Picking Nits

A User's Guide to Nitpick 1.2.1 for Isabelle/HOL 2009

Jasmin Christian Blanchette
Fakultät für Informatik, Technische Universität München

September 25, 2009

Contents

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 tool developed in the 1990s by the Software Design Group.

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*.

Be aware that the tool is experimental and its judgments might be wrong. The known bugs and limitations at the time of writing are listed in §8. Comments and bug reports concerning Nitpick or this manual should be directed to `blanchette@in.tum.de`.

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

2 Installing the Tool

To install Nitpick 1.2.1, download and extract the archive `http://isabelle.in.tum.de/~blanchet/nitpick-1.2.1.tgz`, enter the `nitpick-1.2.1` directory, make sure that Isabelle2009's `bin` directory appears on your path, and run the `build` script.[1] The script performs the following steps:

[1] If you get the error "Unknown logic '*HOL*'—no heap file found," go to the Isabelle root directory and run the Isabelle `build` script, before you retry building Nitpick.

1. It tries to build the efficient SAT solver PicoSAT in the `$ISABELLE_HOME/contrib/PicoSAT` directory. (This step requires a C compiler. A failure to build PicoSAT does not compromise Nitpick's installation.)
2. It installs the Kodkod library, the Kodkodi front-end, and the Java SAT4J solver in `$ISABELLE_HOME/contrib/kodkodi`.
3. It builds the Isabelle logic images `HOL-Nitpick` and `HOL-Nominal-Nitpick` in `$ISABELLE_OUTPUT`.
4. It builds the associated keyword files `isar-keywords-HOL-Nitpick.el` and `isar-keywords-HOL-Nominal-Nitpick.el` in `$ISABELLE_HOME_USER/etc`.

To activate Nitpick, you must invoke `isabelle emacs` with the option `-L HOL-Nitpick` or `-L HOL-Nominal-Nitpick`. Nitpick requires a Java 1.5 virtual machine and expects it to be called `java`. The examples presented in this manual can be found in `Nitpick/examples/ManualNits.thy`.

Note to users of the Aquamacs Isabelle bundle for Mac OS X. Isabelle's `bin` directory is located in `/Applications/Isabelle.app/Contents/Resources/Isabelle-2009/`. To launch Isabelle with Nitpick, ensure that `/Applications/Isabelle.app/Contents/Resources/script` is executable and run it with `-L HOL-Nitpick` or `-L HOL-Nominal-Nitpick`, or pass `-p /Applications/Isabelle.app/Contents/Resources/Emacs.app/Contents/MacOS/Emacs` to `isabelle emacs`.

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 the standard way:

theory *Scratch*
imports *Main*
begin

To obtain the same results as presented here, make sure that PicoSAT is used as the SAT solver and disable multithreading by adding the line

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

after the **begin** keyword. PicoSAT is bundled with Nitpick and should be available if you followed the instructions in §2. Other SAT solvers can also be installed, 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.

Throughout this manual, we will manually invoke the **nitpick** command. Nitpick also provides an automatic mode that can be enabled by specifying

nitpick_params [*auto*]

at the beginning of the theory file. In this mode, Nitpick is run for up to 5 seconds (by default) on every newly entered theorem, much like Auto Quickcheck.

3.1 Propositional Logic

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

lemma "$P \longleftrightarrow Q$"
nitpick

If Nitpick is correctly installed, you should get the following output:

Nitpick found a counterexample:

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

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 "$P\ x \Longrightarrow P\ (\mathrm{THE}\ y.\ P\ y)$"

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 (8 by default), looking for a finite countermodel:

nitpick [*verbose*]

Trying 8 scopes:
card 'a = 1;
card 'a = 2;
⋮
card 'a = 8.

Nitpick found a counterexample for card 'a = 3:

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

Total time: 580 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 "$P\ x \Longrightarrow P\ (\text{THE } y.\ P\ y)$"
nitpick [*show_consts*]

Nitpick found a counterexample for card 'a = 3:

Free variables:
$P = \{a_2, a_3\}$

$x = a_3$

Constant:

The fallback $= a_1$

We can see more clearly now. Since the predicate P isn't true for a unique value, THE y. $P\ y$ can denote any value of type $'a$, even a_1. Since $P\ a_1$ is false, the entire formula is falsified.

As an optimization, Nitpick's preprocessor introduced the special constant "*The* fallback" corresponding to the expression THE y. $P\ y$ (i.e., *The* $(\lambda y.\ P\ y)$) when there doesn't exist a unique y satisfying $P\ y$. We can disable this optimization by passing the *full_descrs* option:

nitpick [*full_descrs*, *show_consts*]

Nitpick found a counterexample for card 'a = 3:

Free variables:

$P = \{a_2, a_3\}$

$x = a_3$

Constant:

THE $y.\ P\ y = a_1$

As the result of another optimization, Nitpick directly assigned a value to the subterm THE y. $P\ y$, rather than to the *The* constant. If we disable this second optimization by using the command

nitpick [*special_depth* = -1, *full_descrs*, *show_consts*]

we finally get *The*:

Constant:

$$\textit{The} = \textit{undefined}(\{\} := a_3,\ \{a_3\} := a_3,\ \{a_2\} := a_2,\ \{a_2, a_3\} := a_1,\ \{a_1\} := a_1,\ \{a_1, a_3\} := a_3,\ \{a_1, a_2\} := a_3,\ \{a_1, a_2, a_3\} := a_3)$$

Notice that *The* $(\lambda y.\ P\ y) =$ *The* $\{a_2, a_3\} = a_1$, just like before.[2]

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 "$\exists!x.\ P\ x \implies P\ (\text{THE } y.\ P\ y)$"

The fix appears to work:

nitpick

Nitpick found no counterexample.

[2]The *undefined* symbol's presence is explained as follows: In higher-order logic, any function can be built from the undefined function using repeated applications of the function update operator $f(x := y)$, just like any list can be built from the empty list using $x \# xs$.

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

nitpick [*card 'a* = 1–50][3]

Nitpick found no counterexample.

Let's see if Sledgehammer [11] can find a proof:

sledgehammer

Sledgehammer: external prover "e" for subgoal 1:
$\exists!x.\ P\ x \Longrightarrow P\ (THE\ y.\ P\ y)$
Try this command: apply (metis the_equality)

apply (*metis the_equality*)

No subgoals!

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:
$f = undefined(b_1 := a_1)$
Skolem constants:
$g = undefined(a_1 := b_1,\ a_2 := b_1)$
$y = a_2$

Although f is the only free variable occurring in the formula, Nitpick also displays values for the bound variables g and 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:
$\lambda x.\ f = undefined(a_1 := undefined(a_1 := a_2,\ a_2 := a_1),$
$a_2 := undefined(a_1 := a_1,\ a_2 := a_1))$

[3]The symbol '–' can be entered as - (hyphen) or `\<midarrow>`.

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 the *sym* constant 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.

Although skolemization is a useful optimization, you can disable it by invoking Nitpick with *skolem_depth* = -1. See §5.5 for details.

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 considers prefixes $\{0, 1, \ldots, K-1\}$ of *nat* (where $K =$ *card nat*) and maps all other numbers to the undefined value (?). The type *int* is handled in a similar way: If $K =$ *card int*, the subset of *int* known to Nitpick is $\{-\lceil K/2\rceil + 1, \ldots, +\lfloor K/2\rfloor\}$. 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:
$i = 0$
$j = 1$
$m = 1$
$n = 0$

With infinite types, we don't always have the luxury of a genuine counterexample and must often content ourselves with a potential one. The tedious task of finding

out whether the potential counterexample is in fact genuine can be outsourced to *auto* by passing the option *check_potential*. For example:

lemma "$\forall n.\ Suc\ n \neq n \implies P$"
nitpick [*card nat* = 100, *check_potential*]

Nitpick found a potential counterexample:

Free variable:
$P = False$

Confirmation by "auto": The above counterexample is genuine.

You might wonder why the counterexample is first reported as potential. The root of the problem 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 satisfied, the putative lemma can never be falsified.

Incidentally, if you distrust the so-called genuine counterexamples, you can enable *check_genuine* to verify them as well. However, be aware that *auto* will often fail to prove that the counterexample is genuine or spurious.

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

lemma "$P\ Suc$"
nitpick [*card* = 1–6]

Nitpick found no counterexample.

For any cardinality k, *Suc* is the partial function $\{0 \mapsto 1, 1 \mapsto 2, \ldots, k-1 \mapsto ?\}$, which evaluates to ? when it is passed as argument to P. As a result, $P\ Suc$ is always ?. The next example is similar:

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

Nitpick found a counterexample:

Free variable:
$P = \{\}$

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 8 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:*
> $xs = []$
> $y = a_3$

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

> *Datatype:*
> *'a list* = $\{[], [a_3, a_3], [a_3], \ldots\}$
> *Constants:*
> $\lambda x_1.\ x_1\ @\ [y, y] = undefined([] := [a_3, a_3],\ [a_3, a_3] := ?,\ [a_3] := ?)$
> $hd = undefined([] := a_2,\ [a_3, a_3] := a_3,\ [a_3] := a_3)$

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

The second constant, $\lambda x_1.\ x_1\ @\ [y, y]$, is simply the append operator whose second argument is fixed to be $[y, y]$. Appending $[a_3, a_3]$ to $[a_3]$ would normally give $[a_3, a_3, a_3]$, but this value is not representable in the subset of *'a list* considered by Nitpick, which is shown under the "Datatype" heading; hence the result is ?. Similarly, appending $[a_3, a_3]$ to itself gives ?.

Given *card 'a* $= 3$ and *card 'a list* $= 3$, Nitpick considers the following subsets (omitting ...):

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

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 $\mathcal{S} = \{[], [a_1], [a_1, a_3]\}$, and observe that $[a_1, a_3]$ (i.e., $a_1 \# [a_3]$) has $[a_3] \notin \mathcal{S}$ as a subterm.

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

lemma "$[\![\mathit{length}\ \mathit{xs} = 1;\ \mathit{length}\ \mathit{ys} = 1]\!] \Longrightarrow \mathit{xs} = \mathit{ys}$"
nitpick [*show_datatypes*]

Nitpick found a counterexample for card 'a = 3:

Free variables:
$\mathit{xs} = [a_2]$
$\mathit{ys} = [a_3]$
Datatypes:
$\mathit{nat} = \{0, 1, 2, \ldots\}$
$'a\ \mathit{list} = \{[], [a_3], [a_2], \ldots\}$

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, 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** "$A \equiv \mathit{Abs\_three}\ 0$"
definition *B* :: *three* **where** "$B \equiv \mathit{Abs\_three}\ 1$"
definition *C* :: *three* **where** "$C \equiv \mathit{Abs\_three}\ 2$"

lemma "$[\![P\ A;\ P\ B]\!] \Longrightarrow P\ x$"
nitpick [*show_datatypes*]

Nitpick found a counterexample:

Free variables:
$P = \{\langle\!\langle 1 \rangle\!\rangle, \langle\!\langle 0 \rangle\!\rangle\}$
$x = \langle\!\langle 2 \rangle\!\rangle$
Datatypes:
$\mathit{nat} = \{0, 1, 2, \ldots\}$
$\mathit{three} = \{\langle\!\langle 2 \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle, \langle\!\langle 0 \rangle\!\rangle, \ldots\}$

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

Records, which are implemented as **typedef**s behind the scenes, are handled in much the same way:

record *point* =
  Xcoord :: *int*
  Ycoord :: *int*

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

Nitpick found a counterexample:

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

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

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

Nitpick found a counterexample:

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

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.[4] 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* = 100, *verbose*]

The inductive predicate "even" was proved well-founded. Nitpick can compute it efficiently.

Trying 1 scope:
card nat = 100.

Nitpick found a potential counterexample for card nat = 100:

Empty assignment

Nitpick could not find a better counterexample.

Total time: 2274 ms.

No genuine counterexample is possible because Nitpick cannot rule out the existence of a natural number $n \geq 100$ such that both *even n* and *even* $(\mathit{Suc}\ n)$ are true. To help Nitpick, we can bound the existential quantifier:

lemma "$\exists n \leq 99.\ \mathit{even}\ n \wedge \mathit{even}\ (\mathit{Suc}\ n)$"
nitpick [*card nat* = 100]

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.

[4] 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.

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 "$\exists n \in \{0, 2, 4, 6, 8\}. \neg$ *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*′ = *undefined*$(2 := \{0, 2, 4, 6, 8, 1^?, 3^?, 5^?, 7^?, 9^?\},$
$1 := \{0, 2, 4, 1^?, 3^?, 5^?, 6^?, 7^?, 8^?, 9^?\},$
$0 := \{0, 2, 1^?, 3^?, 4^?, 5^?, 6^?, 7^?, 8^?, 9^?\})$

Total time: 1140 ms.

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 $\lambda i.$ *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.

Some values are marked with superscripted question marks ('?'). These are the elements for which the predicate evaluates to ?. Thus, *even*′ evaluates to either *True* or ?, never *False*.

When unrolling a predicate, Nitpick tries 0, 1, 2, 4, 8, 12, 16, and 24 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 "*even*′ $(n - 2) \Longrightarrow$ *even*′ *n*"
nitpick [*card nat* = 10, *show_consts*]

Nitpick found a counterexample:

Free variable:
$n = 1$
Constants:
$\lambda i.\ even' = undefined(0 := \{0, 2, 1^?, 3^?, 4^?, 5^?, 6^?, 7^?, 8^?, 9^?\})$
$even' \subseteq \{0, 2, 4, 6, 8, \ldots\}$

Notice the special constraint $even' \subseteq \{0, 2, 4, 6, 8, \ldots\}$ in the output, whose right-hand side represents an arbitrary fixed point (not necessarily the least one). It is used to falsify $even'\ n$. In contrast, the unrolled predicate is used to satisfy $even'\ (n - 2)$.

Coinductive predicates are handled dually. For example:

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

lemma "*nats* $= \{0, 1, 2, 3, 4\}$"
nitpick [*card nat* = 10, *show_consts*]

Nitpick found a counterexample:

Constants:
$\lambda i.\ nats = undefined(0 := \{0^?, 1^?, 2^?, 3^?, 4^?, 5^?, 6^?, 7^?, 8^?, 9^?, ?\})$
$nats \supseteq \{5, 6, 7, 8, 9, ?\}$

3.9 Coinductive Datatypes

While Isabelle regrettably lacks a high-level mechanism for defining coinductive datatypes, the *Coinductive_List* theory provides a coinductive "lazy list" datatype, *'a llist*, defined the hard way. Nitpick supports these lazy lists seamlessly and provides a hook, described in §7.2, to register custom coinductive datatypes.

(Co)intuitively, 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 lazy lists using the *LNil* :: *'a llist* and *LCons* :: *'a* $\Rightarrow$ *'a llist* $\Rightarrow$ *'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:

lemma "*xs* $\neq$ *LCons a xs*"
nitpick

Nitpick found a counterexample for card 'a = 1:

Free variables:
$a = a_1$
$xs = \mu X.\ LCons\ a_1\ X$

The ad hoc notation $\mu X.\ t(X)$ in the output abbreviates THE $x.\ x = t(x)$. Hence, *xs* is simply the infinite list $[a_1, a_1, a_1, \ldots]$.

The next example is more interesting:

> **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 8 scopes:*
> *card 'a = 1, card "'a list" = 1, and bisim_depth = 0.*
> ⋮
> *card 'a = 8, card "'a list" = 8, and bisim_depth = 7.*
>
> *Nitpick found a counterexample for card 'a = 2, card "'a list" = 2, and bisim_depth = 1:*
>
> *Free variables:*
> $a = a_2$
> $b = a_1$
> $xs = \mu X.\ LCons\ a_2\ X$
> $ys = LCons\ a_1\ (\mu X.\ LCons\ a_2\ X)$
>
> *Total time: 726 ms.*

The lazy list *xs* is simply $[a_2, a_2, a_2, \ldots]$, whereas *ys* is $[a_1, a_2, a_2, a_2, \ldots]$, i.e., a lasso-shaped list with $[a_1]$ as its stem and $[a_2]$ as its cycle. In general, the list segment within the scope of the μ 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 lazy lists $\mu X.\ LCons\ a\ (LCons\ b\ X)$ and $LCons\ a\ (\mu X.\ LCons\ b\ (LCons\ a\ X))$ 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 often 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 "likely genuine" and Nitpick recommends to try again with *bisim_depth* set to a nonnegative integer. Disabling the check for the previous example saves approximately 150 milliseconds; the speed gains can be more significant for larger scopes.

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

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

Nitpick found a likely genuine counterexample for card 'a = 2*:*

Free variables:
$a = a_2$
$xs = \mu X.\ LCons\ a_2\ X$
$ys = \mu X.\ LCons\ a_2\ X$
Codatatype:
'a *llist* $= \{\mu X.\ LCons\ a_2\ X,\ \mu X.\ LCons\ a_2\ X,\ \ldots\}$

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 *llist* are bisimilar.

A compromise between leaving out the bisimilarity predicate from the Kodkod problem and performing the after-the-fact check is to specify a lower nonnegative *bisim_depth* value than the default one provided by Nitpick. 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. Be aware that 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* (λ*n*. case *n* of 0 ⇒ *Var* 0 | *Suc m* ⇒ *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 "¬ *loose t* 0 ⟹ *subst* σ *t* = *t*"
nitpick [*verbose*]

Trying 8 scopes:
card nat = 1, *card tm* = 1, and *card* "(*nat* ⇒ *tm*) × *tm*" = 1;
card nat = 2, *card tm* = 2, and *card* "(*nat* ⇒ *tm*) × *tm*" = 2;
⋮
card nat = 8, *card tm* = 8, and *card* "(*nat* ⇒ *tm*) × *tm*" = 8.

Nitpick found a counterexample for card nat = 6, card tm = 6, and card "(nat ⇒ tm) × tm" = 6:

Free variables:
σ = *undefined*(0 := *Var* 0, 1 := *Var* 0, 2 := *Var* 0, 3 := *Var* 0,
4 := *Var* 0, 5 := *Var* 0)
t = *Lam* (*Lam* (*Var* 1))

Total time: 5760 *ms.*

Using *eval*, we find out that *subst* σ *t* = *Lam* (*Lam* (*Var* 0)). Using the traditional λ-term notation, *t* is λ*x y*. *x* whereas *subst* σ *t* is λ*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 8 to the type $(nat \Rightarrow tm) \times tm$, which corresponds to the uncurried domain of the *subst* function.[5] For the formula of interest, knowing 6 values of that type was enough to find the counterexample. Without boxing, 279 936 ($= 6^6 * 6$) values must be considered, a hopeless undertaking:

nitpick [*dont_box*]

Nitpick ran out of time after checking 4 of 8 scopes.

Boxing can be enabled or disabled globally or on a per-type basis using the *box* option. In addition, setting the cardinality of a function or product type implicitly enables boxing for that type. Nitpick normally performs reasonable choices about which types should be boxed, but option tweaking occasionally 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 1–8, no fewer than $8^4 = 4096$ 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 "$length\ xs = length\ ys \implies rev\ (zip\ xs\ ys) = zip\ xs\ (rev\ ys)$"

where *xs* is of type $'a\ list$ and *ys* is of type $'b\ list$. A priori, Nitpick would need to consider 512 scopes to exhaust the specification *card* = 1–8. 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 8 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,

[5] Uncurrying can be disabled by passing the *dont_uncurry* option.

card "'a list" = 2, and card "'b list" = 2.
⋮
card 'a = 8, card 'b = 8, card nat = 8, card "('a × 'b) list" = 8,
card "'a list" = 8, and card "'b list" = 8.

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:
$xs = [a_4, a_5]$
$ys = [b_3, b_3]$

Total time: 1636 ms.

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

nitpick [*card* = 8]

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 "$\exists g.\ \forall x{::}'b.\ g\ (f\ x) = x \implies \forall y{::}'a.\ \exists x.\ y = f\ x$"
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* $\leq$ *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*, *real*, and some **typedef**'d types. Thus, given the cardinality specification 1–8, a formula involving *nat*, *int*, *int list*, *rat*, and *rat list* will lead Nitpick to consider only 8 scopes instead of 32 768.

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 re-

view 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, *max_threads* = 2]

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:

$$\begin{array}{l} S ::= \epsilon \mid bA \mid aB \\ A ::= aS \mid bAA \\ B ::= bS \mid aBB \end{array}$$

The intuition behind the grammar is that A generates all string 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* | *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*:

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

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*:

"$[\![v \in B; v \in B]\!] \implies a \# v @ w \in B$"

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

"$[\![v \in A; w \in A]\!] \implies b \# v @ w \in A$"

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 tree* = Λ | N "*'a*::*linorder*" *nat* "*'a tree*" "*'a tree*"

primrec *data* **where**
"*data* $\Lambda = \mathit{undefined}$" |
"*data* $(N\ x\ \_\ \_\ \_) = x$"

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

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 [12]:

> 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* Λ = *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 7 of 8 scopes.

Furthermore, applying *skew* or *split* to 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* ⟹ *wf* (*insort t x*)"
nitpick

Nitpick found a counterexample for card 'a = 4:

Free variables:
$t = N\ a_3\ 1\ \Lambda\ \Lambda$
$x = a_4$

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* ⟹ *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 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* ⟹ *wf* (*insort t x*)"
nitpick

Nitpick ran out of time after checking 6 of 8 scopes.

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

theorem *dataset_insort*: "*dataset* (*insort t x*) = {*x*} ∪ *dataset t*"
nitpick

Nitpick ran out of time after checking 5 of 8 scopes.

We could continue like this and sketch a complete theory of AA trees without performing a single proof. 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 *s*]

The options are categorized as follows: mode of operation (§5.1), scope of search (§5.2), output format (§5.3), automatic counterexample checks (§5.4), optimizations (§5.5), and timeouts (§5.6). The descriptions below refer to the following syntactic quantities:

- ⟨***string***⟩: A string.
- ⟨***bool***⟩: *true* or *false*.
- ⟨***bool_or_smart***⟩: *true*, *false*, or *smart*.
- ⟨***int***⟩: An integer. Negative integers are prefixed with a hyphen.
- ⟨***int_or_smart***⟩: An integer or *smart*.
- ⟨***int_range***⟩: An integer (e.g., 3) or a range of nonnegative integers (e.g., 1–4). The range symbol '–' can be entered as `-` (hyphen) or `\<midarrow>`.
- ⟨***int_seq***⟩: A comma-separated sequence of ranges of integers (e.g., 1,3,6–8).
- ⟨***time***⟩: An integer followed by *min* (minutes), *s* (seconds), or *ms* (milliseconds), or the keyword *none* (∞ years).
- ⟨***const***⟩: The name of a HOL constant.
- ⟨***term_list***⟩: A space-separated list of HOL terms (e.g., "*f x*" "*g y*").
- ⟨***type***⟩: A HOL type.

Default values are indicated in square brackets. Boolean options have a negated counterpart (e.g., *auto* vs. *no_auto*). When setting Boolean options, "= *true*" may be omitted.

5.1 Mode of Operation

auto $[= \langle bool \rangle]$ **[*false*]** **(neg.: *no_auto*)**

Specifies whether Nitpick should be run automatically on newly entered theorems. For automatic runs, *user_axioms* (§5.1) and *assms* (§5.1) are implicitly enabled, *blocking* (§5.1), *verbose* (§5.3), and *debug* (§5.3) are disabled, *max_potential* (§5.3) is taken to be 0, and *auto_timeout* (§5.6) is used as the time limit instead of *timeout* (§5.6). The output is also more concise.

See also *auto_timeout* (§5.6).

blocking $[= \langle bool \rangle]$ **[*true*]** **(neg.: *non_blocking*)**

Specifies whether the **nitpick** command should block. The non-blocking mode lets the user start proving the putative theorem while Nitpick looks for a counterexample, but it can also be more confusing. For technical reasons, automatic runs always block.

See also *auto* (§5.1).

falsify $[= \langle bool \rangle]$ **[*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 $[= \langle bool\_or\_smart \rangle]$ **[*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 "likely genuine." The *debug* (§5.3) option can be used to find out which axioms were considered.

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

assms $[= \langle bool \rangle]$ **[*true*]** **(neg.: *no_assms*)**

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

See also *auto* (§5.1).

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

Specifies whether Nitpick should generate 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. This option is disabled by default unless your home directory ends with `blanchet` or `blanchette`.

See also *default* (§5.3).

5.2 Scope of Search

card ⟨***type***⟩ **=** ⟨***int_seq***⟩

Specifies the sequence of cardinalities to use for a given type. For *nat* and *int*, 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:

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

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*. Although function and product types are normally mapped directly to the corresponding Kodkod concepts, setting the cardinality of such types is also allowed and implicitly enables "boxing" for them, as explained in the description of the *box* ⟨*type*⟩ and *box* (§5.2) options.

card **=** ⟨***int_seq***⟩ **[1–8]**

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

max ⟨***const***⟩ **=** ⟨***int_seq***⟩

Specifies the sequence of maximum multiplicities to use for a given (co)inductive datatype constructor. The multiplicity of a constructor 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 **=** ⟨***int_seq***⟩

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* ⟨*const*⟩ option described above.

wf ⟨***const***⟩ [**=** ⟨***bool_or_smart***⟩] **(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 "likely genuine."

- ***false***: Treat the (co)inductive predicate as if it were not well-founded. The predicate is then unrolled according to the *iter* ⟨*const*⟩ and *iter* options if necessary.
- ***smart***: Try to prove that the inductive predicate is well-founded using Isabelle's *lexicographic_order* and *sizechange* 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 *tac_timeout* (§5.6).

wf [= ⟨***bool_or_smart***⟩] **[*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.

iter **=** ⟨***int_seq***⟩ **[1,2,4,8,12,16,24,32]**

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***⟩ **[7]**

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 "likely 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***⟩ [= ⟨***bool_or_smart***⟩] **(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.

Setting the *card* ⟨*type*⟩ option for a function or product type implicitly enables boxing for that type.

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

box [= ⟨***bool_or_smart***⟩] **[*smart*]** **(neg.: *dont_box*)**

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

mono ⟨***type***⟩ [= ⟨***bool_or_smart***⟩] **(neg.: *non_mono*)**

Specifies whether the specified type should be considered monotonic when enumerating scopes. 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 also diminishes the chances of finding a counterexample, as demonstrated in §3.11. Use with care.

See also *verbose* (§5.3).

mono [= ⟨***bool_or_smart***⟩] **[*smart*]** **(neg.: *non_box*)**

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

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.

See also *auto* (§5.1), *card* (§5.2), *max* (§5.2), and *iter* (§5.2).

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 *auto* (§5.1), *overlord* (§5.1), and *batch_size* (§5.5).

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.

See also *skolemize_depth* (§5.5).

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

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 potential counterexample is genuine or spurious, but their potential for clutter is real.

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

Enabling this option effectively enables *show_skolems*, *show_datatypes*, and *show_consts*.

max_potential $= \langle \textit{int} \rangle$ **[1]**

Specifies the maximum number of potential 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.

See also *auto* (§5.1).

eval $= \langle \textit{term\_list} \rangle$

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.

See also *check_potential* (§5.4).

expect $= \langle \textit{string} \rangle$

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

- ***genuine***: Nitpick found a genuine counterexample.
- ***likely_genuine***: Nitpick found a likely genuine counterexample.
- ***potential***: Nitpick found a potential 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.

5.4 Authentication

check_potential $[= \langle bool \rangle]$ **[*false*]** **(neg.: *trust_potential*)**

Specifies whether potential counterexamples should be given to Isabelle's *auto* tactic to assess their validity. If a potential counterexample is shown to be genuine, Nitpick displays a message to this effect and terminates.

See also *max_potential* (§5.3) and *auto_timeout* (§5.6).

check_genuine $[= \langle bool \rangle]$ **[*false*]** **(neg.: *trust_genuine*)**

Specifies whether genuine and likely genuine counterexamples should be given to Isabelle's *auto* tactic to assess their validity. If a "genuine" counterexample is shown to be spurious, the user is kindly asked to send a bug report to the author at `blanchette@in.tum.de`.

See also *peephole_optim* (§5.5) and *auto_timeout* (§5.6).

5.5 Optimizations

sat_solver $= \langle string \rangle$ **[*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. The supported solvers are listed below:

- ***SAT4J***: SAT4J is a reasonably efficient solver written in Java. 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.
- ***SAT4JLight***: Variant of SAT4J that is optimized for small problems.
- ***PicoSAT***: PicoSAT is an efficient solver written in C. It is bundled with Kodkodi and requires no further installation or configuration steps. Alternatively, you can install a standard version of PicoSAT and set the environment variable `PICOSAT_HOME` to the directory that contains the `picosat` executable or put the executable in `$ISABELLE_HOME/contrib/PicoSAT`. The C sources for PicoSAT are available at `http://fmv.jku.at/picosat/` and are also bundled with Kodkodi. Nitpick has been tested with version 913.
- ***MiniSat***: MiniSat is an efficient solver written in C++. You can use the version for Java bundled in `nativesolver.tgz`, which you will find on Kodkod's web site [9]. Alternatively, you can install a standard version of MiniSat and set the environment variable `MINISAT_HOME` to the directory that contains the `minisat` executable or put the executable in `$ISABELLE_HOME/contrib/MiniSat`. The C++ sources and executables for MiniSat are available at `http://minisat.se/MiniSat.html`. Nitpick has been tested with versions 1.14 and 2.0 beta (2007-07-21).

- ***zChaff***: zChaff is an efficient solver written in C++. You can use the version for Java bundled in `nativesolver.tgz`, which you will find on Kodkod's web site [9]. Alternatively, you can install a standard version of zChaff and set the environment variable `ZCHAFF_HOME` to the directory that contains the `zchaff` executable or put the executable in `$ISABELLE_HOME/contrib/zChaff`. The C++ sources and executables for zChaff are available at `http://www.princeton.edu/~chaff/zchaff.html`. Nitpick has been tested with versions 2004-05-13, 2004-11-15, and 2007-03-12.
- ***RSat***: RSat is an efficient solver written in C++. To use RSat, set the environment variable `RSAT_HOME` to the directory that contains the `rsat` executable or put the executable in `$ISABELLE_HOME/contrib/RSat`. 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 or put the executable in `$ISABELLE_HOME/contrib/BerkMin`. The BerkMin executables are available at `http://eigold.tripod.com/BerkMin.html`.
- ***BerkMinAlloy***: Variant of BerkMin that is included with Alloy 4 and calls itself "sat56" in its banner text. To use this version of BerkMin, set the environment variable `BERKMINALLOY_HOME` to the directory that contains the `berkmin` executable or put the executable in `$ISABELLE_HOME/contrib/BerkMinAlloy`.
- ***Jerusat***: Jerusat 1.3 is an efficient solver written in C. To use Jerusat, set the environment variable `JERUSAT_HOME` to the directory that contains the `Jerusat1.3` executable or put the executable in `$ISABELLE_HOME/contrib/Jerusat`. The C sources for Jerusat are available at `http://www.cs.tau.ac.il/~ale1/Jerusat1.3.tgz`.
- ***HaifaSat***: HaifaSat 1.0 beta is an experimental solver written in C++. To use HaifaSat, set the environment variable `HAIFASAT_HOME` to the directory that contains the `HaifaSat` executable or put the executable in `$ISABELLE_HOME/contrib/HaifaSat`. The C++ sources for HaifaSat are available at `http://cs.technion.ac.il/~gershman/HaifaSat.htm`.
- ***smart***: If *sat_solver* is set to *smart*, Nitpick selects the first solver among MiniSat, PicoSAT, zChaff, RSat, BerkMin, BerkMinAlloy, and Jerusat that is recognized by Isabelle. If none is found, it falls back on SAT4J, which should always be available. If *verbose* is enabled, Nitpick displays which SAT solver was chosen.

batch_size = ⟨***int_or_smart***⟩ [***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 64 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).

special_depth = ⟨***int***⟩ **[*20*]**

Specifies the maximum depth at which functions invoked with fixed arguments should be specialized. The value -1 disables function specialization, 0 means that only constants occurring in the formula of interest are specialized, 1 means that constants occurring in the formula's immediate axioms are specialized, and in general $n > 0$ means that constants occurring in the formula's nth-level axioms are specialized. This optimization can drastically reduce the search space, especially for higher-order functions.

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

skolem_depth = ⟨***int***⟩ **[*4*]**

Specifies the maximum depth at which skolemization should take place, expressed as the number of outer quantifiers. The value -1 disables skolemization, 0 means that only the outermost $\forall$-quantifiers (or negated $\exists$-quantifiers) in the original (unnegated) formula are skolemized, and $n > 0$ means that $\forall$-quantifiers within the scope of at most n $\exists$-quantifiers are skolemized. However, for performance reasons, $\forall$-quantifiers that occur in the scope of a higher-order $\exists$-quantifier are left unchanged.

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

uncurry [= ⟨***bool***⟩] **[*true*]** **(neg.: *dont_uncurry*)**

Specifies whether Nitpick should uncurry functions. Uncurrying has on its own no effect on efficiency, but it creates opportunities for the boxing optimization.

fast_descrs [= ⟨***bool***⟩] **[*true*]** **(neg.: *full_descrs*)**

Specifies whether Nitpick should optimize the definite and indefinite description operators (THE and SOME). The optimized versions usually help Nitpick generate more counterexamples or at least find them faster, but only the unoptimized versions are complete when all types occurring in the formula are finite.

peephole_optim [= ⟨***bool***⟩] **[*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.

sym_break = ⟨***int***⟩ **[*20*]**

Specifies an upper bound on the number of relations for which Kodkod generates symmetry breaking predicates. According to the Kodkod documentation [10], "in general, the higher this value, the more symmetries will be broken, and the faster the formula will be solved. But, setting the value too high may have the opposite effect and slow down the solving."

sharing_depth = ⟨***int***⟩ **[*3*]**

Specifies the depth to which Kodkod should check circuits for equivalence during the translation to SAT. The default of 3 is the same as in Alloy. The minimum allowed depth is 1. Increasing the sharing may result in a smaller SAT problem, but can also slow down Kodkod.

flatten_props [= ⟨***bool***⟩] **[*false*]** **(neg.: *dont_flatten_props*)**

Specifies whether Kodkod should try to eliminate intermediate Boolean variables. Although this might sound like a good idea, in practice it can drastically slow down Kodkod.

max_threads = ⟨***int***⟩ **[*0*]**

Specifies the maximum number of threads to use in Kodkod. If this option is set to a value less than 1, Kodkod will compute an appropriate value based on the number of processor cores available.

5.6 Timeouts

timeout = ⟨***time***⟩ **[30 *s*]**

Specifies the maximum amount of time 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, *auto_timeout* is used instead.

See also *auto* (§5.1).

auto_timeout = ⟨***time***⟩ **[5 *s*]**

Specifies the maximum amount of time that Nitpick should use to find a counterexample when running automatically. Nitpick tries to honor this constraint as well as it can but offers no guarantees.

See also *auto* (§5.1).

tac_timeout = ⟨***time***⟩ **[500 *ms*]**

Specifies the maximum amount of time that the *auto* tactic should use when

checking a counterexample, and similarly that *lexicographic_order* and *size-change* should use when checking whether a (co)inductive predicate is well-founded. Nitpick tries to honor this constraint as well as it can but offers no guarantees.

See also *wf* (§5.2), *check_potential* (§5.4), and *check_genuine* (§5.4).

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_const_def

This attribute specifies an alternative definition of a constant. The alternative definition should be logically equivalent to the constant's actual axiomatic definition and should be of the form

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

where $?x_1, \ldots, ?x_n$ are distinct variables and c does not occur in t.

nitpick_const_simp

This attribute specifies the equations that constitute the specification of a constant. For functions defined using the **primrec**, **function**, and **nominal_primrec** packages, this corresponds to the *simps* rules. The equations must be of the form

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

nitpick_const_psimp

This attribute specifies the equations that constitute the partial specification of a constant. For functions defined using the **function** package, this corresponds to the *psimps* rules. The conditional equations must be of the form

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

nitpick_ind_intro

This attribute specifies the introduction rules of a (co)inductive predicate. For predicates defined using the **inductive** or **coinductive** command, this corresponds to the *intros* rules. The introduction rules must be of the form

$$[\![P_1; \ldots; P_m; c\ t_{11}\ \ldots\ t_{1n}; \ldots; c\ t_{k1}\ \ldots\ t_{kn}]\!] \Longrightarrow c\ u_1\ \ldots\ u_n.$$

When faced with a constant, Nitpick proceeds as follows:

1. If the *nitpick_const_simp* set associated with the constant is not empty, Nitpick uses these rules as the specification of the constant.
2. Otherwise, if the *nitpick_const_psimp* set associated with the constant is not empty, it uses these rules as the specification of the constant.
3. Otherwise, it looks up the definition of the constant:
    1. If the *nitpick_const_def* 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.
    2. If the definition is of the form
    $$c\ ?x_1\ \ldots\ ?x_m \equiv \lambda y_1\ \ldots\ y_n.\ lfp\ (\lambda f.\ t),$$
    then Nitpick assumes that the definition was made using an inductive package and based on the introduction rules marked with *nitpick_ind_intros* tries to determine whether the definition is well-founded.

As an illustration, consider the inductive definition

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

Isabelle automatically attaches the *nitpick_ind_intro* attribute to the above rules. Nitpick then uses the *lfp*-based definition in conjunction with these rules. To override this, we can specify an alternative definition as follows:

lemma *odd_def*′ [*nitpick_const_def*]: "*odd n* $\equiv$ *n* mod 2 = 1"

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

lemma *odd_simp*′ [*nitpick_const_simp*]: "*odd n* = (*n* mod 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_const_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.

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 coinductive datatypes.

7.1 Invocation of Nitpick

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

val *pick_nits_in_term* :
  Proof.state → *params* → *bool* → *term list* → *term* → *string* ∗ *Proof.state*
val *pick_nits_in_subgoal* :
  Proof.state → *params* → *bool* → *int* → *string* ∗ *Proof.state*

The return value is a new proof state paired with an outcome string ("genuine", "likely_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 *NitpickIsar* 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:

val *params* = *NitpickIsar.default_params thy* [("timeout", "none")]
val (*outcome*, *state*′) = *Nitpick.pick_nits_in_subgoal state params false subgoal*

7.2 Registration of Coinductive Datatypes

If you have defined a custom coinductive datatype, you can tell Nitpick about it, so that it can use an efficient Kodkod axiomatization similar to the one it uses for lazy lists. The interface for registering and unregistering coinductive datatypes consists of the following pair of functions defined in the *Nitpick* structure:

val *register_codatatype* : *typ* → *string* → *styp list* → *theory* → *theory*
val *unregister_codatatype* : *typ* → *theory* → *theory*

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:

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

The *register_codatatype* function takes a coinductive type, its case function, and the list of its constructors. 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:

setup {∗ *Nitpick.unregister_codatatype* @{typ "*'a list*"} ∗}

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 considered bad style because they rely on the internal representation of functions synthesized by Isabelle, which is an implementation detail.

- Nitpick produces spurious counterexamples when invoked after a **guess** command in a structured proof.
- The *nitpick_* attributes and the *Nitpick.register_codatatype* function can cause havoc if used improperly.
- Local definitions are not supported and result in an error.
- All constants and types whose names start with *Nitpick.* or *NitpickDefs.* are reserved for internal use.

References

[1] Andersson, A.: Balanced search trees made simple. In: Dehne, F. K. H. A., Santoro, N., Whitesides, S. (eds.) *WADS 1993*, LNCS vol. 709, pp. 61–70. Springer, Heidelberg (1993)

[2] Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) *SEFM 2004*, pp. 230–239. IEEE C.S. (2004)

[3] Blanchette, J. C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder (extended abstract). In *TAP 2009: Short Papers*, ETH Technical Report 630 (2009)

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

[5] Nipkow, T., Paulson, L. C., Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS vol. 2283. Springer, Heidelberg (2002). Latest version `http://isabelle.in.tum.de/doc/tutorial.pdf` (2009)

[6] Paulson, L. C.: A fixedpoint approach to (co)inductive and (co)datatype definitions. Latest version `http://isabelle.in.tum.de/doc/ind-defs.pdf` (2009)

[7] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*, LNCS vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

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

[9] Kodkod: Constraint Solver for Relational Logic, `http://alloy.mit.edu/kodkod/`

[10] Kodkod API: Class Options, `http://alloy.mit.edu/kodkod/docs/kodkod/engine/config/Options.html`

[11] The Sledgehammer: Let Automatic Theorem Provers Write Your Isabelle Scripts, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/sledgehammer.html`

[12] Wikipedia: AA Tree, `http://en.wikipedia.org/wiki/AA_tree`