in legal \Longrightarrow \forall a\ b.\ a \diamond b \in g \longrightarrow max\ a\ b \leq 40$
**by** (*induct set*: *legal*) *auto*

Explicit universal quantification is a standard proof heuristic [13, pp. 33–36]. An equally valid approach would have been to precisely characterize the possible scores in a legal game and then use that characterization to prove the desired property:

**theorem** ALL_LEGAL:
$g \in legal \Longrightarrow \forall s \in g.\ s \in \{m \diamond n \mid \{m, n\} \subseteq \{0, 15, 30, 40\}\}$
$\cup\ range\ Adv \cup range\ Game$
**by** (*induct set*: *legal*) *auto*

**theorem** LE_40: $g \in legal \Longrightarrow a \diamond b \in g \longrightarrow max\ a\ b \leq 40$
**by** (*frule* ALL_LEGAL [**THEN** BALL_E, **where** $x =$ "$a \diamond b$"]) *auto*

What can we learn from this example? In general, proofs by rule induction give rise to subgoals of the form $\mathcal{R} \wedge \mathcal{IH} \wedge \mathcal{SC} \Longrightarrow \mathcal{G}$, where $\mathcal{R}$ represents the recursive antecedents of the rule, $\mathcal{IH}$ represents the induction hypotheses, and $\mathcal{SC}$ is the rule's side condition. When we claim that an induction hypothesis is "strong enough," we usually mean that we can carry out the proof without invoking $\mathcal{R}$. If $\mathcal{IH} \wedge \mathcal{SC} \Longrightarrow \mathcal{G}$ admits a counterexample, the induction hypothesis is too weak: We must strengthen the formula we want to prove or exploit $\mathcal{R}$ in some way.

This issue arises whenever we perform induction proofs over inductively defined "legal" values or "reachable" states. In the next section, we will carry this idea over to structural induction.

## 3.2 Structural Induction

As an example, consider the following mini-formalization of full binary trees:

**datatype** $\alpha$ *btree* = *Lf* $\alpha$ | *Br* ($\alpha$ *btree*) ($\alpha$ *btree*)

**fun** *labels* :: $\alpha$ *btree* $\to \alpha \to$ *bool* **where**
*labels* (*Lf a*) $= \{a\}$
*labels* (*Br* $t_1$ $t_2$) $=$ *labels* $t_1 \cup$ *labels* $t_2$

**fun** *swap* :: $\alpha$ *btree* $\to \alpha \to \alpha \to \alpha$ *btree* **where**
*swap* (*Lf c*) *a b* $=$ *Lf* (if $c = a$ then $b$ else if $c = b$ then $a$ else $c$)
*swap* (*Br* $t_1$ $t_2$) *a b* $=$ *Br* (*swap* $t_1$ *a b*) (*swap* $t_2$ *a b*)

A tree is either a labeled leaf (*Lf*) or an unlabeled inner node (*Br*) with a left and right child. The *labels* function returns the set of labels that occur on a tree's leaves, and *swap* simultaneously substitutes two labels for each other. Intuitively, if two labels $a$ and $b$ occur in a tree $t$, they should also occur in the tree obtained by swapping $a$ and $b$:

**theorem** LABELS_SWAP: $\{a, b\} \subseteq$ *labels t* $\longrightarrow$ *labels* (*swap t a b*) $=$ *labels t*

Nitpick cannot disprove this, so we proceed with structural induction on the tree $t$:

**proof** (*induct t*)
  **case** LF **thus** *?case* **by** *simp*
  **case** (BR $t_1$ $t_2$) **thus** *?case*

The induction hypotheses are

$$\{a, b\} \subseteq \textit{labels}\ t_1 \longrightarrow \textit{labels}\ (\textit{swap}\ t_1\ a\ b) = \textit{labels}\ t_1 \qquad (\mathcal{IH}_1)$$
$$\{a, b\} \subseteq \textit{labels}\ t_2 \longrightarrow \textit{labels}\ (\textit{swap}\ t_2\ a\ b) = \textit{labels}\ t_2 \qquad (\mathcal{IH}_2)$$

and the goal is

$$\{a, b\} \subseteq \textit{labels}\ (\textit{Br}\ t_1\ t_2) \longrightarrow \textit{labels}\ (\textit{swap}\ (\textit{Br}\ t_1\ t_2)\ a\ b) = \textit{labels}\ (\textit{Br}\ t_1\ t_2). \qquad (\mathcal{G})$$

Nitpick cannot find any counterexample to $\mathcal{IH}_1 \wedge \mathcal{IH}_2 \Longrightarrow \mathcal{G}$, but thanks to the technique we present in this paper it now makes the following suggestion:

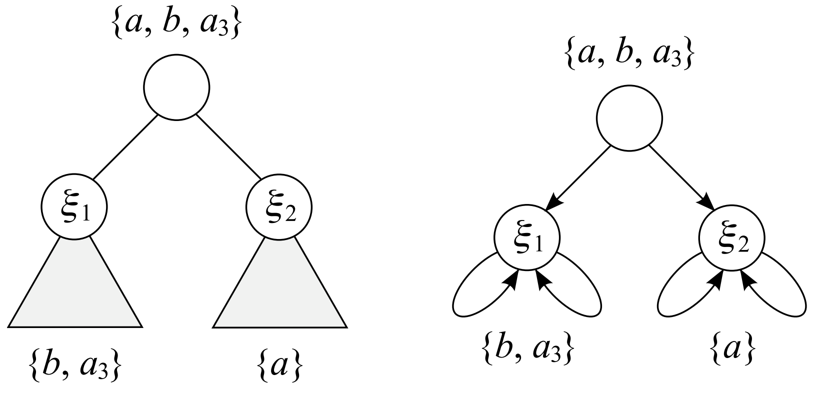
*Hint: To check that the induction hypothesis is general enough, try this command:* ***nitpick*** *[non_std, show_all].*

If we follow the hint, we get the output below.

*Nitpick found a nonstandard counterexample for* $|\alpha| = 3$:

```
   Free variables:  a = a_1   b = a_2   t_1 = ξ_1   t_2 = ξ_2
         Datatype:  α btree = {ξ_1 = Br ξ_1 ξ_1, ξ_2 = Br ξ_2 ξ_2, Br ξ_1 ξ_2}
        Constants:  labels = (λx. ?)(ξ_1 ↦ {a_2, a_3}, ξ_2 ↦ {a_1},
                                     Br ξ_1 ξ_2 ↦ {a_1, a_2, a_3})
                    λx. swap x a b = (λx. ?)(ξ_1 ↦ ξ_2, ξ_2 ↦ ξ_2, Br ξ_1 ξ_2 ↦ ξ_2)
```

*The existence of a nonstandard model suggests that the induction hypothesis is not general enough or may even be wrong. See the Nitpick manual's "Inductive Properties" section for details.*



(a) abstract view (b) concrete view

**Fig. 1.** A nonstandard tree

What is happening here? The *non_std* option told the tool to look for nonstandard models of binary trees, which means that new nonstandard trees $\xi_1$, $\xi_2$, …, are now allowed in addition to the standard trees generated by *Lf* and *Br*. Unlike standard trees, these new trees contain cycles: The "Datatype" section of Nitpick's output tells us that $\xi_1 = Br\ \xi_1\ \xi_1$ and $\xi_2 = Br\ \xi_2\ \xi_2$. Although this may seem counterintuitive, every property of acyclic objects that can be proved without using induction also holds for cyclic objects. Hence, if Nitpick finds a counterexample with cyclic objects in it (a nonstandard countermodel), the property cannot be proved without using induction.

Here the tool found the nonstandard trees $t_1 = \xi_1$ and $t_2 = \xi_2$ such that $a \notin labels\ t_1$, $b \in labels\ t_1$, $a \in labels\ t_2$, and $b \notin labels\ t_2$. The situation is depicted in Figure 1. Because neither subtree contains both $a$ and $b$, the induction hypothesis tells us nothing about the labels of $swap\ t_1\ a\ b$ and $swap\ t_2\ a\ b$. Thus, the model finder can assign arbitrary values to the results of *labels* and *swap* for the nonstandard trees, as long as the equations defining those functions are respected. The theorem is "falsified" because $labels\ (swap\ t_1\ a\ b) = \{b, a_3\}$ but $labels\ t_1 = \{a\}$. This could never happen for a standard tree $t_1$, but we need induction to prove this.

We can repair the proof of the theorem by ensuring that we always know what the labels of the subtrees are in the induction step, by also covering the cases where $a$ and/or $b$ is not in $t$:

**theorem** LABELS_SWAP:
$labels\ (swap\ t\ a\ b) = (\text{if } a \in labels\ t \text{ then}$
$\qquad\qquad \text{if } b \in labels\ t \text{ then } labels\ t \text{ else } (labels\ t - \{a\}) \cup \{b\}$
$\qquad \text{else}$
$\qquad\qquad \text{if } b \in labels\ t \text{ then } (labels\ t - \{b\}) \cup \{a\} \text{ else } labels\ t)$

This time Nitpick will not find any nonstandard counterexample, and we can prove the induction step using the *auto* tactic.

## 4 The Approach

The previous section offered a black-box view of our approach to debugging structural induction steps. Let us now take a look inside the box.

### 4.1 Description

Our approach consists in weakening the datatype axioms so that the induction principle no longer holds. As a result, properties that can only be proved by induction are no longer valid and admit countermodels. To illustrate this, we restrict our attention to the type *nat* of natural numbers generated from $0 :: nat$ and $Suc :: nat \rightarrow nat$. It is axiomatized as follows [3]:

$$\begin{aligned} \textsc{Distinct:}\quad & 0 \neq Suc\ n \\ \textsc{Inject:}\quad & Suc\ m = Suc\ n \longleftrightarrow m = n \\ \textsc{Induct:}\quad & P\,0 \Longrightarrow \big(\textstyle\bigwedge n.\ P\,n \Longrightarrow P\,(Suc\ n)\big) \Longrightarrow P\,n \end{aligned}$$

When we declare a datatype, Isabelle constructs a set-theoretic definition for the type, derives characteristic theorems from the definition, and derives other useful theorems that follow from the characteristic theorems. From the user's point of view, the characteristic theorems axiomatize the datatype, and the underlying set-theoretic definition can be ignored. Accordingly, we will allow ourselves to write "axioms" instead of "characteristic theorems."[3]

The following theorem is a consequence of INDUCT:

$$\textsc{Nchotomy:}\quad n = 0 \vee (\exists m.\ n = Suc\ m)$$

A well-known result from first-order logic is that if we consider only the DISTINCT and INJECT axioms and leave out INDUCT (which is second-order), nonstandard models of natural numbers are allowed alongside the standard model [15]. In these nonstandard models, we still have distinct values for 0, *Suc* 0, *Suc* (*Suc* 0), ..., but also additional values ("junk") that cannot be reached starting from 0 by applying *Suc* a finite number of times. For example, the domain

$$|\mathcal{M}| = \{0, 1, 2, \ldots\} \cup \{\tilde{0}, \tilde{1}, \tilde{2}, \ldots\} \cup \{a, b, c\}$$

with

$$\begin{array}{llll} 0^{\mathcal{M}} = 0 & Suc^{\mathcal{M}}(0) = 1 & Suc^{\mathcal{M}}(\tilde{0}) = \tilde{1} & Suc^{\mathcal{M}}(a) = a \\ & Suc^{\mathcal{M}}(1) = 2 & Suc^{\mathcal{M}}(\tilde{1}) = \tilde{2} & Suc^{\mathcal{M}}(b) = c \\ & \quad\vdots & \quad\vdots & Suc^{\mathcal{M}}(c) = b \end{array}$$

is a nonstandard model of natural numbers (Figure 2). If we introduce NCHOTOMY as an axiom, the above is no longer a model, because $\tilde{0}$ is neither zero nor the successor of some number. In contrast, $|\mathcal{M}'| = \{0, 1, 2, \ldots\} \cup \{a, b, c\}$ is a model, with $0^{\mathcal{M}'}$ and $Suc^{\mathcal{M}'}$ defined as for $\mathcal{M}$.

[3] Isabelle's definitional approach stands in contrast to the axiomatic approach adopted by PVS and other provers, where the datatype axioms' consistency must be trusted [14]. Here, we take a PVS view of Isabelle.

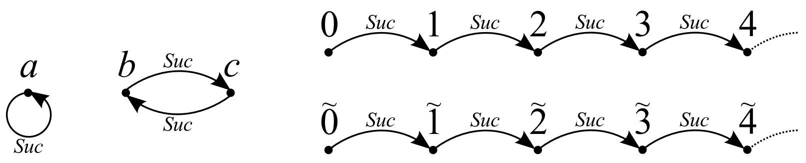


**Fig. 2.** A nonstandard model of the natural numbers

Our method relies on the following key observation: *If a property P is "general enough," the induction step $P\ n \Longrightarrow P\ (Suc\ n)$ can be proved without using the* INDUCT *axiom and hence it admits no countermodel even if we substitute* NCHOTOMY *for* INDUCT. It makes sense to add NCHOTOMY because users normally do not think of case distinction as a form of induction; NCHOTOMY is first-order and easy to apply.

The method was illustrated on natural numbers but is easy to generalize to all recursive datatypes. Self-recursive datatypes such as $\alpha$ *list* and $\alpha$ *btree* are handled in the same way. Mutually recursive datatypes share their INDUCT axiom, but each type has its own NCHOTOMY theorem; we simply replace INDUCT by the NCHOTOMY theorems.

### 4.2 Theoretical Properties

The soundness of our method follows directly from the definition.

**Definition 1 (Nonstandard Models).** Let $\bar{\tau}$ be some datatypes, $\mathcal{C}$ be a formula, and $\mathcal{A}$ the set of relevant axioms to $\mathcal{C}$. A *$\bar{\tau}$-nonstandard model* of $\mathcal{C}$ with respect to $\mathcal{A}$ is a model of $\tilde{\mathcal{A}} \vdash \mathcal{C}$, where $\tilde{\mathcal{A}}$ is constructed from $\mathcal{A}$ by replacing INDUCT with NCHOTOMY for all types $\bar{\tau}$.

**Theorem 1 (Soundness).** *If there exists a $\bar{\tau}$-nonstandard countermodel to $\mathcal{C}$, then $\mathcal{C}$ cannot be proved using only the* DISTINCT, INJECT, *and* NCHOTOMY *properties of $\bar{\tau}$.*

*Proof.* This follows directly from the definition of nonstandard models and the soundness of the proof system. □

The converse to Theorem 1, completeness, does not hold, because the HOL proof system is incomplete with respect to standard models, and because model finders such as Nitpick must necessarily miss some infinite models or be unsound.

### 4.3 Implementation

The implementation in Nitpick deviates from the above description, because it has its own axiomatization of datatypes based on selectors, directly encoded in Kodkod's relational logic [4]. The type *nat* would be axiomatized as follows:

$$\begin{array}{rlrl}
\textsc{Disj}: & \textsf{no}\ zero \cap sucs & \textsc{Uniq}_0: & \textsf{lone}\ zero \\
\textsc{Exhaust}: & zero \cup sucs = nat & \textsc{Uniq}_{Suc}: & \textsf{lone}\ prec^{-1}(n) \\
\textsc{Select}_{prec}: & \textsf{if}\ n \in sucs\ \textsf{then one}\ prec(n) & \textsc{Acycl}: & (n,n) \notin prec^{+}. \\
& \textsf{else no}\ prec(n) & &
\end{array}$$

In Kodkod's logic, terms denote relations; for example, $prec(n)$ denotes the set (or unary relation) of all $m$ such that $(n,m)$ belongs to the binary relation $prec$. Free variables denote singletons. The constraint $\mathsf{no}\ r$ expresses that $r$ is the empty relation, $\mathsf{one}\ r$ expresses that $r$ is a singleton, and $\mathsf{lone}\ r \iff \mathsf{no}\ r \vee \mathsf{one}\ r$.

Users can instruct Nitpick to generate nonstandard models by specifying the *non_std* option, in which case the ACYCL axiom is omitted. For finite model finding, the ACYCL axiom, together with EXHAUST, is equivalent to INDUCT, but it can be expressed comfortably in Kodkod's logic.

Cyclic objects are displayed as $\xi_1, \xi_2, \ldots$, and their internal structure is shown under the "Datatypes" heading. For the benefit of users who have not read the manual, Nitpick detects structural induction steps and gives a hint to the user, as we saw in Section 3.2.

## 5 A More Advanced Example

The next example is taken from the Isabelle tutorial [13, pp. 9–15]. We want to prove that reversing a list twice yields the original list:

**theorem** REV_REV: $rev\ (rev\ ys) = ys$

The *rev* function is defined in terms of the append operator (@). Their equational specifications follow:

$$\begin{aligned} rev\ [] &= [] & [] \mathbin{@} ys &= ys \\ rev\ (x{\cdot}xs) &= rev\ xs \mathbin{@} [x] & (x{\cdot}xs) \mathbin{@} ys &= x \cdot (xs \mathbin{@} ys). \end{aligned}$$

The base case of the induction proof of REV_REV is easy to discharge using the *simp* method. For the induction step, we may assume

$$rev\ (rev\ zs) = zs \qquad (\mathcal{IH})$$

and the goal is

$$rev\ (rev\ (z{\cdot}zs)) = z{\cdot}zs. \qquad (\mathcal{G})$$

Applying *simp* rewrites the goal to

$$rev\ (rev\ zs \mathbin{@} [z]) = z{\cdot}zs \qquad (\mathcal{G}')$$

using the equational specification of *rev*. If we run Nitpick at this point, it does not find any standard countermodel, suggesting that $\mathcal{IH} \Longrightarrow \mathcal{G}'$ is valid. And if we instruct it to look for nonstandard countermodels, it quickly finds one:

$$\begin{aligned} \textit{Free variables:}\quad & z = a_1 \quad zs = \xi_1 \\ \textit{Datatype:}\quad & \alpha\ list = \{[],\ [a_1],\ \xi_1 = a_1{\cdot}\xi_2,\ \xi_2 = a_1{\cdot}\xi_1,\ \ldots\} \\ \textit{Constants:}\quad & rev = (\lambda x.\ ?)([] \mapsto [],\ [a_1] \mapsto [a_1],\ \xi_1 \mapsto \xi_1,\ \xi_2 \mapsto \xi_1) \\ & op\ @ = (\lambda x.\ ?)(([], []) \mapsto [],\ ([], [a_1]) \mapsto [a_1],\ ([], \xi_1) \mapsto \xi_1, \\ & \qquad ([], \xi_2) \mapsto \xi_2,\ ([a_1], []) \mapsto [a_1],\ ([a_1], \xi_1) \mapsto \xi_2, \\ & \qquad ([a_1], \xi_2) \mapsto \xi_1,\ (\xi_1, [a_1]) \mapsto \xi_1,\ (\xi_2, [a_1]) \mapsto \xi_2) \end{aligned}$$

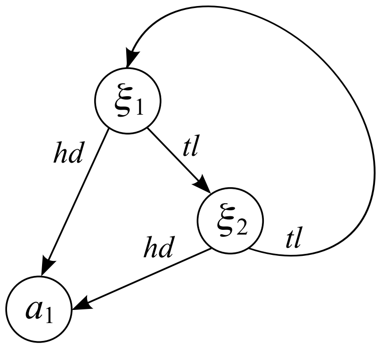


**Fig. 3.** Two nonstandard lists

The existence of the countermodel tells us that we must provide additional properties of *rev*, @, or both. It turns out the countermodel can provide some insight as to how to proceed. The model contains two distinct nonstandard lists, $\xi_1$ and $\xi_2$, that falsify the conjecture: $\mathit{rev}\ (\mathit{rev}\ \xi_2) = \xi_1$. The function table for @ contains the following information about them:

$$\begin{array}{ll} \xi_1\ @\ [a_1] = \xi_1 & \qquad [a_1]\ @\ \xi_1 = \xi_2 \\ \xi_2\ @\ [a_1] = \xi_2 & \qquad [a_1]\ @\ \xi_2 = \xi_1. \end{array}$$

The left column of the function table implies that both $\xi_1$ and $\xi_2$ represent infinite lists, because appending elements does not change them. The right column is depicted in Figure 3. Both lists $\xi_1$ and $\xi_2$ seem to only consist of the element $a_1$ repeated infinitely ($[a_1, a_1, \ldots]$), and yet $\xi_1 \neq \xi_2$.

A striking property of the function table of @ is that the left and right columns are not symmetric. For standard finite lists, appending and prepending an element should be symmetric. We can involve the function *rev* to make this relationship explicit: Appending an element and reversing the resulting list should construct the same list as reversing the list first and prepending the element. This clearly does not hold for the nonstandard model: $\mathit{rev}\ (\xi_1\ @\ [a_1]) = \xi_1$ but $[a_1]\ @\ \mathit{rev}\ \xi_1 = \xi_2$.

We thus add and prove the following lemma.

**theorem** REV_SNOC: $\mathit{rev}\ (\mathit{xs}\ @\ [x]) = [x]\ @\ \mathit{rev}\ \mathit{xs}$
**by** (*induct xs*) *auto*

Equipped with this lemma, the induction step of $\mathit{rev}\ (\mathit{rev}\ \mathit{ys}) = \mathit{ys}$ is now straightforward to prove:

**case** (CONS *y ys*) **note** IH = *this*
**have** $\mathit{rev}\ (\mathit{rev}\ (y \cdot \mathit{ys})) = \mathit{rev}\ (\mathit{rev}\ \mathit{ys}\ @\ [y])$ **by** *simp*
**moreover have** $\ldots = y \cdot \mathit{rev}\ (\mathit{rev}\ \mathit{ys})$ **using** REV_SNOC .
**moreover have** $\ldots = y \cdot \mathit{ys}$ **using** IH **by** *simp*
**ultimately show** *?case* **by** *simp*

Admittedly, some imagination is required to come up with the REV_SNOC lemma. However, it is the same kind of reasoning (building an intuition and then generalizing) that is needed to repair non-theorems on the basis of standard counterexamples.

## 6 Evaluation

Evaluating our method directly would require observing Isabelle users and estimating how much time they saved (or wasted) thanks to it. We chose instead to benchmark our core procedure: finding nonstandard models to properties that require structural induction. To achieve this, we took all the theories from the Archive of Formal Proofs [10] that are based on Isabelle's *HOL* theory and that contain at least 5 theorems proved by structural induction. We assumed that every theorem that was proved by structural induction needed it (but took out a few obviously needless inductions). For every theorem, we invoked Nitpick with the *non_std* option and a time limit of 30 seconds.

The table below summarizes the results per theory.

| THEORY | FOUND | SUCCESS | THEORY | FOUND | SUCCESS |
|---|---|---|---|---|---|
| *Comp.-Except.-Correctly* | 8/8 | 100.0% | *Prog.-Conflict-Analysis* | 20/53 | 37.7% |
| *Huffman* | 27/28 | 96.4% | *Simpl* | 38/109 | 34.9% |
| *FOL-Fitting* | 31/38 | 81.6% | *Completeness* | 15/44 | 34.1% |
| *POPLmark-deBruijn* | 90/112 | 80.4% | *CoreC++* | 17/60 | 28.3% |
| *RSAPSS* | 33/47 | 70.2% | *Group-Ring-Module* | 41/151 | 27.2% |
| *HotelKeyCards* | 16/23 | 69.6% | *SIFPL* | 6/23 | 26.1% |
| *Presburger-Automata* | 12/19 | 63.2% | *BinarySearchTree* | 5/22 | 22.7% |
| *SATSolverVerification* | 106/172 | 61.6% | *Functional-Automata* | 9/41 | 22.0% |
| *AVL-Trees* | 8/13 | 61.5% | *Fermat3_4* | 2/12 | 16.7% |
| *Collections* | 33/58 | 56.9% | *Recursion-Theory-I* | 5/31 | 16.1% |
| *FeatherweightJava* | 9/16 | 56.2% | *Cauchy* | 1/9 | 11.1% |
| *BytecodeLogicJmlTypes* | 10/18 | 55.6% | *Coinductive* | 4/53 | 7.5% |
| *Flyspeck-Tame* | 92/166 | 55.4% | *Lazy-Lists-II* | 1/25 | 4.0% |
| *Tree-Automata* | 32/64 | 50.0% | *MiniML* | 0/98 | 0.0% |
| *MuchAdoAboutTwo* | 4/8 | 50.0% | *Ordinal* | 0/20 | 0.0% |
| *Verified-Prover* | 4/8 | 50.0% | *Integration* | 0/8 | 0.0% |
| *VolpanoSmith* | 3/6 | 50.0% | *Topology* | 0/5 | 0.0% |
| *NormByEval* | 16/41 | 39.0% | | | |

An entry $m/n$ indicates that Nitpick found a nonstandard model for $m$ of $n$ theorems proved by induction. The last column expresses the same result as a percentage.

The success rates vary greatly from theory to theory. Nitpick performed best on theories involving lists, trees, and terms (*Compiling-Exceptions-Correctly*, *Huffman*, *POPLmark-deBruijn*). It performed poorly on theories involving arithmetic (*Cauchy*, *Integration*), complex set-theoretic constructions (*Lazy-List-II*, *Ordinal*), a large state space (*CoreC++*, *SIFPL*, *Recursion-Theory-I*), or nondefinitional axioms (*MiniML*). This is consistent with previous experience with Nitpick for finding standard models [5].

The main reason for the failures of our method is the inherent limitations of model finding in general, rather than our definition of nonstandard models. Our tool Nitpick is essentially a finite model finder, so only finite (fragments of) nonstandard models can be found. Nonstandard models are slightly easier to find than standard models because they have fewer axioms to fulfill, but they otherwise present the same challenges to Nitpick. Users who are determined to employ Nitpick can customize its behavior using various options, adapt their theories to avoid difficult idioms, or run it in an unsound mode that finds more genuine countermodels but also some spurious ones.

At first glance, the theory *BinarySearchTrees* seemed perfectly suited to our method, so we were surprised by the very low success rate. It turns out *BinarySearchTree* uses the type *int* to label its leaf nodes. In Isabelle, *int* is not defined as a datatype, and Nitpick handles it specially. Hence, proofs by induction on the structure or height of the trees could be avoided by applying an induction-like principle on *int*. Replacing *int* with *nat* for the labels increased the theory's score to 16/22 (72.7%).

## 7 Discussion and Related Work

*Interpretation of Nonstandard Models.* Our method presents the user with a countermodel whenever it detects that a property is noninductive. In some cases, the user must understand the cyclic structure of the nonstandard values $\xi_i$ and see how they interfere with the induction step; in other cases, it is better to ignore the cyclic structures. Either way, it is usually difficult to isolate the relevant parts of the model and infer which properties were violated by the model and any other countermodel. The traditional approach of studying the form of the current goal to determine how to proceed always remains an alternative. Even then, the concrete nonstandard countermodel we compute provides added value, because we can use it to validate any assumption we want to add.

*Inductiveness Modulo Theories.* Arguably, users rarely want to know whether the step $\mathcal{IH} \Longrightarrow \mathcal{G}$ can be performed without induction; rather, they have already proved various theorems $\mathcal{T}$ and want to know whether $\mathcal{IH} \wedge \mathcal{T} \Longrightarrow \mathcal{G}$ can be proved without induction. The theorems $\mathcal{T}$ could be all the theorems available in Isabelle's database (numbering in the thousands), or perhaps those that are known to Isabelle's automatic proof methods. Some users may also want to specify the set $\mathcal{T}$ themselves.

The main difficulty here is that these theorems are generally free-form universally quantified formulas and would occur on the left-hand side of an implication: If any of the universal variables range over an infinite type, Nitpick gives up immediately and enters an unsound mode in which the quantifiers are artificially bounded. Counterexamples are then marked as "potential." Infinite universal quantification is problematic for any finite model finder. We have yet to try infinite (deductive) model finding [6] on this type of problem.

On the other hand, users can instantiate the relevant theorems and add them as assumptions. This requires guessing the proper instantiations for the theorem's universal variables. Nitpick can be quite helpful when trying different instantiations.

*Possible Application to First-Order Proof Search.* Isabelle includes a tool called Sledgehammer that translates the current HOL goal to first-order logic and dispatches it to automatic theorem provers (ATPs) [11, 12]. The tool heuristically selects theorems from Isabelle's database and encodes these along with the goal. Using nonstandard model finding, it should sometimes be possible to determine that no first-order proof of a goal exists and use this information to guide the theorem selection. Unfortunately, this suffers from the same limitations as "inductiveness modulo theories" described above.

*Alternative Approach: A Junk Constructor.* Our first attempt at detecting noninductive properties was also rooted in the world of nonstandard models, but instead of allowing cyclic values we added an extra constructor $\mathit{Junk} :: \sigma \to \tau$ to each datatype $\tau$, where $\sigma$ is a fresh type. The values constructed with *Junk* were displayed as $\xi_1, \xi_2, \ldots,$ to the user. For many examples the results are essentially the same as with the "cyclic value" approach, but the approaches behave differently for two classes of properties: (1) For properties that can be proved with case distinction alone, the "extra constructor" approach leads to spurious countermodels. (2) For properties that follow from the acyclicity of constructors, the "extra constructor" approach fails to exhibit counterexamples. An example of such a property is $\mathit{Suc}\ n \neq n$, which can only be falsified by a cyclic $n$.

The "extra constructor" approach also has its merits: It relieves the user from having to reason about cyclic structures, and it is easier to implement in counterexample generators such as Quickcheck [2] that translate HOL datatypes directly to ML datatypes.

*More Related Work.* There are two pieces of related work that have directly inspired the ideas behind this paper. The first is work by Claessen and Svensson on different kinds of counterexample generation in the context of reachability [8]; the rule induction example presented in Section 3.1 is a direct adaptation of their approach. The second inspiration is work by Ahrendt on counterexample generation in the context of specifications of datatypes [1]. Ahrendt finds finite (and thus "nonstandard") counterexamples by weakening the specifications, which would otherwise only admit infinite models. Ahrendt's work was not done in the context of induction, and his notion of nonstandard models is thus very different from ours.

There exists a lot of prior work on automating induction proofs. For example, using *rippling* [16], failed proof attempts are analyzed and may lead to candidates for generalized induction hypotheses or new lemmas. Work in this area has the same general aim as ours: providing useful feedback to provers (humans or machines) who get stuck in induction proofs. Our approach differs most notably with this work in that it provides definite feedback to the prover about the inadequacy of the used induction technique. When our method generates a counterexample, it is certain that more induction is needed to proceed with the proof, if the conjecture is provable at all. Another difference is that we focus on counterexamples rather than on failed proof attempts. However, our longer-term hope is to exploit the nonstandard counterexamples in an automatic induction prover. This remains future work.

## 8 Conclusion

We described a procedure for automatically detecting that a structural induction hypothesis is too weak to support the induction step when proving a theorem, and explained how we modified the model finder Nitpick for Isabelle/HOL to support it. The procedure is based on the concept of nonstandard models of datatypes. The tight integration with a model finder allows for precise feedback in the form of a concrete counterexample that indicates why the induction step fails.

Although our focus is on interactive theorem proving, our approach is also applicable to automatic inductive theorem provers. In particular, the nonstandard models produced by the method contain a wealth of information that could be used to guide the search for an induction proof. An exciting direction for future work would be to see how to exploit this information automatically.

***Acknowledgment.*** We want to thank Tobias Nipkow for sponsoring this collaboration between Gothenburg and Munich, Sascha Böhme for helping with the implementation, and Alexander Krauss, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements.

## References

1. W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In *Proc. of 18th Int. Conf. on Automated Deduction (CADE)*. Springer, 2002.
2. 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.
3. S. Berghofer and M. Wenzel. Inductive datatypes in HOL—lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds., *TPHOLs '99*, vol. 1690 of *LNCS*, pp. 19–36, 1999.
4. J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. In G. Fraser and A. Gargantini, eds., *TAP 2010*, LNCS. Springer, 2010. To appear.
5. 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, eds., *ITP-10*, LNCS. Springer, 2010. To appear.
6. R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*, vol. 31 of *Applied Logic*. Springer, 2004.
7. A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5:56–68, 1940.
8. K. Claessen and H. Svensson. Finding counter examples in induction proofs. In B. Beckert and R. Hähnle, eds., *TAP 2008*, vol. 4966 of *LNCS*, pp. 48–65. Springer, 2008.
9. M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
10. G. Klein, T. Nipkow, and L. Paulson. The archive of formal proofs. `http://afp.sf.net/`.
11. J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Auto. Reas.*, 40(1):35–60, 2008.
12. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
14. S. Owre and N. Shankar. Abstract datatypes in PVS. Technical report, SRI, 1993.
15. T. Skolem. Über die Nicht-charakterisierbarkeit der Zahlenreihe mittels endlich oder abzählbar unendlich vieler Aussagen mit ausschließlich Zahlenvariablen. *Fundam. Math.*, 23:150–161, 1934.
16. J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Proc. 8th Int. Workshop on Logic Based Program Synthesis and Transformation*, 1998.
17. 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.
18. M. Wenzel and F. Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *J. Auto. Reas.*, 29(3–4):389–411, 2002.