

Picking Nits

A User's Guide to Nitpick for Isabelle/HOL

Jasmin Blanchette
Institut für Informatik, Technische Universität München

October 24, 2026

Contents

1 Introduction **2**

2 Installation **3**

3 First Steps **4**
3.1 Propositional Logic 4
3.2 Type Variables 5
3.3 Constants 6
3.4 Skolemization 7
3.5 Natural Numbers and Integers 8
3.6 Inductive Datatypes 10
3.7 Typedefs, Quotient Types, Records, Rationals, and Reals . . . 12
3.8 Inductive and Coinductive Predicates 14
3.9 Coinductive Datatypes 18

3.10 Boxing 21
3.11 Scope Monotonicity 23
3.12 Inductive Properties 25

4 Case Studies **26**
4.1 A Context-Free Grammar 27
4.2 AA Trees 29

5 Option Reference **33**
5.1 Mode of Operation 34
5.2 Scope of Search 35
5.3 Output Format 39
5.4 Regression Testing 41
5.5 Optimizations 42
5.6 Timeouts 46

6 Attribute Reference **46**

7 Standard ML Interface **49**
7.1 Invoking Nitpick 49
7.2 Registering Term Postprocessors 50
7.3 Registering Coinductive Datatypes 50

8 Known Bugs and Limitations **51**

1 Introduction

Nitpick [3] is a counterexample generator for Isabelle/HOL [5] that is designed to handle formulas combining (co)inductive datatypes, (co)inductively defined predicates, and quantifiers. It builds on Kodkod [7], a highly optimized first-order relational model finder developed by the Software Design Group at MIT. It is conceptually similar to Refute [8], from which it borrows many ideas and code fragments, but it benefits from Kodkod's optimizations and a new encoding scheme. The name Nitpick is shamelessly appropriated from a now retired Alloy precursor.

Nitpick is easy to use—you simply enter **nitpick** after a putative theorem and wait a few seconds. Nonetheless, there are situations where knowing how it works under the hood and how it reacts to various options helps increase the test coverage. This manual also explains how to install the tool on your

workstation. Should the motivation fail you, think of the many hours of hard work Nitpick will save you. Proving non-theorems is *hard work*.

Another common use of Nitpick is to find out whether the axioms of a locale are satisfiable, while the locale is being developed. To check this, it suffices to write

> **lemma** "*False*"
> **nitpick** [*show_all*]

after the locale's **begin** keyword. To falsify *False*, Nitpick must find a model for the axioms. If it finds no model, we have an indication that the axioms might be unsatisfiable.

Nitpick provides an automatic mode that can be enabled via the "Auto Nitpick" option under "Plugins > Plugin Options > Isabelle > General" in Isabelle/jEdit. In this mode, Nitpick is run on every newly entered theorem.

To run Nitpick, you must also make sure that the theory *Nitpick* is imported—this is rarely a problem in practice since it is part of *Main*. The examples presented in this manual can be found in Isabelle's `src/HOL/Nitpick_Examples/Manual_Nits.thy` theory. The known bugs and limitations at the time of writing are listed in §8. Comments and bug reports concerning the tool or the manual should be directed to the author at `jasmin.blanchette@inria.fr`.

Acknowledgment. The author would like to thank Mark Summerfield for suggesting several textual improvements.

2 Installation

Nitpick is part of Isabelle, so you do not need to install it. It relies on a third-party Kodkod front-end called Kodkodi, which in turn requires a Java virtual machine. Both are provided as official Isabelle components.

To check whether Kodkodi is successfully installed, you can try out the example in §3.1.

3 First Steps

This section introduces Nitpick by presenting small examples. If possible, you should try out the examples on your workstation. Your theory file should start as follows:

theory *Scratch*
imports *Main Quotient_Product RealDef*
begin

The results presented here were obtained using the JNI (Java Native Interface) version of MiniSat and with multithreading disabled to reduce nondeterminism. This was done by adding the line

nitpick_params [*sat_solver* = *MiniSat_JNI*, *max_threads* = 1]

after the **begin** keyword. The JNI version of MiniSat is bundled with Kodkodi and is precompiled for Linux, Mac OS X, and Windows (Cygwin). Other SAT solvers can also be used, as explained in §5.5. If you have already configured SAT solvers in Isabelle (e.g., for Refute), these will also be available to Nitpick.

3.1 Propositional Logic

Let's start with a trivial example from propositional logic:

lemma "$P \longleftrightarrow Q$"
nitpick

You should get the following output:

Nitpick found a counterexample:

Free variables:
P = *True*
Q = *False*

Nitpick can also be invoked on individual subgoals, as in the example below:

apply *auto*

goal (2 subgoals):
1. $P \Longrightarrow Q$
2. $Q \Longrightarrow P$

nitpick 1

Nitpick found a counterexample:

Free variables:
$P = \mathit{True}$
$Q = \mathit{False}$

nitpick 2

Nitpick found a counterexample:

Free variables:
$P = \mathit{False}$
$Q = \mathit{True}$

oops

3.2 Type Variables

If you are left unimpressed by the previous example, don't worry. The next one is more mind- and computer-boggling:

lemma "$x \in A \Longrightarrow (\text{THE } y.\ y \in A) \in A$"

The putative lemma involves the definite description operator, THE, presented in section 5.10.1 of the Isabelle tutorial [5]. The operator is defined by the axiom $(\text{THE } x.\ x = a) = a$. The putative lemma is merely asserting the indefinite description operator axiom with THE substituted for SOME.

The free variable x and the bound variable y have type $'a$. For formulas containing type variables, Nitpick enumerates the possible domains for each type variable, up to a given cardinality (10 by default), looking for a finite countermodel:

nitpick [*verbose*]

Trying 10 scopes:
card 'a = 1;
card 'a = 2;
⋮
card 'a = 10

Nitpick found a counterexample for card 'a = 3:

Free variables:
$A = \{a_2,\ a_3\}$
$x = a_3$

Total time: 963 ms

Nitpick found a counterexample in which $'a$ has cardinality 3. (For cardinalities 1 and 2, the formula holds.) In the counterexample, the three values of type $'a$ are written a_1, a_2, and a_3.

The message "Trying n scopes: ..." is shown only if the option *verbose* is enabled. You can specify *verbose* each time you invoke **nitpick**, or you can set it globally using the command

nitpick_params [*verbose*]

This command also displays the current default values for all of the options supported by Nitpick. The options are listed in §5.

3.3 Constants

By just looking at Nitpick's output, it might not be clear why the counterexample in §3.2 is genuine. Let's invoke Nitpick again, this time telling it to show the values of the constants that occur in the formula:

lemma "$x \in A \Longrightarrow (\text{THE } y.\ y \in A) \in A$"
nitpick [*show_consts*]

Nitpick found a counterexample for card $'a$ = 3:

Free variables:
$A = \{a_2,\ a_3\}$
$x = a_3$
Constant:
THE $y.\ y \in A = a_1$

As the result of an optimization, Nitpick directly assigned a value to the subterm THE $y.\ y \in A$, rather than to the *The* constant. We can disable this optimization by using the command

nitpick [*dont_specialize*, *show_consts*]

Our misadventures with THE suggest adding '$\exists!x.$' ("there exists a unique x such that") at the front of our putative lemma's assumption:

lemma "$\exists!x.\ x \in A \Longrightarrow (\text{THE } y.\ y \in A) \in A$"

The fix appears to work:

nitpick

Nitpick found no counterexample

We can further increase our confidence in the formula by exhausting all cardinalities up to 50:

nitpick [*card 'a* = 1–50][1]

Nitpick found no counterexample.

Let's see if Sledgehammer can find a proof:

sledgehammer

Sledgehammer: "e" on goal
Try this: **by** *(metis theI) (42 ms)*
⋮

by (*metis theI*)

This must be our lucky day.

3.4 Skolemization

Are all invertible functions onto? Let's find out:

lemma "$\exists g.\ \forall x.\ g\ (f\ x) = x \Longrightarrow \forall y.\ \exists x.\ y = f\ x$"
nitpick

Nitpick found a counterexample for card 'a = 2 and card 'b = 1:

Free variable:
$f = (\lambda x.\ _)(b_1 := a_1)$
Skolem constants:
$g = (\lambda x.\ _)(a_1 := b_1,\ a_2 := b_1)$
$y = a_2$

(The Isabelle/HOL notation $f(x := y)$ denotes the function that maps x to y and that otherwise behaves like f.) Although f is the only free variable occurring in the formula, Nitpick also displays values for the bound variables g and

[1]The symbol '–' is entered as - (hyphen).

y. These values are available to Nitpick because it performs skolemization as a preprocessing step.

In the previous example, skolemization only affected the outermost quantifiers. This is not always the case, as illustrated below:

lemma "$\exists x.\ \forall f.\ f\ x = x$"
nitpick

Nitpick found a counterexample for card 'a = 2:

Skolem constant:
$\lambda x.\ f = (\lambda x.\ _)(a_1 := (\lambda x.\ _)(a_1 := a_2,\ a_2 := a_1),$
$a_2 := (\lambda x.\ _)(a_1 := a_1,\ a_2 := a_1))$

The variable f is bound within the scope of x; therefore, f depends on x, as suggested by the notation $\lambda x.\, f$. If $x = a_1$, then f is the function that maps a_1 to a_2 and vice versa; otherwise, $x = a_2$ and f maps both a_1 and a_2 to a_1. In both cases, $f\ x \neq x$.

The source of the Skolem constants is sometimes more obscure:

lemma "*refl r* $\Longrightarrow$ *sym r*"
nitpick

Nitpick found a counterexample for card 'a = 2:

Free variable:
$r = \{(a_1, a_1),\ (a_2, a_1),\ (a_2, a_2)\}$
Skolem constants:
$sym.x = a_2$
$sym.y = a_1$

What happened here is that Nitpick expanded *sym* to its definition:

$$sym\ r \equiv \forall x\ y.\ (x, y) \in r \longrightarrow (y, x) \in r.$$

As their names suggest, the Skolem constants $sym.x$ and $sym.y$ are simply the bound variables x and y from *sym*'s definition.

3.5 Natural Numbers and Integers

Because of the axiom of infinity, the type *nat* does not admit any finite models. To deal with this, Nitpick's approach is to consider finite subsets N of *nat* and maps all numbers $\notin N$ to the undefined value (displayed as

'_'). The type *int* is handled similarly. Internally, undefined values lead to a three-valued logic.

Here is an example involving *int*:

> **lemma** "$[\![i \leq j;\ n \leq (m::int)]\!] \Longrightarrow i * n + j * m \leq i * m + j * n$"
> **nitpick**
>
> *Nitpick found a counterexample:*
>
> *Free variables:*
> $i = 0$
> $j = 1$
> $m = 1$
> $n = 0$

Internally, Nitpick uses either a unary or a binary representation of numbers. The unary representation is more efficient but only suitable for numbers very close to zero. By default, Nitpick attempts to choose the more appropriate encoding by inspecting the formula at hand. This behavior can be overridden by passing either *unary_ints* or *binary_ints* as option. For binary notation, the number of bits to use can be specified using the *bits* option. For example:

> **nitpick** [*binary_ints*, *bits* = 16]

With infinite types, we don't always have the luxury of a genuine counterexample and must often content ourselves with a potentially spurious one. For example:

> **lemma** "$\forall n.\ Suc\ n \neq n \Longrightarrow P$"
> **nitpick** [*card nat* = 50]
>
> *Warning: The conjecture either trivially holds for the given scopes or lies outside Nitpick's supported fragment; only potentially spurious counterexamples may be found*
>
> *Nitpick found a potentially spurious counterexample:*
>
> *Free variable:*
> $P = False$

The issue is that the bound variable in $\forall n.\ Suc\ n \neq n$ ranges over an infinite type. If Nitpick finds an n such that $Suc\ n = n$, it evaluates the assumption to *False*; but otherwise, it does not know anything about values of $n \geq card\ nat$ and must therefore evaluate the assumption to _, not *True*. Since the assumption can never be fully satisfied by Nitpick, the putative lemma can never be falsified.

Some conjectures involving elementary number theory make Nitpick look like a giant with feet of clay:

lemma "*P Suc*"
nitpick

Nitpick found no counterexample

On any finite set N, *Suc* is a partial function; for example, if $N = \{0, 1, \ldots, k\}$, then *Suc* is $\{0 \mapsto 1, 1 \mapsto 2, \ldots, k \mapsto _\}$, which evaluates to _ when passed as argument to P. As a result, P *Suc* is always _. The next example is similar:

lemma "P (*op* +::*nat* ⇒ *nat* ⇒ *nat*)"
nitpick [*card nat* = 1]

Nitpick found a counterexample:

Free variable:
$P = (\lambda x.\ _)((\lambda x.\ _)(0 := (\lambda x.\ _)(0 := 0)) := \mathit{False})$

nitpick [*card nat* = 2]

Nitpick found no counterexample.

The problem here is that *op* + is total when *nat* is taken to be $\{0\}$ but becomes partial as soon as we add 1, because $1 + 1 \notin \{0, 1\}$.

Because numbers are infinite and are approximated using a three-valued logic, there is usually no need to systematically enumerate domain sizes. If Nitpick cannot find a genuine counterexample for *card nat* = k, it is very unlikely that one could be found for smaller domains. (The P (*op* +) example above is an exception to this principle.) Nitpick nonetheless enumerates all cardinalities from 1 to 10 for *nat*, mainly because smaller cardinalities are fast to handle and give rise to simpler counterexamples. This is explained in more detail in §3.11.

3.6 Inductive Datatypes

Like natural numbers and integers, inductive datatypes with recursive constructors admit no finite models and must be approximated by a subterm-closed subset. For example, using a cardinality of 10 for *'a list*, Nitpick looks for all counterexamples that can be built using at most 10 different lists.

Let's see with an example involving *hd* (which returns the first element of a list) and @ (which concatenates two lists):

lemma "*hd* (*xs* @ [*y*, *y*]) = *hd xs*"
nitpick

Nitpick found a counterexample for card '*a* = 3*:*

Free variables:
$xs = []$
$y = a_1$

To see why the counterexample is genuine, we enable *show_consts* and *show_datatypes*:

Type:
'*a list* = {[], [a_1], [a_1, a_1], ...}
Constants:
$\lambda x_1.\ x_1 \mathbin{@} [y, y] = (\lambda x.\ _)([] := [a_1, a_1])$
$hd = (\lambda x.\ _)([] := a_2,\ [a_1] := a_1,\ [a_1, a_1] := a_1)$

Since *hd* [] is undefined in the logic, it may be given any value, including a_2.

The second constant, $\lambda x_1.\ x_1 \mathbin{@} [y, y]$, is simply the append operator whose second argument is fixed to be $[y, y]$. Appending $[a_1, a_1]$ to $[a_1]$ would normally give $[a_1, a_1, a_1]$, but this value is not representable in the subset of '*a list* considered by Nitpick, which is shown under the "Type" heading; hence the result is _. Similarly, appending $[a_1, a_1]$ to itself gives _.

Given *card* '*a* = 3 and *card* '*a list* = 3, Nitpick considers the following subsets:

{[], [a_1], [a_2]};	{[], [a_1], [a_2, a_1]};	{[], [a_2], [a_3, a_2]};
{[], [a_1], [a_3]};	{[], [a_1], [a_3, a_1]};	{[], [a_3], [a_1, a_3]};
{[], [a_2], [a_3]};	{[], [a_2], [a_1, a_2]};	{[], [a_3], [a_2, a_3]};
{[], [a_1], [a_1, a_1]};	{[], [a_2], [a_2, a_2]};	{[], [a_3], [a_3, a_3]}.

All subterm-closed subsets of '*a list* consisting of three values are listed and only those. As an example of a non-subterm-closed subset, consider $\mathcal{S} = \{[], [a_1], [a_1, a_2]\}$, and observe that $[a_1, a_2]$ (i.e., $a_1 \# [a_2]$) has $[a_2] \notin \mathcal{S}$ as a subterm.

Here's another möchtegern-lemma that Nitpick can refute without a blink:

lemma "$[\![length\ xs = 1;\ length\ ys = 1]\!] \Longrightarrow xs = ys$"
nitpick [*show_types*]

Nitpick found a counterexample for $card\ 'a = 3$*:*

Free variables:
$xs = [a_2]$
$ys = [a_1]$
Types:
$nat = \{0, 1, 2, \ldots\}$
$'a\ list = \{[], [a_1], [a_2], \ldots\}$

Because datatypes are approximated using a three-valued logic, there is usually no need to systematically enumerate cardinalities: If Nitpick cannot find a genuine counterexample for $card\ 'a\ list = 10$, it is very unlikely that one could be found for smaller cardinalities.

3.7 Typedefs, Quotient Types, Records, Rationals, and Reals

Nitpick generally treats types declared using **typedef** as datatypes whose single constructor is the corresponding *Abs_* function. For example:

typedef *three* = "{0::*nat*, 1, 2}"
by *blast*

definition A :: *three* **where** "$A \equiv$ *Abs_three* 0"
definition B :: *three* **where** "$B \equiv$ *Abs_three* 1"
definition C :: *three* **where** "$C \equiv$ *Abs_three* 2"

lemma "$[\![A \in X;\ B \in X]\!] \Longrightarrow c \in X$"
nitpick [*show_types*]

Nitpick found a counterexample:

Free variables:
$X = \{\langle\!\langle\, 0 \,\rangle\!\rangle, \langle\!\langle\, 1 \,\rangle\!\rangle\}$
$c = \langle\!\langle\, 2 \,\rangle\!\rangle$
Types:
$nat = \{0, 1, 2, \ldots\}$
$three = \{\langle\!\langle\, 0 \,\rangle\!\rangle, \langle\!\langle\, 1 \,\rangle\!\rangle, \langle\!\langle\, 2 \,\rangle\!\rangle, \ldots\}$

In the output above, $\langle\!\langle\, n \,\rangle\!\rangle$ abbreviates *Abs_three* n.

Quotient types are handled in much the same way. The following fragment defines the integer type *my_int* by encoding the integer x by a pair of natural numbers (m, n) such that $x + n = m$:

fun *my_int_rel* **where**
"*my_int_rel* (*x*, *y*) (*u*, *v*) = (*x* + *v* = *u* + *y*)"

quotient_type *my_int* = "*nat* × *nat*" / *my_int_rel*
by (*auto simp add*: *equivp_def fun_eq_iff*)

definition *add_raw* **where**
"*add_raw* ≡ λ(*x*, *y*) (*u*, *v*). (*x* + (*u*::*nat*), *y* + (*v*::*nat*))"

quotient_definition "*add*::*my_int* ⇒ *my_int* ⇒ *my_int*" **is** *add_raw*

lemma "*add x y* = *add x x*"
nitpick [*show_types*]

Nitpick found a counterexample:

Free variables:
x = « (0, 0) »
y = « (0, 1) »
Types:
nat = {0, 1, ...}
nat × *nat* [*boxed*] = {(0, 0), (1, 0), ...}
my_int = {« (0, 0) », « (0, 1) », ...}

The values « (0, 0) » and « (0, 1) » represent the integers 0 and −1, respectively. Other representants would have been possible—e.g., « (5, 5) » and « (11, 12) ». If we are going to use *my_int* extensively, it pays off to install a term postprocessor that converts the pair notation to the standard mathematical notation:

ML {*
fun *my_int_postproc* _ _ _ *T* (*Const* _ $ (*Const* _ $ *t1* $ *t2*)) =
HOLogic.mk_number T (*snd* (*HOLogic.dest_number t1*)
− *snd* (*HOLogic.dest_number t2*))
| *my_int_postproc* _ _ _ _ *t* = *t*
*}

declaration {*
Nitpick_Model.register_term_postprocessor @{typ *my_int*}
my_int_postproc
*}

Records are handled as datatypes with a single constructor:

record *point* =
Xcoord :: *int*

Ycoord :: *int*

lemma "*Xcoord* (*p*::*point*) = *Xcoord* (*q*::*point*)"
nitpick [*show_types*]

Nitpick found a counterexample:

Free variables:
$p = (\!|Xcoord = 1,\ Ycoord = 1|\!)$
$q = (\!|Xcoord = 0,\ Ycoord = 0|\!)$
Types:
$int = \{0, 1, \ldots\}$
$point = \{(\!|Xcoord = 0,\ Ycoord = 0|\!),$
$(\!|Xcoord = 1,\ Ycoord = 1|\!), \ldots\}$

Finally, Nitpick provides rudimentary support for rationals and reals using a similar approach:

lemma "$4 * x + 3 * (y$::*real*$) \neq 1/2$"
nitpick [*show_types*]

Nitpick found a counterexample:

Free variables:
$x = 1/2$
$y = -1/2$
Types:
$nat = \{0, 1, 2, 3, 4, 5, 6, 7, \ldots\}$
$int = \{-3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$
$real = \{-3/2, -1/2, 0, 1/2, 1, 2, 3, 4, \ldots\}$

3.8 Inductive and Coinductive Predicates

Inductively defined predicates (and sets) are particularly problematic for counterexample generators. They can make Quickcheck [2] loop forever and Refute [8] run out of resources. The crux of the problem is that they are defined using a least fixed-point construction.

Nitpick's philosophy is that not all inductive predicates are equal. Consider the *even* predicate below:

inductive *even* **where**
"*even* 0" |
"*even n* $\Longrightarrow$ *even* (*Suc* (*Suc n*))"

This predicate enjoys the desirable property of being well-founded, which means that the introduction rules don't give rise to infinite chains of the form

$$\cdots \Longrightarrow \mathit{even}\ k'' \Longrightarrow \mathit{even}\ k' \Longrightarrow \mathit{even}\ k.$$

For *even*, this is obvious: Any chain ending at k will be of length $k/2 + 1$:

$$\mathit{even}\ 0 \Longrightarrow \mathit{even}\ 2 \Longrightarrow \cdots \Longrightarrow \mathit{even}\ (k-2) \Longrightarrow \mathit{even}\ k.$$

Wellfoundedness is desirable because it enables Nitpick to use a very efficient fixed-point computation.[2] Moreover, Nitpick can prove wellfoundedness of most well-founded predicates, just as Isabelle's **function** package usually discharges termination proof obligations automatically.

Let's try an example:

> **lemma** "$\exists n.\ \mathit{even}\ n \wedge \mathit{even}\ (\mathit{Suc}\ n)$"
> **nitpick** [*card nat* = 50, *unary_ints*, *verbose*]
>
> *The inductive predicate "even" was proved well-founded; Nitpick can compute it efficiently*
>
> *Trying 1 scope:*
> *card nat = 50.*
>
> *Warning: The conjecture either trivially holds for the given scopes or lies outside Nitpick's supported fragment; only potentially spurious counterexamples may be found*
>
> *Nitpick found a potentially spurious counterexample for card nat = 50:*
>
> *Empty assignment*
>
> *Nitpick could not find a better counterexample It checked 1 of 1 scope*
>
> *Total time: 1.62 s.*

No genuine counterexample is possible because Nitpick cannot rule out the existence of a natural number $n \geq 50$ such that both *even n* and *even* (*Suc n*) are true. To help Nitpick, we can bound the existential quantifier:

> **lemma** "$\exists n \leq 49.\ \mathit{even}\ n \wedge \mathit{even}\ (\mathit{Suc}\ n)$"
> **nitpick** [*card nat* = 50, *unary_ints*]

[2] If an inductive predicate is well-founded, then it has exactly one fixed point, which is simultaneously the least and the greatest fixed point. In these circumstances, the computation of the least fixed point amounts to the computation of an arbitrary fixed point, which can be performed using a straightforward recursive equation.

Nitpick found a counterexample:

Empty assignment

So far we were blessed by the wellfoundedness of *even*. What happens if we use the following definition instead?

inductive *even′* **where**
"*even′* (0::*nat*)" |
"*even′* 2" |
"⟦*even′ m*; *even′ n*⟧ ⟹ *even′* (*m* + *n*)"

This definition is not well-founded: From *even′* 0 and *even′* 0, we can derive that *even′* 0. Nonetheless, the predicates *even* and *even′* are equivalent.

Let's check a property involving *even′*. To make up for the foreseeable computational hurdles entailed by non-wellfoundedness, we decrease *nat*'s cardinality to a mere 10:

lemma "∃*n* ∈ {0, 2, 4, 6, 8}. ¬ *even′ n*"
nitpick [*card nat* = 10, *verbose*, *show_consts*]

The inductive predicate "even′" could not be proved well-founded; Nitpick might need to unroll it

Trying 6 scopes:
card nat = 10 and iter even′ = 0;
card nat = 10 and iter even′ = 1;
card nat = 10 and iter even′ = 2;
card nat = 10 and iter even′ = 4;
card nat = 10 and iter even′ = 8;
card nat = 10 and iter even′ = 9

Nitpick found a counterexample for card nat = 10 and iter even′ = 2:

Constant:
$\lambda i.\ even' = (\lambda x.\ _)(0 := (\lambda x.\ _)(0 := True,\ 2 := True),$
$1 := (\lambda x.\ _)(0 := True,\ 2 := True,\ 4 := True),$
$2 := (\lambda x.\ _)(0 := True,\ 2 := True,\ 4 := True,$
$6 := True,\ 8 := True))$

Total time: 1.87 s.

Nitpick's output is very instructive. First, it tells us that the predicate is unrolled, meaning that it is computed iteratively from the empty set. Then it lists six scopes specifying different bounds on the numbers of iterations: 0, 1, 2, 4, 8, and 9.

The output also shows how each iteration contributes to *even*′. The notation $\lambda i.\ \mathit{even}'$ indicates that the value of the predicate depends on an iteration counter. Iteration 0 provides the basis elements, 0 and 2. Iteration 1 contributes 4 ($= 2 + 2$). Iteration 2 throws 6 ($= 2 + 4 = 4 + 2$) and 8 ($= 4 + 4$) into the mix. Further iterations would not contribute any new elements. The predicate *even*′ evaluates to either *True* or _, never *False*.

When unrolling a predicate, Nitpick tries 0, 1, 2, 4, 8, 12, 16, 20, 24, and 28 iterations. However, these numbers are bounded by the cardinality of the predicate's domain. With *card nat* = 10, no more than 9 iterations are ever needed to compute the value of a *nat* predicate. You can specify the number of iterations using the *iter* option, as explained in §5.2.

In the next formula, *even*′ occurs both positively and negatively:

lemma "$\mathit{even}'\ (n - 2) \Longrightarrow \mathit{even}'\ n$"
nitpick [*card nat* = 10, *show_consts*]

Nitpick found a counterexample:

Free variable:
$n = 1$
Constants:
$\lambda i.\ \mathit{even}' = (\lambda x.\ _)(0 := (\lambda x.\ _)(0 := \mathit{True}, 2 := \mathit{True}))$
$\mathit{even}' \leq (\lambda x.\ _)(0 := \mathit{True}, 1 := \mathit{False}, 2 := \mathit{True},$
$4 := \mathit{True}, 6 := \mathit{True}, 8 := \mathit{True})$

Notice the special constraint $\mathit{even}' \leq \ldots$ in the output, whose right-hand side represents an arbitrary fixed point (not necessarily the least one). It is used to falsify $\mathit{even}'\ n$. In contrast, the unrolled predicate is used to satisfy $\mathit{even}'\ (n - 2)$.

Coinductive predicates are handled dually. For example:

coinductive *nats* **where**
"$\mathit{nats}\ (x{::}\mathit{nat}) \Longrightarrow \mathit{nats}\ x$"

lemma "$\mathit{nats} = (\lambda n.\ n \in \{0, 1, 2, 3, 4\})$"
nitpick [*card nat* = 10, *show_consts*]

Nitpick found a counterexample:

Constants:
$\lambda i.\ \mathit{nats} = (\lambda x.\ _)(0 := (\lambda x.\ _), 1 := (\lambda x.\ _), 2 := (\lambda x.\ _))$
$\mathit{nats} \geq (\lambda x.\ _)(3 := \mathit{True}, 4 := \mathit{False}, 5 := \mathit{True})$

As a special case, Nitpick uses Kodkod's transitive closure operator to encode negative occurrences of non-well-founded "linear inductive predicates," i.e., inductive predicates for which each the predicate occurs in at most one assumption of each introduction rule. For example:

inductive *odd* **where**
"*odd* 1" |
"⟦*odd m*; *even n*⟧ ⟹ *odd* (*m* + *n*)"

lemma "*odd n* ⟹ *odd* (*n* − 2)"
nitpick [*card nat* = 4, *show_consts*]

Nitpick found a counterexample:

Free variable:
$n = 1$
Constants:
$even = (\lambda x.\ _)(0 := True, 1 := False, 2 := True, 3 := False)$
$odd_{\text{base}} =$
$(\lambda x.\ _)(0 := False,\ 1 := True,\ 2 := False,\ 3 := False)$
$odd_{\text{step}} = (\lambda x.\ _)$
$(0 := (\lambda x.\ _)(0 := True,\ 1 := False,\ 2 := True,\ 3 := False),$
$1 := (\lambda x.\ _)(0 := False,\ 1 := True,\ 2 := False,\ 3 := True),$
$2 := (\lambda x.\ _)(0 := False,\ 1 := False,\ 2 := True,\ 3 := False),$
$3 := (\lambda x.\ _)(0 := False,\ 1 := False,\ 2 := False,\ 3 := True))$
$odd \leq (\lambda x.\ _)(0 := False,\ 1 := True,\ 2 := False,\ 3 := True)$

In the output, odd_{base} represents the base elements and odd_{step} is a transition relation that computes new elements from known ones. The set *odd* consists of all the values reachable through the reflexive transitive closure of odd_{step} starting with any element from odd_{base}, namely 1 and 3. Using Kodkod's transitive closure to encode linear predicates is normally either more thorough or more efficient than unrolling (depending on the value of *iter*), but you can disable it by passing the *dont_star_linear_preds* option.

3.9 Coinductive Datatypes

A coinductive datatype is similar to an inductive datatype but allows infinite objects. Thus, the infinite lists $ps = [a, a, a, \ldots]$, $qs = [a, b, a, b, \ldots]$, and $rs = [0, 1, 2, 3, \ldots]$ can be defined as coinductive lists, or "lazy lists," using the *LNil* :: *'a llist* and *LCons* :: *'a* ⇒ *'a llist* ⇒ *'a llist* constructors.

Although it is otherwise no friend of infinity, Nitpick can find counterexamples involving cyclic lists such as *ps* and *qs* above as well as finite lists:

codatatype *'a llist = LNil | LCons 'a "'a llist"*

lemma "*xs ≠ LCons a xs*"
nitpick

Nitpick found a counterexample for card 'a = 1:

Free variables:
$a = a_1$
$xs = THE\ \omega.\ \omega = LCons\ a_1\ \omega$

The notation THE $\omega.\ \omega = t(\omega)$ stands for the infinite term $t(t(t(\ldots)))$. Hence, *xs* is simply the infinite list $[a_1, a_1, a_1, \ldots]$.

The next example is more interesting:

primcorec *iterates* **where**
"*iterates f a = LCons a (iterates f (f a))*"

lemma "$[\![xs = LCons\ a\ xs;\ ys = iterates\ (\lambda b.\ a)\ b]\!] \Longrightarrow xs = ys$"
nitpick [*verbose*]

The type 'a passed the monotonicity test; Nitpick might be able to skip some scopes

Trying 10 scopes:
card 'a = 1, card "'a list" = 1, and bisim_depth = 0;
⋮
card 'a = 10, card "'a list" = 10, and bisim_depth = 9

Nitpick found a counterexample for card 'a = 2, card "'a llist" = 2, and bisim_depth = 1:

Free variables:
$a = a_1$
$b = a_2$
$xs = THE\ \omega.\ \omega = LCons\ a_1\ \omega$
$ys = LCons\ a_2\ (THE\ \omega.\ \omega = LCons\ a_1\ \omega)$

Total time: 1.11 s

The lazy list *xs* is simply $[a_1, a_1, a_1, \ldots]$, whereas *ys* is $[a_2, a_1, a_1, a_1, \ldots]$, i.e., a lasso-shaped list with $[a_2]$ as its stem and $[a_1]$ as its cycle. In general, the list segment within the scope of the THE binder corresponds to the lasso's cycle, whereas the segment leading to the binder is the stem.

A salient property of coinductive datatypes is that two objects are considered equal if and only if they lead to the same observations. For example, the two lazy lists

$$\text{THE } \omega.\ \omega = \mathit{LCons}\ a\ (\mathit{LCons}\ b\ \omega)$$
$$\mathit{LCons}\ a\ (\text{THE } \omega.\ \omega = \mathit{LCons}\ b\ (\mathit{LCons}\ a\ \omega))$$

are identical, because both lead to the sequence of observations a, b, a, b, ... (or, equivalently, both encode the infinite list $[a, b, a, b, \ldots]$). This concept of equality for coinductive datatypes is called bisimulation and is defined coinductively.

Internally, Nitpick encodes the coinductive bisimilarity predicate as part of the Kodkod problem to ensure that distinct objects lead to different observations. This precaution is somewhat expensive and often unnecessary, so it can be disabled by setting the *bisim_depth* option to -1. The bisimilarity check is then performed *after* the counterexample has been found to ensure correctness. If this after-the-fact check fails, the counterexample is tagged as "quasi genuine" and Nitpick recommends to try again with *bisim_depth* set to a nonnegative integer.

The next formula illustrates the need for bisimilarity (either as a Kodkod predicate or as an after-the-fact check) to prevent spurious counterexamples:

lemma "$[\![xs = \mathit{LCons}\ a\ xs;\ \ ys = \mathit{LCons}\ a\ ys]\!] \Longrightarrow xs = ys$"
nitpick [*bisim_depth* $= -1$, *show_types*]

Nitpick found a quasi genuine counterexample for $\mathit{card}\ 'a = 2$*:*

Free variables:
$a = a_1$
$xs = \mathit{THE}\ \omega.\ \omega = \mathit{LCons}\ a_1\ \omega$
$ys = \mathit{THE}\ \omega.\ \omega = \mathit{LCons}\ a_1\ \omega$
Type:
$'a\ \mathit{llist} = \{\mathit{THE}\ \omega.\ \omega = \mathit{LCons}\ a_1\ \omega,$
$\mathit{THE}\ \omega.\ \omega = \mathit{LCons}\ a_1\ \omega,\ \ldots\}$

Try again with "bisim_depth" set to a nonnegative value to confirm that the counterexample is genuine

nitpick

Nitpick found no counterexample

In the first **nitpick** invocation, the after-the-fact check discovered that the two known elements of type $'a\ \mathit{llist}$ are bisimilar, prompting Nitpick to label

the example as only "quasi genuine."

A compromise between leaving out the bisimilarity predicate from the Kodkod problem and performing the after-the-fact check is to specify a low nonnegative *bisim_depth* value. In general, a value of K means that Nitpick will require all lists to be distinguished from each other by their prefixes of length K. However, setting K to a too low value can overconstrain Nitpick, preventing it from finding any counterexamples.

3.10 Boxing

Nitpick normally maps function and product types directly to the corresponding Kodkod concepts. As a consequence, if $'a$ has cardinality 3 and $'b$ has cardinality 4, then $'a \times 'b$ has cardinality 12 ($= 4 \times 3$) and $'a \Rightarrow 'b$ has cardinality 64 ($= 4^3$). In some circumstances, it pays off to treat these types in the same way as plain datatypes, by approximating them by a subset of a given cardinality. This technique is called "boxing" and is particularly useful for functions passed as arguments to other functions, for high-arity functions, and for large tuples. Under the hood, boxing involves wrapping occurrences of the types $'a \times 'b$ and $'a \Rightarrow 'b$ in isomorphic datatypes, as can be seen by enabling the *debug* option.

To illustrate boxing, we consider a formalization of λ-terms represented using de Bruijn's notation:

datatype *tm* = *Var nat* | *Lam tm* | *App tm tm*

The *lift t k* function increments all variables with indices greater than or equal to k by one:

primrec *lift* **where**
"*lift* (*Var j*) *k* = *Var* (if *j* < *k* then *j* else *j* + 1)" |
"*lift* (*Lam t*) *k* = *Lam* (*lift t* (*k* + 1))" |
"*lift* (*App t u*) *k* = *App* (*lift t k*) (*lift u k*)"

The *loose t k* predicate returns *True* if and only if term t has a loose variable with index k or more:

primrec *loose* **where**
"*loose* (*Var j*) *k* = (*j* ≥ *k*)" |
"*loose* (*Lam t*) *k* = *loose t* (*Suc k*)" |
"*loose* (*App t u*) *k* = (*loose t k* ∨ *loose u k*)"

Next, the *subst* σ t function applies the substitution σ on t:

primrec *subst* **where**
"*subst* σ (*Var* j) = σ j" |
"*subst* σ (*Lam* t) =
Lam (*subst* ($\lambda n.$ case n of 0 $\Rightarrow$ *Var* 0 | *Suc* m $\Rightarrow$ *lift* (σ m) 1) t)" |
"*subst* σ (*App* t u) = *App* (*subst* σ t) (*subst* σ u)"

A substitution is a function that maps variable indices to terms. Observe that σ is a function passed as argument and that Nitpick can't optimize it away, because the recursive call for the *Lam* case involves an altered version. Also notice the *lift* call, which increments the variable indices when moving under a *Lam*.

A reasonable property to expect of substitution is that it should leave closed terms unchanged. Alas, even this simple property does not hold:

lemma "$\neg$ *loose* t 0 $\Longrightarrow$ *subst* σ t = t"
nitpick [*verbose*]

Trying 10 scopes:
card nat = 1, *card tm* = 1, and *card "nat* $\Rightarrow$ *tm"* = 1;
card nat = 2, *card tm* = 2, and *card "nat* $\Rightarrow$ *tm"* = 2;
$\vdots$
card nat = 10, *card tm* = 10, and *card "nat* $\Rightarrow$ *tm"* = 10

Nitpick found a counterexample for card nat = 6, *card tm* = 6, *and card "nat* $\Rightarrow$ *tm"* = 6:

Free variables:
σ = ($\lambda x.$ _)(0 := *Var* 0, 1 := *Var* 0, 2 := *Var* 0,
3 := *Var* 0, 4 := *Var* 0, 5 := *Lam* (*Lam* (*Var* 0)))
t = *Lam* (*Lam* (*Var* 1))

Total time: 3.08 s

Using *eval*, we find out that *subst* σ t = *Lam* (*Lam* (*Var* 0)). Using the traditional λ-calculus notation, t is $\lambda x\, y.\ x$ whereas *subst* σ t is (wrongly) $\lambda x\, y.\ y$. The bug is in *subst*: The *lift* (σ m) 1 call should be replaced with *lift* (σ m) 0.

An interesting aspect of Nitpick's verbose output is that it assigned inceasing cardinalities from 1 to 10 to the type *nat* $\Rightarrow$ *tm* of the higher-order argument σ of *subst*. For the formula of interest, knowing 6 values of that type was

enough to find the counterexample. Without boxing, $6^6 = 46\,656$ values must be considered, a hopeless undertaking:

> **nitpick** [*dont_box*]
>
> *Nitpick ran out of time after checking 3 of 10 scopes*

Boxing can be enabled or disabled globally or on a per-type basis using the *box* option. Nitpick usually performs reasonable choices about which types should be boxed, but option tweaking sometimes helps.

3.11 Scope Monotonicity

The *card* option (together with *iter*, *bisim_depth*, and *max*) controls which scopes are actually tested. In general, to exhaust all models below a certain cardinality bound, the number of scopes that Nitpick must consider increases exponentially with the number of type variables (and **typedecl**'d types) occurring in the formula. Given the default cardinality specification of 1–10, no fewer than $10^4 = 10\,000$ scopes must be considered for a formula involving $'a$, $'b$, $'c$, and $'d$.

Fortunately, many formulas exhibit a property called *scope monotonicity*, meaning that if the formula is falsifiable for a given scope, it is also falsifiable for all larger scopes [4, p. 165].

Consider the formula

> **lemma** "$length\ xs = length\ ys \Longrightarrow rev\ (zip\ xs\ ys) = zip\ xs\ (rev\ ys)$"

where *xs* is of type $'a\ list$ and *ys* is of type $'b\ list$. A priori, Nitpick would need to consider 1 000 scopes to exhaust the specification *card* = 1–10 (10 cardinalies for $'a \times 10$ cardinalities for $'b \times 10$ cardinalities for the datatypes). However, our intuition tells us that any counterexample found with a small scope would still be a counterexample in a larger scope—by simply ignoring the fresh $'a$ and $'b$ values provided by the larger scope. Nitpick comes to the same conclusion after a careful inspection of the formula and the relevant definitions:

> **nitpick** [*verbose*]
>
> *The types 'a and 'b passed the monotonicity test; Nitpick might be able to skip some scopes.*
>
> *Trying 10 scopes:*

card 'a = 1, card 'b = 1, card nat = 1, card "('a × 'b) list" = 1,
card "'a list" = 1, and card "'b list" = 1;
card 'a = 2, card 'b = 2, card nat = 2, card "('a × 'b) list" = 2,
card "'a list" = 2, and card "'b list" = 2;
⋮
card 'a = 10, card 'b = 10, card nat = 10, card "('a × 'b) list" = 10,
card "'a list" = 10, and card "'b list" = 10

Nitpick found a counterexample for card 'a = 5, card 'b = 5, card nat = 5, card "('a × 'b) list" = 5, card "'a list" = 5, and card "'b list" = 5:

Free variables:
$xs = [a_1, a_2]$
$ys = [b_1, b_1]$

Total time: 1.63 s.

In theory, it should be sufficient to test a single scope:

nitpick [*card* = 10]

However, this is often less efficient in practice and may lead to overly complex counterexamples.

If the monotonicity check fails but we believe that the formula is monotonic (or we don't mind missing some counterexamples), we can pass the *mono* option. To convince yourself that this option is risky, simply consider this example from §3.4:

lemma "$\exists g.\ \forall x{::}'b.\ g\ (f\ x) = x \Longrightarrow \forall y{::}'a.\ \exists x.\ y = f\ x$"
nitpick [*mono*]

Nitpick found no counterexample

nitpick

Nitpick found a counterexample for card 'a = 2 and card 'b = 1:
⋮

(It turns out the formula holds if and only if *card 'a* ≤ *card 'b*.) Although this is rarely advisable, the automatic monotonicity checks can be disabled by passing *non_mono* (§5.5).

As insinuated in §3.5 and §3.6, *nat*, *int*, and inductive datatypes are normally monotonic and treated as such. The same is true for record types, *rat*, and *real*. Thus, given the cardinality specification 1–10, a formula involving *nat*,

int, *int list*, *rat*, and *rat list* will lead Nitpick to consider only 10 scopes instead of $10^4 = 10\,000$. On the other hand, **typedef**s and quotient types are generally nonmonotonic.

3.12 Inductive Properties

Inductive properties are a particular pain to prove, because the failure to establish an induction step can mean several things:

1. The property is invalid.
2. The property is valid but is too weak to support the induction step.
3. The property is valid and strong enough; it's just that we haven't found the proof yet.

Depending on which scenario applies, we would take the appropriate course of action:

1. Repair the statement of the property so that it becomes valid.
2. Generalize the property and/or prove auxiliary properties.
3. Work harder on a proof.

How can we distinguish between the three scenarios? Nitpick's normal mode of operation can often detect scenario 1, and Isabelle's automatic tactics help with scenario 3. Using appropriate techniques, it is also often possible to use Nitpick to identify scenario 2. Consider the following transition system, in which natural numbers represent states:

inductive_set *reach* **where**
"$(4::nat) \in reach$" |
"$[\![n < 4;\ n \in reach]\!] \Longrightarrow 3 * n + 1 \in reach$" |
"$n \in reach \Longrightarrow n + 2 \in reach$"

We will try to prove that only even numbers are reachable:

lemma "$n \in reach \Longrightarrow 2$ dvd n"

Does this property hold? Nitpick cannot find a counterexample within 30 seconds, so let's attempt a proof by induction:

apply (*induct set*: *reach*)
apply *auto*

This leaves us in the following proof state:

goal (2 subgoals):
1. $\bigwedge n.\ [\![n \in reach;\ n < 4;\ 2\ dvd\ n]\!] \Longrightarrow 2\ dvd\ Suc\ (3 * n)$
2. $\bigwedge n.\ [\![n \in reach;\ 2\ dvd\ n]\!] \Longrightarrow 2\ dvd\ Suc\ (Suc\ n)$

If we run Nitpick on the first subgoal, it still won't find any counterexample; and yet, *auto* fails to go further, and *arith* is helpless. However, notice the $n \in reach$ assumption, which strengthens the induction hypothesis but is not immediately usable in the proof. If we remove it and invoke Nitpick, this time we get a counterexample:

apply (*thin_tac* "$n \in reach$")
nitpick

Nitpick found a counterexample:

Skolem constant:
$n = 0$

Indeed, $0 < 4$, 2 divides 0, but 2 does not divide 1. We can use this information to strength the lemma:

lemma "$n \in reach \Longrightarrow 2$ dvd $n \vee n \neq 0$"

Unfortunately, the proof by induction still gets stuck, except that Nitpick now finds the counterexample $n = 2$. We generalize the lemma further to

lemma "$n \in reach \Longrightarrow 2$ dvd $n \vee n \geq 4$"

and this time *arith* can finish off the subgoals.

4 Case Studies

As a didactic device, the previous section focused mostly on toy formulas whose validity can easily be assessed just by looking at the formula. We will now review two somewhat more realistic case studies that are within Nitpick's reach: a context-free grammar modeled by mutually inductive sets and a functional implementation of AA trees. The results presented in this section were produced with the following settings:

nitpick_params [*max_potential* = 0]

4.1 A Context-Free Grammar

Our first case study is taken from section 7.4 in the Isabelle tutorial [5]. The following grammar, originally due to Hopcroft and Ullman, produces all strings with an equal number of a's and b's:

$S ::= \epsilon \mid bA \mid aB$
$A ::= aS \mid bAA$
$B ::= bS \mid aBB$

The intuition behind the grammar is that A generates all strings with one more a than b's and B generates all strings with one more b than a's.

The alphabet consists exclusively of a's and b's:

datatype *alphabet* = $a \mid b$

Strings over the alphabet are represented by *alphabet list*s. Nonterminals in the grammar become sets of strings. The production rules presented above can be expressed as a mutually inductive definition:

inductive_set S **and** A **and** B **where**
R1: "$[] \in S$" |
R2: "$w \in A \Longrightarrow b \# w \in S$" |
R3: "$w \in B \Longrightarrow a \# w \in S$" |
R4: "$w \in S \Longrightarrow a \# w \in A$" |
R5: "$w \in S \Longrightarrow b \# w \in S$" |
R6: "$[\![v \in B;\ v \in B]\!] \Longrightarrow a \# v @ w \in B$"

The conversion of the grammar into the inductive definition was done manually by Joe Blow, an underpaid undergraduate student. As a result, some errors might have sneaked in.

Debugging faulty specifications is at the heart of Nitpick's *raison d'être*. A good approach is to state desirable properties of the specification (here, that S is exactly the set of strings over $\{a, b\}$ with as many a's as b's) and check them with Nitpick. If the properties are correctly stated, counterexamples will point to bugs in the specification. For our grammar example, we will proceed in two steps, separating the soundness and the completeness of the set S. First, soundness:

theorem *S_sound*:
"$w \in S \longrightarrow length\ [x \leftarrow w.\ x = a] = length\ [x \leftarrow w.\ x = b]$"
nitpick

Nitpick found a counterexample:

Free variable:
$w = [b]$

It would seem that $[b] \in S$. How could this be? An inspection of the introduction rules reveals that the only rule with a right-hand side of the form $b \# \ldots \in S$ that could have introduced $[b]$ into S is *R5*:

"$w \in S \Longrightarrow b \# w \in S$"

On closer inspection, we can see that this rule is wrong. To match the production $B ::= bS$, the second S should be a B. We fix the typo and try again:

nitpick

Nitpick found a counterexample:

Free variable:
$w = [a, a, b]$

Some detective work is necessary to find out what went wrong here. To get $[a, a, b] \in S$, we need $[a, b] \in B$ by *R3*, which in turn can only come from *R6*:

"$[\![v \in B;\ v \in B]\!] \Longrightarrow a \# v \mathbin{@} w \in B$"

Now, this formula must be wrong: The same assumption occurs twice, and the variable w is unconstrained. Clearly, one of the two occurrences of v in the assumptions should have been a w.

With the correction made, we don't get any counterexample from Nitpick. Let's move on and check completeness:

theorem *S_complete*:
"*length* $[x \leftarrow w.\ x = a] =$ *length* $[x \leftarrow w.\ x = b] \longrightarrow w \in S$"
nitpick

Nitpick found a counterexample:

Free variable:
$w = [b, b, a, a]$

Apparently, $[b, b, a, a] \notin S$, even though it has the same numbers of a's and b's. But since our inductive definition passed the soundness check, the introduction rules we have are probably correct. Perhaps we simply lack an introduction rule. Comparing the grammar with the inductive definition, our

suspicion is confirmed: Joe Blow simply forgot the production $A ::= bAA$, without which the grammar cannot generate two or more b's in a row. So we add the rule

"$[\![v \in A;\ w \in A]\!] \Longrightarrow b \# v \mathbin{@} w \in A$"

With this last change, we don't get any counterexamples from Nitpick for either soundness or completeness. We can even generalize our result to cover A and B as well:

theorem *S_A_B_sound_and_complete*:
"$w \in S \longleftrightarrow \mathit{length}\ [x \leftarrow w.\ x = a] = \mathit{length}\ [x \leftarrow w.\ x = b]$"
"$w \in A \longleftrightarrow \mathit{length}\ [x \leftarrow w.\ x = a] = \mathit{length}\ [x \leftarrow w.\ x = b] + 1$"
"$w \in B \longleftrightarrow \mathit{length}\ [x \leftarrow w.\ x = b] = \mathit{length}\ [x \leftarrow w.\ x = a] + 1$"
nitpick

Nitpick found no counterexample

4.2 AA Trees

AA trees are a kind of balanced trees discovered by Arne Andersson that provide similar performance to red-black trees, but with a simpler implementation [1]. They can be used to store sets of elements equipped with a total order $<$. We start by defining the datatype and some basic extractor functions:

datatype *'a aa_tree* =
  Λ | *N* "*'a*::*linorder*" *nat* "*'a aa_tree*" "*'a aa_tree*"

primrec *data* **where**
"*data* $\Lambda = (\lambda x.\ _)$" |
"*data* (*N* x _ _ _) = x"

primrec *dataset* **where**
"*dataset* $\Lambda = \{\}$" |
"*dataset* (*N* x _ t u) = $\{x\} \cup$ *dataset* $t \cup$ *dataset* u"

primrec *level* **where**
"*level* $\Lambda = 0$" |
"*level* (*N* _ k _ _) = k"

primrec *left* **where**
"*left* $\Lambda = \Lambda$" |
"*left* (*N* _ _ t _) = t"

primrec *right* **where**
"*right* $\Lambda = \Lambda$" |
"*right* $(N _ _ _ u) = u$"

The wellformedness criterion for AA trees is fairly complex. Wikipedia states it as follows [9]:

> Each node has a level field, and the following invariants must remain true for the tree to be valid:
>
> 1. The level of a leaf node is one.
> 2. The level of a left child is strictly less than that of its parent.
> 3. The level of a right child is less than or equal to that of its parent.
> 4. The level of a right grandchild is strictly less than that of its grandparent.
> 5. Every node of level greater than one must have two children.

The *wf* predicate formalizes this description:

primrec *wf* **where**
"*wf* $\Lambda =$ *True*" |
"*wf* $(N _ k\ t\ u) =$
(if $t = \Lambda$ then
$k = 1 \wedge (u = \Lambda \vee (level\ u = 1 \wedge left\ u = \Lambda \wedge right\ u = \Lambda))$
else
$wf\ t \wedge wf\ u \wedge u \neq \Lambda \wedge level\ t < k \wedge level\ u \leq k$
$\wedge\ level\ (right\ u) < k)$"

Rebalancing the tree upon insertion and removal of elements is performed by two auxiliary functions called *skew* and *split*, defined below:

primrec *skew* **where**
"*skew* $\Lambda = \Lambda$" |
"*skew* $(N\ x\ k\ t\ u) =$
(if $t \neq \Lambda \wedge k = level\ t$ then
$N\ (data\ t)\ k\ (left\ t)\ (N\ x\ k\ (right\ t)\ u)$
else
$N\ x\ k\ t\ u$)"

primrec *split* **where**
"*split* $\Lambda = \Lambda$" |

"*split* (*N x k t u*) =
 (if $u \neq \Lambda \wedge k = level\ (right\ u)$ then
    N (*data u*) (*Suc k*) (*N x k t* (*left u*)) (*right u*)
  else
    N x k t u)"

Performing a *skew* or a *split* should have no impact on the set of elements stored in the tree:

theorem *dataset_skew_split*:
"*dataset* (*skew t*) = *dataset t*"
"*dataset* (*split t*) = *dataset t*"
nitpick

Nitpick ran out of time after checking 9 of 10 scopes

Furthermore, applying *skew* or *split* on a well-formed tree should not alter the tree:

theorem *wf_skew_split*:
"*wf t* $\Longrightarrow$ *skew t* = *t*"
"*wf t* $\Longrightarrow$ *split t* = *t*"
nitpick

Nitpick found no counterexample

Insertion is implemented recursively. It preserves the sort order:

primrec *insort* **where**
"*insort* Λ *x* = *N x* 1 Λ Λ" |
"*insort* (*N y k t u*) *x* =
 (∗ (*split* ∘ *skew*) ∗) (*N y k* (if *x* < *y* then *insort t x* else *t*)
                              (if *x* > *y* then *insort u x* else *u*))"

Notice that we deliberately commented out the application of *skew* and *split*. Let's see if this causes any problems:

theorem *wf_insort*: "*wf t* $\Longrightarrow$ *wf* (*insort t x*)"
nitpick

Nitpick found a counterexample for card 'a = 4*:*

Free variables:
$t = N\ a_1\ 1\ \Lambda\ \Lambda$
$x = a_2$

It's hard to see why this is a counterexample. To improve readability, we will restrict the theorem to *nat*, so that we don't need to look up the value of the *op* $<$ constant to find out which element is smaller than the other. In addition, we will tell Nitpick to display the value of *insort t x* using the *eval* option. This gives

> **theorem** *wf_insort_nat*: "*wf t* $\Longrightarrow$ *wf* (*insort t* (*x*::*nat*))"
> **nitpick** [*eval* = "*insort t x*"]
>
> *Nitpick found a counterexample:*
>
> *Free variables:*
> $t = N\ 1\ 1\ \Lambda\ \Lambda$
> $x = 0$
> *Evaluated term:*
> *insort t x* $= N\ 1\ 1\ (N\ 0\ 1\ \Lambda\ \Lambda)\ \Lambda$

Nitpick's output reveals that the element 0 was added as a left child of 1, where both nodes have a level of 1. This violates the second AA tree invariant, which states that a left child's level must be less than its parent's. This shouldn't come as a surprise, considering that we commented out the tree rebalancing code. Reintroducing the code seems to solve the problem:

> **theorem** *wf_insort*: "*wf t* $\Longrightarrow$ *wf* (*insort t x*)"
> **nitpick**
>
> *Nitpick ran out of time after checking 8 of 10 scopes*

Insertion should transform the set of elements represented by the tree in the obvious way:

> **theorem** *dataset_insort*: "*dataset* (*insort t x*) = $\{x\} \cup$ *dataset t*"
> **nitpick**
>
> *Nitpick ran out of time after checking 7 of 10 scopes*

We could continue like this and sketch a full-blown theory of AA trees. Once the definitions and main theorems are in place and have been thoroughly tested using Nitpick, we could start working on the proofs. Developing theories this way usually saves time, because faulty theorems and definitions are discovered much earlier in the process.

5 Option Reference

Nitpick's behavior can be influenced by various options, which can be specified in brackets after the **nitpick** command. Default values can be set using **nitpick_params**. For example:

> **nitpick_params** [*verbose*, *timeout* = 60]

The options are categorized as follows: mode of operation (§5.1), scope of search (§5.2), output format (§5.3), regression testing (§5.4), optimizations (§5.5), and timeouts (§5.6).

Nitpick also provides an automatic mode that can be enabled via the "Auto Nitpick" option under "Plugins > Plugin Options > Isabelle > General" in Isabelle/jEdit. For automatic runs, *user_axioms* (§5.1), *assms* (§5.1), and *mono* (§5.2) are implicitly enabled, *verbose* (§5.3) and *debug* (§5.3) are disabled, *max_potential* (§5.3) is taken to be 0, *max_threads* (§5.5) is taken to be 1, and *timeout* (§5.6) is superseded by the "Auto Time Limit" in Isabelle/jEdit. Nitpick's output is also more concise.

The number of options can be overwhelming at first glance. Do not let that worry you: Nitpick's defaults have been chosen so that it almost always does the right thing, and the most important options have been covered in context in §3.

The descriptions below refer to the following syntactic quantities:

- ⟨***string***⟩: A string.
- ⟨***string_list***⟩: A space-separated list of strings (e.g., "*ichi ni san*").
- ⟨***bool***⟩: *true* or *false*.
- ⟨***smart_bool***⟩: *true*, *false*, or *smart*.
- ⟨***int***⟩: An integer. Negative integers are prefixed with a hyphen.
- ⟨***smart_int***⟩: An integer or *smart*.
- ⟨***int_range***⟩: An integer (e.g., 3) or a range of nonnegative integers (e.g., 1–4). The range symbol '–' is entered as - (hyphen).
- ⟨***int_seq***⟩: A comma-separated sequence of ranges of integers (e.g., 1,3, 6–8).
- ⟨***float***⟩: An floating-point number (e.g., 0.5 or 60) expressing a number of seconds.

- ⟨***const***⟩: The name of a HOL constant.
- ⟨***term***⟩: A HOL term (e.g., "$f\ x$").
- ⟨***term_list***⟩: A space-separated list of HOL terms (e.g., "$f\ x$" "$g\ y$").
- ⟨***type***⟩: A HOL type.

Default values are indicated in curly brackets ({}). Boolean options have a negated counterpart (e.g., *mono* vs. *non_mono*). When setting them, "= *true*" may be omitted.

5.1 Mode of Operation

falsify [= ⟨***bool***⟩] **{*true*}** **(neg.: *satisfy*)**

Specifies whether Nitpick should look for falsifying examples (countermodels) or satisfying examples (models). This manual assumes throughout that *falsify* is enabled.

user_axioms [= ⟨***smart_bool***⟩] **{*smart*}(neg.: *no_user_axioms*)**

Specifies whether the user-defined axioms (specified using **axiomatization** and **axioms**) should be considered. If the option is set to *smart*, Nitpick performs an ad hoc axiom selection based on the constants that occur in the formula to falsify. The option is implicitly set to *true* for automatic runs.

Warning: If the option is set to *true*, Nitpick might nonetheless ignore some polymorphic axioms. Counterexamples generated under these conditions are tagged as "quasi genuine." The *debug* (§5.3) option can be used to find out which axioms were considered.

See also *assms* (§5.1) and *debug* (§5.3).

assms [= ⟨***bool***⟩] **{*true*}** **(neg.: *no_assms*)**

Specifies whether the relevant assumptions in structured proofs should be considered. The option is implicitly enabled for automatic runs.

See also *user_axioms* (§5.1).

spy [= ⟨***bool***⟩] **{*false*}** **(neg.: *dont_spy*)**

Specifies whether Nitpick should record statistics in `$ISABELLE_HOME_USER/spy_nitpick`. These statistics can be useful to the developer of Nitpick. If you are willing to have your interactions recorded in the

name of science, please enable this feature and send the statistics file every now and then to the author of this manual (`jasmin.blanchette@inria.fr`). To change the default value of this option globally, set the environment variable `NITPICK_SPY` to `yes`.

See also *debug* (§5.3).

overlord $[= \langle \textit{bool} \rangle]$ **{*false*}** **(neg.: *no_overlord*)**

Specifies whether Nitpick should put its temporary files in `$ISABELLE_HOME_USER`, which is useful for debugging Nitpick but also unsafe if several instances of the tool are run simultaneously. The files are identified by the extensions `.kki`, `.cnf`, `.out`, and `.err`; you may safely remove them after Nitpick has run.

Warning: This option is not thread-safe. Use at your own risks.

See also *debug* (§5.3).

5.2 Scope of Search

card $\langle \textit{type} \rangle = \langle \textit{int_seq} \rangle$

Specifies the sequence of cardinalities to use for a given type. For free types, and often also for **typedecl**'d types, it usually makes sense to specify cardinalities as a range of the form 1–n.

See also *box* (§5.2) and *mono* (§5.2).

card $= \langle \textit{int_seq} \rangle$ **{1–10}**

Specifies the default sequence of cardinalities to use. This can be overridden on a per-type basis using the *card* $\langle$*type*$\rangle$ option described above.

max $\langle \textit{const} \rangle = \langle \textit{int_seq} \rangle$

Specifies the sequence of maximum multiplicities to use for a given (co)inductive datatype constructor. A constructor's multiplicity is the number of distinct values that it can construct. Nonsensical values (e.g., *max* $[] = 2$) are silently repaired. This option is only available for datatypes equipped with several constructors.

max $= \langle \textit{int_seq} \rangle$

Specifies the default sequence of maximum multiplicities to use for (co)inductive datatype constructors. This can be overridden on a per-constructor basis using the *max* $\langle$*const*$\rangle$ option described above.

binary_ints $[= \langle$***smart_bool***$\rangle]$ **{*smart*}** **(neg.: *unary_ints*)**

Specifies whether natural numbers and integers should be encoded using a unary or binary notation. In unary mode, the cardinality fully specifies the subset used to approximate the type. For example:

$$\begin{array}{rll} \textit{card nat} = 4 & \text{induces} & \{0, 1, 2, 3\} \\ \textit{card int} = 4 & \text{induces} & \{-1, 0, +1, +2\} \\ \textit{card int} = 5 & \text{induces} & \{-2, -1, 0, +1, +2\}. \end{array}$$

In general:

$$\begin{array}{rll} \textit{card nat} = K & \text{induces} & \{0, \ldots, K-1\} \\ \textit{card int} = K & \text{induces} & \{-\lceil K/2\rceil + 1, \ldots, +\lfloor K/2\rfloor\}. \end{array}$$

In binary mode, the cardinality specifies the number of distinct values that can be constructed. Each of these value is represented by a bit pattern whose length is specified by the *bits* (§5.2) option. By default, Nitpick attempts to choose the more appropriate encoding by inspecting the formula at hand, preferring the binary notation for problems involving multiplicative operators or large constants.

Warning: For technical reasons, Nitpick always reverts to unary for problems that refer to the types *rat* or *real* or the constants *Suc*, *gcd*, or *lcm*.

See also *bits* (§5.2) and *show_types* (§5.3).

bits $= \langle$***int_seq***$\rangle$ **{1–10}**

Specifies the number of bits to use to represent natural numbers and integers in binary, excluding the sign bit. The minimum is 1 and the maximum is 31.

See also *binary_ints* (§5.2).

wf $\langle$***const***$\rangle$ $[= \langle$***smart_bool***$\rangle]$ **(neg.: *non_wf*)**

Specifies whether the specified (co)inductively defined predicate is well-founded. The option can take the following values:

- ***true*:** Tentatively treat the (co)inductive predicate as if it were well-founded. Since this is generally not sound when the predicate is not well-founded, the counterexamples are tagged as "quasi genuine."
- ***false*:** Treat the (co)inductive predicate as if it were not well-founded. The predicate is then unrolled as prescribed by the *star_linear_preds*, *iter* $\langle const\rangle$, and *iter* options.

- ***smart*:** Try to prove that the inductive predicate is well-founded using Isabelle's *lexicographic_order* and *size_change* tactics. If this succeeds (or the predicate occurs with an appropriate polarity in the formula to falsify), use an efficient fixed-point equation as specification of the predicate; otherwise, unroll the predicates according to the *iter* ⟨*const*⟩ and *iter* options.

See also *iter* (§5.2), *star_linear_preds* (§5.5), and *tac_timeout* (§5.6).

wf [= ⟨***smart*_*bool***⟩] **{*smart*}** **(neg.: *non*_*wf*)**

Specifies the default wellfoundedness setting to use. This can be overridden on a per-predicate basis using the *wf* ⟨*const*⟩ option above.

iter ⟨***const***⟩ = ⟨***int*_*seq***⟩

Specifies the sequence of iteration counts to use when unrolling a given (co)inductive predicate. By default, unrolling is applied for inductive predicates that occur negatively and coinductive predicates that occur positively in the formula to falsify and that cannot be proved to be well-founded, but this behavior is influenced by the *wf* option. The iteration counts are automatically bounded by the cardinality of the predicate's domain.

See also *wf* (§5.2) and *star_linear_preds* (§5.5).

iter = ⟨***int*_*seq***⟩ **{0,1,2,4,8,12,16,20,24,28}**

Specifies the sequence of iteration counts to use when unrolling (co)inductive predicates. This can be overridden on a per-predicate basis using the *iter* ⟨*const*⟩ option above.

bisim*_*depth = ⟨***int*_*seq***⟩ **{9}**

Specifies the sequence of iteration counts to use when unrolling the bisimilarity predicate generated by Nitpick for coinductive datatypes. A value of −1 means that no predicate is generated, in which case Nitpick performs an after-the-fact check to see if the known coinductive datatype values are bidissimilar. If two values are found to be bisimilar, the counterexample is tagged as "quasi genuine." The iteration counts are automatically bounded by the sum of the cardinalities of the coinductive datatypes occurring in the formula to falsify.

box ⟨***type***⟩ [= ⟨***smart*_*bool***⟩] **(neg.: *dont*_*box*)**

Specifies whether Nitpick should attempt to wrap ("box") a given function or product type in an isomorphic datatype internally. Boxing is

an effective mean to reduce the search space and speed up Nitpick, because the isomorphic datatype is approximated by a subset of the possible function or pair values. Like other drastic optimizations, it can also prevent the discovery of counterexamples. The option can take the following values:

- ***true*:** Box the specified type whenever practicable.
- ***false*:** Never box the type.
- ***smart*:** Box the type only in contexts where it is likely to help. For example, n-tuples where $n > 2$ and arguments to higher-order functions are good candidates for boxing.

See also *finitize* (§5.2), *verbose* (§5.3), and *debug* (§5.3).

box $[= \langle$***smart_bool***$\rangle]$ **{*smart*}** **(neg.: *dont_box*)**

Specifies the default boxing setting to use. This can be overridden on a per-type basis using the *box* $\langle$*type*$\rangle$ option described above.

finitize $\langle$***type***$\rangle$ $[= \langle$***smart_bool***$\rangle]$ **(neg.: *dont_finitize*)**

Specifies whether Nitpick should attempt to finitize an infinite datatype. The option can then take the following values:

- ***true*:** Finitize the datatype. Since this is unsound, counterexamples generated under these conditions are tagged as "quasi genuine."
- ***false*:** Don't attempt to finitize the datatype.
- ***smart*:** If the datatype's constructors don't appear in the problem, perform a monotonicity analysis to detect whether the datatype can be soundly finitized; otherwise, don't finitize it.

See also *box* (§5.2), *mono* (§5.2), *verbose* (§5.3), and *debug* (§5.3).

finitize $[= \langle$***smart_bool***$\rangle]$ **{*smart*}** **(neg.: *dont_finitize*)**

Specifies the default finitization setting to use. This can be overridden on a per-type basis using the *finitize* $\langle$*type*$\rangle$ option described above.

mono $\langle$***type***$\rangle$ $[= \langle$***smart_bool***$\rangle]$ **(neg.: *non_mono*)**

Specifies whether the given type should be considered monotonic when enumerating scopes and finitizing types. If the option is set to *smart*, Nitpick performs a monotonicity check on the type. Setting this option to *true* can reduce the number of scopes tried, but it can also diminish

the chance of finding a counterexample, as demonstrated in §3.11. The option is implicitly set to *true* for automatic runs.

See also *card* (§5.2), *finitize* (§5.2), *merge_type_vars* (§5.2), and *verbose* (§5.3).

mono [= ⟨***smart_bool***⟩] {***smart***} **(neg.: *non_mono*)**

Specifies the default monotonicity setting to use. This can be overridden on a per-type basis using the *mono* ⟨*type*⟩ option described above.

merge_type_vars [= ⟨***bool***⟩] {***false***} **(neg.: *dont_merge_type_vars*)**

Specifies whether type variables with the same sort constraints should be merged. Setting this option to *true* can reduce the number of scopes tried and the size of the generated Kodkod formulas, but it also diminishes the theoretical chance of finding a counterexample.

See also *mono* (§5.2).

5.3 Output Format

verbose [= ⟨***bool***⟩] {***false***} **(neg.: *quiet*)**

Specifies whether the **nitpick** command should explain what it does. This option is useful to determine which scopes are tried or which SAT solver is used. This option is implicitly disabled for automatic runs.

debug [= ⟨***bool***⟩] {***false***} **(neg.: *no_debug*)**

Specifies whether Nitpick should display additional debugging information beyond what *verbose* already displays. Enabling *debug* also enables *verbose* and *show_all* behind the scenes. The *debug* option is implicitly disabled for automatic runs.

See also *spy* (§5.1), *overlord* (§5.1), and *batch_size* (§5.5).

show_types [= ⟨***bool***⟩] {***false***} **(neg.: *hide_types*)**

Specifies whether the subsets used to approximate (co)inductive datatypes should be displayed as part of counterexamples. Such subsets are sometimes helpful when investigating whether a potentially spurious counterexample is genuine, but their potential for clutter is real.

show_skolems [= ⟨***bool***⟩] {***true***} **(neg.: *hide_skolem*)**

Specifies whether the values of Skolem constants should be displayed

as part of counterexamples. Skolem constants correspond to bound variables in the original formula and usually help us to understand why the counterexample falsifies the formula.

show_consts $[= \langle$***bool***$\rangle]$ **{*false*}** **(neg.: *hide_consts*)**

Specifies whether the values of constants occurring in the formula (including its axioms) should be displayed along with any counterexample. These values are sometimes helpful when investigating why a counterexample is genuine, but they can clutter the output.

show_all $= \langle$***bool***$\rangle$

Abbreviation for *show_types*, *show_skolems*, and *show_consts*.

max_potential $= \langle$***int***$\rangle$ **{1}**

Specifies the maximum number of potentially spurious counterexamples to display. Setting this option to 0 speeds up the search for a genuine counterexample. This option is implicitly set to 0 for automatic runs. If you set this option to a value greater than 1, you will need an incremental SAT solver, such as *MiniSat_JNI* (recommended) and *SAT4J*. Be aware that many of the counterexamples may be identical.

See also *sat_solver* (§5.5).

max_genuine $= \langle$***int***$\rangle$ **{1}**

Specifies the maximum number of genuine counterexamples to display. If you set this option to a value greater than 1, you will need an incremental SAT solver, such as *MiniSat_JNI* (recommended) and *SAT4J*. Be aware that many of the counterexamples may be identical.

See also *sat_solver* (§5.5).

eval $= \langle$***term_list***$\rangle$

Specifies the list of terms whose values should be displayed along with counterexamples. This option suffers from an "observer effect": Nitpick might find different counterexamples for different values of this option.

atoms $\langle$***type***$\rangle = \langle$***string_list***$\rangle$

Specifies the names to use to refer to the atoms of the given type. By default, Nitpick generates names of the form $a_1, \ldots, a_n$, where a is the first letter of the type's name.

atoms $= \langle$***string_list***$\rangle$

Specifies the default names to use to refer to atoms of any type. For

example, to call the three atoms of type $'a$ *ichi*, *ni*, and *san* instead of a_1, a_2, a_3, specify the option "*atoms* $'a$ = *ichi ni san*". The default names can be overridden on a per-type basis using the *atoms* ⟨*type*⟩ option described above.

format ⟨***term***⟩ = ⟨***int_seq***⟩

Specifies how to uncurry the value displayed for a variable or constant. Uncurrying sometimes increases the readability of the output for high-arity functions. For example, given the variable $y::'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow 'g$, setting *format* $y = 3$ tells Nitpick to group the last three arguments, as if the type had been $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \times 'e \times 'f \Rightarrow 'g$. In general, a list of values $n_1, \ldots, n_k$ tells Nitpick to show the last n_k arguments as an n_k-tuple, the previous n_{k-1} arguments as an n_{k-1}-tuple, and so on; arguments that are not accounted for are left alone, as if the specification had been $1, \ldots, 1, n_1, \ldots, n_k$.

format = ⟨***int_seq***⟩ **{1}**

Specifies the default format to use. Irrespective of the default format, the extra arguments to a Skolem constant corresponding to the outer bound variables are kept separated from the remaining arguments, the **for** arguments of an inductive definitions are kept separated from the remaining arguments, and the iteration counter of an unrolled inductive definition is shown alone. The default format can be overridden on a per-variable or per-constant basis using the *format* ⟨*term*⟩ option described above.

5.4 Regression Testing

expect = ⟨***string***⟩

Specifies the expected outcome, which must be one of the following:

- ***genuine*****:** Nitpick found a genuine counterexample.
- ***quasi_genuine*****:** Nitpick found a "quasi genuine" counterexample (i.e., a counterexample that is genuine unless it contradicts a missing axiom or a dangerous option was used inappropriately).
- ***potential*****:** Nitpick found a potentially spurious counterexample.
- ***none*****:** Nitpick found no counterexample.
- ***unknown*****:** Nitpick encountered some problem (e.g., Kodkod ran out of memory).

Nitpick emits an error if the actual outcome differs from the expected outcome. This option is useful for regression testing.

5.5 Optimizations

sat_solver = ⟨***string***⟩ **{*smart*}**

Specifies which SAT solver to use. SAT solvers implemented in C or C++ tend to be faster than their Java counterparts, but they can be more difficult to install. Also, if you set the *max_potential* (§5.3) or *max_genuine* (§5.3) option to a value greater than 1, you will need an incremental SAT solver, such as *MiniSat_JNI* (recommended) or *SAT4J*.

The supported solvers are listed below:

- ***Lingeling_JNI*:** Lingeling is an efficient solver written in C. The JNI (Java Native Interface) version of Lingeling is bundled with Kodkodi and is precompiled for Linux and Mac OS X. It is also available from the Kodkod web site [6].
- ***CryptoMiniSat*:** CryptoMiniSat is the winner of the 2010 SAT Race. To use CryptoMiniSat, set the environment variable `CRYPTOMINISAT_HOME` to the directory that contains the `cryptominisat` executable.The C++ sources and executables for CryptoMiniSat are available at `http://planete.inrialpes.fr/~soos/CryptoMiniSat2/index.php`. Nitpick has been tested with version 2.51.
- ***CryptoMiniSat_JNI*:** The JNI (Java Native Interface) version of CryptoMiniSat is bundled with Kodkodi and is precompiled for Linux and Mac OS X. It is also available from the Kodkod web site [6].
- ***MiniSat*:** MiniSat is an efficient solver written in C++. To use MiniSat, set the environment variable `MINISAT_HOME` to the directory that contains the `minisat` executable.The C++ sources and executables for MiniSat are available at `http://minisat.se/MiniSat.html`. Nitpick has been tested with versions 1.14 and 2.2.
- ***MiniSat_JNI*:** The JNI version of MiniSat is bundled with Kodkodi and is precompiled for Linux, Mac OS X, and Windows (Cygwin). It is also available from the Kodkod web site [6]. Unlike the standard version of MiniSat, the JNI version can be used incrementally.

- ***Riss3g*:** Riss3g is an efficient solver written in C++. To use Riss3g, set the environment variable `RISS3G_HOME` to the directory that contains the `riss3g` executable.The C++ sources for Riss3g are available at `http://tools.computational-logic.org/content/riss3g.php`. Nitpick has been tested with the SAT Competition 2013 version.
- ***zChaff*:** zChaff is an older solver written in C++. To use zChaff, set the environment variable `ZCHAFF_HOME` to the directory that contains the `zchaff` executable.The C++ sources and executables for zChaff are available at `http://www.princeton.edu/~chaff/zchaff.html`. Nitpick has been tested with versions 2004-05-13, 2004-11-15, and 2007-03-12.
- ***RSat*:** RSat is an efficient solver written in C++. To use RSat, set the environment variable `RSAT_HOME` to the directory that contains the `rsat` executable.The C++ sources for RSat are available at `http://reasoning.cs.ucla.edu/rsat/`. Nitpick has been tested with version 2.01.
- ***BerkMin*:** BerkMin561 is an efficient solver written in C. To use BerkMin, set the environment variable `BERKMIN_HOME` to the directory that contains the `BerkMin561` executable. The BerkMin executables are available at `http://eigold.tripod.com/BerkMin.html`.
- ***BerkMin_Alloy*:** Variant of BerkMin that is included with Alloy 4 and calls itself "sat56" in its banner text. To use this version of BerkMin, set the environment variable `BERKMINALLOY_HOME` to the directory that contains the `berkmin` executable.
- ***SAT4J*:** SAT4J is a reasonably efficient solver written in Java that can be used incrementally. It is bundled with Kodkodi and requires no further installation or configuration steps. Do not attempt to install the official SAT4J packages, because their API is incompatible with Kodkod.
- ***SAT4J_Light*:** Variant of SAT4J that is optimized for small problems. It can also be used incrementally.
- ***smart*:** If *sat_solver* is set to *smart*, Nitpick selects the first solver among the above that is recognized by Isabelle. If *verbose* (§5.3) is enabled, Nitpick displays which SAT solver was chosen.

***batch_size* = ⟨*smart_int*⟩ **[*smart*]**

Specifies the maximum number of Kodkod problems that should be

lumped together when invoking Kodkodi. Each problem corresponds to one scope. Lumping problems together ensures that Kodkodi is launched less often, but it makes the verbose output less readable and is sometimes detrimental to performance. If *batch_size* is set to *smart*, the actual value used is 1 if *debug* (§5.3) is set and 50 otherwise.

destroy_constrs [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_destroy_constrs*)**

Specifies whether formulas involving (co)inductive datatype constructors should be rewritten to use (automatically generated) discriminators and destructors. This optimization can drastically reduce the size of the Boolean formulas given to the SAT solver.

See also *debug* (§5.3).

specialize [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_specialize*)**

Specifies whether functions invoked with static arguments should be specialized. This optimization can drastically reduce the search space, especially for higher-order functions.

See also *debug* (§5.3) and *show_consts* (§5.3).

star_linear_preds [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_star_linear_preds*)**

Specifies whether Nitpick should use Kodkod's transitive closure operator to encode non-well-founded "linear inductive predicates," i.e., inductive predicates for which each the predicate occurs in at most one assumption of each introduction rule. Using the reflexive transitive closure is in principle equivalent to setting *iter* to the cardinality of the predicate's domain, but it is usually more efficient.

See also *wf* (§5.2), *debug* (§5.3), and *iter* (§5.2).

whack = ⟨***term_list***⟩

Specifies a list of atomic terms (usually constants, but also free and schematic variables) that should be taken as being _ (unknown). This can be useful to reduce the size of the Kodkod problem if you can guess in advance that a constant might not be needed to find a countermodel.

See also *debug* (§5.3).

need = ⟨***term_list***⟩

Specifies a list of datatype values (normally ground constructor terms) that should be part of the subterm-closed subsets used to approximate datatypes. If you know that a value must necessarily belong to the

subset of representable values that approximates a datatype, specifying it can speed up the search, especially for high cardinalities.

total_consts $[= \langle$***smart_bool***$\rangle]$ **{*smart*}** **(neg.: *partial_consts*)**

Specifies whether constants occurring in the problem other than constructors can be assumed to be considered total for the representable values that approximate a datatype. This option is highly incomplete; it should be used only for problems that do not construct datatype values explicitly. Since this option is (in rare cases) unsound, counterexamples generated under these conditions are tagged as "quasi genuine."

datatype_sym_break $= \langle$***int***$\rangle$ **{5}**

Specifies an upper bound on the number of datatypes for which Nitpick generates symmetry breaking predicates. Symmetry breaking can speed up the SAT solver considerably, especially for unsatisfiable problems, but too much of it can slow it down.

kodkod_sym_break $= \langle$***int***$\rangle$ **{15}**

Specifies an upper bound on the number of relations for which Kodkod generates symmetry breaking predicates. Symmetry breaking can speed up the SAT solver considerably, especially for unsatisfiable problems, but too much of it can slow it down.

peephole_optim $[= \langle$***bool***$\rangle]$ **{*true*}** **(neg.: *no_peephole_optim*)**

Specifies whether Nitpick should simplify the generated Kodkod formulas using a peephole optimizer. These optimizations can make a significant difference. Unless you are tracking down a bug in Nitpick or distrust the peephole optimizer, you should leave this option enabled.

max_threads $= \langle$***int***$\rangle$ **{0}**

Specifies the maximum number of threads to use in Kodkod. If this option is set to 0, Kodkod will compute an appropriate value based on the number of processor cores available. The option is implicitly set to 1 for automatic runs.

See also *batch_size* (§5.5) and *timeout* (§5.6).

5.6 Timeouts

timeout $= \langle$***float***$\rangle$ **{30}**

Specifies the maximum number of seconds that the **nitpick** command should spend looking for a counterexample. Nitpick tries to honor this constraint as well as it can but offers no guarantees. For automatic runs, the "Auto Time Limit" option under "Plugins > Plugin Options > Isabelle > General" is used instead.

See also *max_threads* (§5.5).

tac_timeout $= \langle$***float***$\rangle$ **{0.5}**

Specifies the maximum number of seconds that should be used by internal tactics—*lexicographic_order* and *size_change* when checking whether a (co)inductive predicate is well-founded or the monotonicity inference. Nitpick tries to honor this constraint but offers no guarantees.

See also *wf* (§5.2) and *mono* (§5.2).

6 Attribute Reference

Nitpick needs to consider the definitions of all constants occurring in a formula in order to falsify it. For constants introduced using the **definition** command, the definition is simply the associated *_def* axiom. In contrast, instead of using the internal representation of functions synthesized by Isabelle's **primrec**, **function**, and **nominal_primrec** packages, Nitpick relies on the more natural equational specification entered by the user.

Behind the scenes, Isabelle's built-in packages and theories rely on the following attributes to affect Nitpick's behavior:

nitpick_unfold

This attribute specifies an equation that Nitpick should use to expand a constant. The equation should be logically equivalent to the constant's actual definition and should be of the form

$$c\ ?x_1\ \ldots\ ?x_n\ =\ t,$$

or

$$c\ ?x_1\ \ldots\ ?x_n\ \equiv\ t,$$

where $?x_1, \ldots, ?x_n$ are distinct variables and c does not occur in t. Each occurrence of c in the problem is expanded to $\lambda x_1 \ldots x_n.\ t$.

nitpick_simp

This attribute specifies the equations that constitute the specification of a constant. The **primrec**, **function**, and **nominal_primrec** packages automatically attach this attribute to their *simps* rules. The equations must be of the form

$$c\ t_1\ \ldots\ t_n\ \left[= u\right]$$

or

$$c\ t_1\ \ldots\ t_n \equiv u.$$

nitpick_psimp

This attribute specifies the equations that constitute the partial specification of a constant. The **function** package automatically attaches this attribute to its *psimps* rules. The conditional equations must be of the form

$$[\![P_1;\ \ldots;\ P_m]\!] \Longrightarrow c\ t_1\ \ldots\ t_n\ \left[= u\right]$$

or

$$[\![P_1;\ \ldots;\ P_m]\!] \Longrightarrow c\ t_1\ \ldots\ t_n \equiv u.$$

nitpick_choice_spec

This attribute specifies the (free-form) specification of a constant defined using the **specification** command.

When faced with a constant, Nitpick proceeds as follows:

1. If the *nitpick_simp* set associated with the constant is not empty, Nitpick uses these rules as the specification of the constant.
2. Otherwise, if the *nitpick_psimp* set associated with the constant is not empty, it uses these rules as the specification of the constant.
3. Otherwise, if the constant was defined using the **specification** command and the *nitpick_choice_spec* set associated with the constant is not empty, it uses these theorems as the specification of the constant.
4. Otherwise, it looks up the definition of the constant. If the *nitpick_unfold* set associated with the constant is not empty, it uses the latest rule added to the set as the definition of the constant; otherwise it uses the actual definition axiom.

1. If the definition is of the form

   $$c\ ?x_1\ \ldots\ ?x_m \equiv \lambda y_1\ \ldots\ y_n.\ \mathit{lfp}\ (\lambda f.\ t)$$

   or

   $$c\ ?x_1\ \ldots\ ?x_m \equiv \lambda y_1\ \ldots\ y_n.\ \mathit{gfp}\ (\lambda f.\ t).$$

   Nitpick assumes that the definition was made using a (co)inductive package based on the user-specified introduction rules registered in Isabelle's internal *Spec_Rules* table. The tool uses the introduction rules to ascertain whether the definition is well-founded and the definition to generate a fixed-point equation or an unrolled equation.
2. If the definition is compact enough, the constant is *unfolded* wherever it appears; otherwise, it is defined equationally, as with the *nitpick_simp* attribute.

As an illustration, consider the inductive definition

inductive *odd* **where**
"*odd* 1" |
"*odd* $n \Longrightarrow$ *odd* (*Suc* (*Suc* n))"

By default, Nitpick uses the *lfp*-based definition in conjunction with the introduction rules. To override this, you can specify an alternative definition as follows:

lemma *odd_alt_unfold* [*nitpick_unfold*]: "*odd* $n \equiv n \bmod 2 = 1$"

Nitpick then expands all occurrences of *odd* n to $n \bmod 2 = 1$. Alternatively, you can specify an equational specification of the constant:

lemma *odd_simp* [*nitpick_simp*]: "*odd* $n = (n \bmod 2 = 1)$"

Such tweaks should be done with great care, because Nitpick will assume that the constant is completely defined by its equational specification. For example, if you make "*odd* $(2 * k + 1)$" a *nitpick_simp* rule and neglect to provide rules to handle the $2 * k$ case, Nitpick will define *odd* n arbitrarily for even values of n. The *debug* (§5.3) option is extremely useful to understand what is going on when experimenting with *nitpick_* attributes.

Because of its internal three-valued logic, Nitpick tends to lose a lot of precision in the presence of partially specified constants. For example,

lemma *odd_simp* [*nitpick_simp*]: "*odd* $x = \neg$ *even* x"

is superior to

lemma *odd_psimps* [*nitpick_simp*]:
"*even x* $\Longrightarrow$ *odd x* = *False*"
"$\neg$ *even x* $\Longrightarrow$ *odd x* = *True*"

Because Nitpick sometimes unfolds definitions but never simplification rules, you can ensure that a constant is defined explicitly using the *nitpick_simp*. For example:

definition *optimum* **where** [*nitpick_simp*]:
"*optimum t* = (∀*u*. *consistent u* ∧ *alphabet t* = *alphabet u*
∧ *freq t* = *freq u* ⟶ *cost t* ≤ *cost u*)"

In some rare occasions, you might want to provide an inductive or coinductive view on top of an existing constant c. The easiest way to achieve this is to define a new constant c' (co)inductively. Then prove that c equals c' and let Nitpick know about it:

lemma *c_alt_unfold* [*nitpick_unfold*]: "$c \equiv c'$"

This ensures that Nitpick will substitute c' for c and use the (co)inductive definition.

7 Standard ML Interface

Nitpick provides a rich Standard ML interface used mainly for internal purposes and debugging. Among the most interesting functions exported by Nitpick are those that let you invoke the tool programmatically and those that let you register and unregister custom term postprocessors as well as coinductive datatypes.

7.1 Invoking Nitpick

The *Nitpick* structure offers the following functions for invoking your favorite counterexample generator:

val *pick_nits_in_term* :
Proof.state → *params* → *mode* → *int* → *int* → *int*
→ (*term* ∗ *term*) *list* → *term list* → *term* → *string* ∗ *Proof.state*
val *pick_nits_in_subgoal* :
Proof.state → *params* → *mode* → *int* → *int* → *string* ∗ *Proof.state*

The return value is a new proof state paired with an outcome string ("genuine", "quasi_genuine", "potential", "none", or "unknown"). The *params* type is a large record that lets you set Nitpick's options. The current default options can be retrieved by calling the following function defined in the *Nitpick_Isar* structure:

val *default_params* : *theory* → (*string* ∗ *string*) *list* → *params*

The second argument lets you override option values before they are parsed and put into a *params* record. Here is an example where Nitpick is invoked on subgoal *i* of *n* with no time limit:

val *params* = *Nitpick_Isar.default_params thy* [("timeout", "none")]
val (*outcome*, *state*′) =
  Nitpick.pick_nits_in_subgoal state params Nitpick.Normal i n

7.2 Registering Term Postprocessors

It is possible to change the output of any term that Nitpick considers a datatype by registering a term postprocessor. The interface for registering and unregistering postprocessors consists of the following pair of functions defined in the *Nitpick_Model* structure:

type *term_postprocessor* =
  Proof.context → *string* → (*typ* → *term list*) → *typ* → *term* → *term*
val *register_term_postprocessor* :
  typ → *term_postprocessor* → *morphism* → *Context.generic*
  → *Context.generic*
val *unregister_term_postprocessor* :
  typ → *morphism* → *Context.generic* → *Context.generic*

§3.7 and `src/HOL/Library/Multiset.thy` illustrate this feature in context.

7.3 Registering Coinductive Datatypes

Coinductive datatypes defined using the **codatatype** command that do not involve nested recursion through non-codatatypes are supported by Nitpick. If you have defined a custom coinductive datatype, you can tell Nitpick about it, so that it can use an efficient Kodkod axiomatization. The interface for registering and unregistering coinductive datatypes consists of the following pair of functions defined in the *Nitpick_HOL* structure:

val *register_codatatype* :
morphism → *typ* → *string* → (*string* × *typ*) *list* → *Context.generic*
→ *Context.generic*
val *unregister_codatatype* :
morphism → *typ* → *Context.generic* → *Context.generic*

The type *'a llist* of lazy lists is already registered; had it not been, you could have told Nitpick about it by adding the following line to your theory file:

declaration {∗
Nitpick_HOL.register_codatatype @{typ "*'a llist*"}
@{const_name *llist_case*}
(*map dest_Const* [@{term *LNil*}, @{term *LCons*}])
∗}

The *register_codatatype* function takes a coinductive datatype, its case function, and the list of its constructors (in addition to the current morphism and generic proof context). The case function must take its arguments in the order that the constructors are listed. If no case function with the correct signature is available, simply pass the empty string.

On the other hand, if your goal is to cripple Nitpick, add the following line to your theory file and try to check a few conjectures about lazy lists:

declaration {∗
Nitpick_HOL.unregister_codatatype @{typ "*'a llist*"}
∗}

Inductive datatypes can be registered as coinductive datatypes, given appropriate coinductive constructors. However, doing so precludes the use of the inductive constructors—Nitpick will generate an error if they are needed.

8 Known Bugs and Limitations

Here are the known bugs and limitations in Nitpick at the time of writing:

- Underspecified functions defined using the **primrec**, **function**, or **nominal_primrec** packages can lead Nitpick to generate spurious counterexamples for theorems that refer to values for which the function is not defined. For example:

  primrec *prec* **where**
  "*prec* (*Suc n*) = *n*"

lemma "*prec* 0 = *undefined*"
nitpick

Nitpick found a counterexample for card nat = 2:

Empty assignment

by (*auto simp*: *prec_def*)

Such theorems are generally considered bad style because they rely on the internal representation of functions synthesized by Isabelle, an implementation detail.

- Similarly, Nitpick might find spurious counterexamples for theorems that rely on the use of the indefinite description operator internally by **specification** and **quot_type**.
- Axioms or definitions that restrict the possible values of the *undefined* constant or other partially specified built-in Isabelle constants (e.g., *Abs_* and *Rep_* constants) are in general ignored. Again, such nonconservative extensions are generally considered bad style.
- Nitpick produces spurious counterexamples when invoked after a **guess** command in a structured proof.
- Datatypes defined using **datatype** and codatatypes defined using **codatatype** that involve nested (co)recursion through non-(co)datatypes are not properly supported and may result in spurious counterexamples.
- Types that are registered with several distinct sets of constructors, including *enat* if the *Coinductive* entry of the *Archive of Formal Proofs* is loaded, can confuse Nitpick.
- The *nitpick_xxx* attributes and the *Nitpick_xxx.register_yyy* functions can cause havoc if used improperly.
- Although this has never been observed, arbitrary theorem morphisms could possibly confuse Nitpick, resulting in spurious counterexamples.
- All constants, types, free variables, and schematic variables whose names start with *Nitpick.* are reserved for internal use.
- Some users report technical issues with the default SAT solver on Windows. Setting the *sat_solver* option (§5.5) to *MiniSat_JNI* should solve this.

References

[1] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, N. Santoro, and S. Whitesides, editors, *WADS 1993*, volume 709 of *Lecture Notes in Computer Science*, pages 61–70. Springer-Verlag, 1993.

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

[3] 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, *Interactive Theorem Proving: ITP-10*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

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

[5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[6] E. Torlak. Kodkod: Constraint solver for relational logic. `http://alloy.mit.edu/kodkod/`.

[7] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer-Verlag, 2007.

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

[9] Wikipedia: AA tree. `http://en.wikipedia.org/wiki/AA_tree`.