

Picking Nits

A User's Guide to Nitpick 1.0.2 for the Isabelle/HOL 2009 Snapshots

Jasmin Christian Blanchette
Fakultät für Informatik, Technische Universität München

March 12, 2009

Contents

1	Introduction	2
2	Installing the Tool	2
3	First Steps	2
3.1	Propositional Logic	3
3.2	Type Variables	3
3.3	Constants	4
3.4	Skolemization	6
3.5	Natural Numbers and Integers	7
3.6	Inductive Datatypes	9
3.7	Typedefs and Records	10
3.8	Inductively Defined Predicates	11
4	Case Studies	14
4.1	A Context-Free Grammar	14
4.2	AA Trees	16
5	Option Reference	19
5.1	Mode of Operation	20
5.2	Scope of Search	20
5.3	Output Format	22
5.4	Automatic Counterexample Checks	23
5.5	Optimizations	23
5.6	Timeouts	26
6	Known Bugs and Limitations	27

1 Introduction

Nitpick is a new counterexample generator for Isabelle/HOL [5] that is designed to handle formulas combining inductive datatypes, 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 stolen from a (now obsolete) 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 behind the scenes 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*.

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

2 Installing the Tool

To install Nitpick 1.0.2, download and extract the archive <http://isabelle.in.tum.de/~blanchet/nitpick-1.0.2.tgz>, then run the build script. The script performs the following steps:

1. It installs the Kodkod library, the Kodkodi front-end, and the portable SAT4J solver in the `$ISABELLE_HOME/contrib/kodkodi` directory.
2. It builds a HOL-Nitpick Isabelle image in the `$ISABELLE_OUTPUT` directory.
3. It builds an associated keyword file `isar-keywords-HOL-Nitpick.el` in the `$ISABELLE_HOME_USER/etc` directory.

To activate Nitpick, you must either invoke `isabelle emacs` with the option `-L HOL-Nitpick` or choose the HOL-Nitpick logic from the Isabelle menu in 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 SAT4J is used as the SAT solver by adding the line

```

nitpick_params [sat_solver = SAT4J]

```

after the **begin** keyword. Being written in Java, SAT4J is the most portable SAT solver supported by Nitpick and the only one that is guaranteed to be available if you followed the instructions in §2. Faster 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 " $P \longrightarrow Q$ "
nitpick

```

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

Nitpick found a genuine counterexample:

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

In cases like this, there is only one thing to say:

```

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

```

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

nitpick [*verbose*]

Trying 6 scopes:

$\text{card } 'a = 1;$
 $\text{card } 'a = 2;$
 $\text{card } 'a = 3;$
 $\text{card } 'a = 4;$
 $\text{card } 'a = 5;$
 $\text{card } 'a = 6.$

Nitpick found a genuine counterexample for $\text{card } 'a = 3$:

Free variables:

$P = \{a_2, a_3\}$
 $x = a_3$

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 “ $P x \implies P (\text{THE } y. P y)$ ”

nitpick [*show_consts*]

Nitpick found a genuine counterexample for $\text{card } 'a = 3$:

Free variables:

$P = \{a_2, a_3\}$

$x = a_3$
Constant:
THE $y. P y = a_1$

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 directly assigned a value to the expression THE $y. P y$ (i.e., *The* $(\lambda y. P y)$), rather than to the *The* constant. If we disable this optimization by using the command

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

we get *The* as expected:

Constant:
The = *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* $(\lambda y. P y) = \textit{The } \{a_2, a_3\} = a_1$, just like before.¹

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.

We can further increase our confidence in the formula by exhausting all cardinalities up to 100:

nitpick [*card* 'a = 1-100]
Nitpick found no counterexample.

Let's see if Sledgehammer [6] can find a proof:

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

¹The *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 $z \# zs$.

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 " $\exists g. \forall x. g (f x) = x \implies \forall y. \exists x. y = f x$ "

nitpick

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

Free variable:

$f = \text{undefined}(b_1 := a_1)$

Skolem constants:

$g = \text{undefined}(a_1 := b_1, a_2 := b_1)$

$y = a_2$

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 genuine counterexample for card 'a = 2:

Skolem constant:

$\lambda x. f = \text{undefined}(a_1 := \text{undefined}(a_1 := a_2, a_2 := a_1),$
 $a_2 := \text{undefined}(a_1 := a_1, a_2 := a_1))$

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 " $\text{refl } r \implies \text{sym } r$ "

nitpick

Nitpick found a genuine counterexample:

Free variable:

$r = \{(a_1, a_1), (a_2, a_1), (a_2, a_2)\}$

Skolem constants:

$$\begin{aligned} \text{sym}.x &= a_2 \\ \text{sym}.y &= a_1 \end{aligned}$$

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

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

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 $\{0, 1, \dots, K - 1\}$ of *nat* (where $K = \text{card } \text{nat}$) and maps all other numbers to the undefined value (\perp). The type *int* is handled in a similar way: If $K = \text{card } \text{int}$, the fragment of *int* known to Nitpick is $\{-\lceil K/2 \rceil + 1, \dots, +\lfloor K/2 \rfloor\}$.

Undefined values lead to a three-valued logic. If the formula evaluates to \perp , Nitpick reports the model as a potential counterexample and continues looking for a model that makes the negated formula evaluate to *True*. For example:

lemma “[$i \leq j$; $n \leq (m::\text{nat})$] $\implies i * n + j * m \leq i * m + j * n$ ”
nitpick

Nitpick found a potential counterexample:

Free variables:

$$\begin{aligned} i &= 1 \\ j &= 1 \\ m &= 1 \\ n &= 1 \end{aligned}$$

Nitpick found a genuine counterexample:

Free variables:

$$\begin{aligned} i &= 0 \\ j &= 1 \\ m &= 1 \\ n &= 0 \end{aligned}$$

Nitpick first finds a potential counterexample for $\text{card } \text{nat} = 2$ (which we can find out by enabling *verbose*); then it finds a genuine one for the same cardinality. The spurious counterexample manifests itself because $1 * 1 + 1 * 1$ equals 2, which lies outside the fragment $\{0, 1\}$: The right-hand side of \implies then evaluates to \perp , and

the left-hand side evaluates to *True*; following Kleene’s three-valued logic, this gives \perp for the entire formula.

One way to carry out such investigations is to tell Nitpick to display the subterms of interest by using the *eval* option:

```
nitpick [eval = "i * n + j * m" "i * m + j * n"]
```

Nitpick found a potential counterexample:

Free variables:

$i = 1$

$j = 1$

$m = 1$

$n = 1$

Evaluated terms:

$i * n + j * m = \perp$

$i * m + j * n = \perp$

Nitpick found a genuine counterexample:

Free variables:

$i = 0$

$j = 1$

$m = 1$

$n = 0$

Evaluated terms:

$i * n + j * m = 0$

$i * m + j * n = 1$

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 "∀n. Suc n > n ⟹ P"
```

```
nitpick [card nat = 100, check_potential]
```

Nitpick found a potential counterexample:

Free variable:

$P = \text{False}$

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 n in $\forall n. \text{Suc } n > n$ ranges over an infinite type. If Nitpick finds an n such that $\text{Suc } n \not> n$, it evaluates the assumption to *False*; but otherwise, it does not know anything about values of $n \geq \text{card nat}$ and must therefore evaluate the assumption to \perp , not *True*. Since the assumption can never be satisfied, the putative lemma can never be falsified.

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.

A final remark before we move on: Because numbers are infinite and are approximated using a three-valued logic, there is no need to systematically enumerate domain sizes. If Nitpick cannot find a genuine counterexample for *card nat* = 10, none could be found for smaller domains. Nitpick nonetheless enumerates all cardinalities from 1 to 6 for *nat*, because smaller cardinalities are fast to handle and give rise to simpler counterexamples.

3.6 Inductive Datatypes

Like natural numbers and integers, inductive datatypes with recursive constructors admit no finite models and must be approximated. Conceptually, each constructor is seen as building a different kind of object, for which you can specify a *multiplicity* (abbreviated *mult*). For example, using a multiplicity of 1 for $[]$ (*Nil*) and 10 for *op #* (*Cons*), Nitpick looks for all counterexamples that can be built using at most 11 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 @ ys) = hd xs*"

nitpick

Nitpick found a potential counterexample for card 'a = 1:

Free variables:

xs = $[a_1]$

ys = $[a_1]$

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

Free variables:

xs = $[]$

ys = $[a_1]$

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

Datatype:

'a list = $\{[], [a_1], \perp\}$

Constants:

hd = *undefined*($[] := a_2, [a_1] := a_1$)

** @ ys* = *undefined*($[] := [a_1], [a_1] := \perp$)

Since *hd []* is undefined in the logic, it may be given any value, including a_1 .

The second constant, $\star @ ys$, is simply the append operator whose second argument is fixed to be ys (i.e., $\lambda xs. xs @ ys$). Appending $[a_1]$ to itself would normally give $[a_1, a_1]$, but this value is not representable in the fragment of $'a$ list considered by Nitpick, which is shown under the “Datatype” heading.

Given $card\ 'a = 2$, $mult\ [] = 1$, and $mult\ op\ \# = 2$, Nitpick considers the following fragments:

$$\begin{array}{lll} \{[], [a_1], [a_2]\}; & \{[], [a_1], [a_1, a_1]\}; & \{[], [a_1], [a_2, a_1]\}; \\ \{[], [a_2], [a_1, a_2]\}; & \{[], [a_2], [a_2, a_2]\}. & \end{array}$$

All subterm-closed fragments consisting of one $[]$ value and two $op\ \#$ values are listed and only those. As an example of a non-subterm-closed fragment, consider $\mathcal{F} = \{[], [a_1], [a_1, a_2]\}$, and observe that $[a_1, a_2]$ (i.e., $a_1 \# [a_2]$) has $[a_2] \notin \mathcal{F}$ as a subterm.

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

lemma “ $[x] = [y]$ ”

nitpick $[show_datatypes, max_potential = 0]$

Nitpick found a genuine counterexample for $card\ 'a = 2$:

Free variables:

$$x = a_2$$

$$y = a_1$$

Datatype:

$$'a\ list = \{[], [a_1], [a_2], \perp\}$$

This time we asked Nitpick not to display any potential counterexamples by specifying $max_potential = 0$.

We saw earlier that type cardinalities range from 1 to 6 by default. Similarly, Nitpick tries multiplicities 1 to 6 for each constructor. This can be changed using the *mult* option, as explained in §5.2. Because datatypes are approximated using a three-valued logic, there is no need to systematically enumerate multiplicities: If Nitpick cannot find a genuine counterexample for $mult\ op\ \# = 10$, none could be found for smaller multiplicities.

Inconsistencies in the multiplicity settings are automatically resolved. For example, setting $mult\ [] = 2$ will result in a multiplicity of 1.

3.7 Typedefs and Records

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 ≡ Abs_three 0"
definition B :: three where "B ≡ Abs_three 1"
definition C :: three where "C ≡ Abs_three 2"

```

```

lemma "[P A; P B] ⇒ P x"
nitpick [show_datatypes]

```

Nitpick found a genuine counterexample:

Free variables:

$P = \{Abs_three\ 0, Abs_three\ 1\}$
 $x = Abs_three\ 2$

Datatypes:

$nat = \{0, 1, 2, \perp\}$
 $three = \{Abs_three\ 0, Abs_three\ 1, Abs_three\ 2, \perp\}$

Records, which are implemented as **typedefs** behind the scenes, are handled in exactly the same way:

```

record point =
  Xcoord :: int
  Ycoord :: int

lemma "Xcoord (|Xcoord = x, Ycoord = y|) = y"
nitpick [max_potential = 0]

```

Nitpick found a genuine counterexample:

Free variables:

$x = 1$
 $y = 0$

3.8 Inductively Defined Predicates

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

Nitpick addresses this problem by first acknowledging that not all inductively defined predicates are equal. Consider the *even* predicate below:

```

inductive even where
  "even 0" |
  "even n ⇒ even (Suc (Suc n))"

```

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

$\dots \Rightarrow even\ k'' \Rightarrow even\ k' \Rightarrow even\ k.$

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

$$\text{even } 0 \implies \text{even } 2 \implies \dots \implies \text{even } (k - 2) \implies \text{even } k.$$

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

Let's try an example:

lemma " $\exists n. \text{even } n \wedge \text{even } (\text{Suc } n)$ "

nitpick [*card nat = 100, verbose, show_consts*]

The inductively defined predicate "even" was proved wellfounded. Nitpick can compute it efficiently.

Trying 1 scope:

card nat = 100.

Nitpick found a potential counterexample for card nat = 100:

Constant:

*even = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,
68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98}*

Nitpick could not find a better counterexample.

We see that Nitpick correctly computed the set of even numbers less than 100. Nitpick cannot rule out the existence of a natural number $n \geq 100$ such that both *even* n and *even* (*Suc* n) are true. To help Nitpick, we could bound the existential quantifier:

lemma " $\exists n \leq 99. \text{even } n \wedge \text{even } (\text{Suc } n)$ "

nitpick [*card nat = 100*]

Nitpick found a genuine counterexample:

Trivial

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]] \implies even' (m + n)"

²If an inductive predicate is wellfounded, 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.

This definition is not wellfounded: 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 " $\neg even' 8$ "

nitpick [$card\ nat = 10, verbose, show_consts$]

The inductively defined predicate " $even'$ " could not be proved wellfounded. Nitpick will unroll it.

Trying 5 scopes:

$card\ nat = 10$ 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 potential counterexample for $card\ nat = 10$ and $iter\ even' = 1$:

Constant:

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

Nitpick found a genuine counterexample for $card\ nat = 10$ and $iter\ even' = 2$:

Constant:

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

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 five scopes specifying different bounds on the numbers of iterations: 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 subscripted question marks ('?'). These are the elements for which the predicate evaluates to \perp . Thus, $even'$ evaluates to either *True* or \perp , never *False*. This reflects the fundamental incompleteness of predicate unrolling.

When unrolling a predicate, Nitpick tries 1, 2, 4, 8, 16, and 32 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.

4 Case Studies

As a didactic device, the previous section focused 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.

Since these examples are larger than the ones considered earlier, more time is spent solving SAT problems. If you haven't done so already, now would be a perfect time to install MiniSat 1.14. To reproduce the results presented in this section, add the command

```
nitpick_params [no_auto, max_potential = 0, sat_solver = MiniSat]
```

at the beginning of your theory.

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:

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

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 lists*. 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

$$\begin{aligned} R1: & \text{“} [] \in S \text{”} \mid \\ R2: & \text{“} w \in A \implies b \# w \in S \text{”} \mid \\ R3: & \text{“} w \in B \implies a \# w \in S \text{”} \mid \\ R4: & \text{“} w \in S \implies a \# w \in A \text{”} \mid \end{aligned}$$

R5: " $w \in S \implies b \# w \in S$ " |
 R6: " $\llbracket v \in B; v \in B \rrbracket \implies a \# v @ w \in B$ "

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 part 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 :

" $w \in S \longrightarrow \text{length } [x \leftarrow w. x = a] = \text{length } [x \leftarrow w. x = b]$ "

nitpick

Nitpick found a genuine 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 $b \# \dots \in S$ 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 genuine 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:

" $\llbracket v \in B; v \in B \rrbracket \implies a \# v @ w \in B$ "

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, even if we enlarge the scope. Let's move on and check completeness:

theorem $S_complete$:

" $\text{length } [x \leftarrow w. x = a] = \text{length } [x \leftarrow w. x = b] \longrightarrow w \in S$ "

nitpick

Nitpick found a genuine 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 introduction rule

$$“\llbracket v \in A; w \in A \rrbracket \implies b \# v @ w \in A”$$

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:*

$$“w \in S \iff \text{length } [x \leftarrow w. x = a] = \text{length } [x \leftarrow w. x = b]”$$

$$“w \in A \iff \text{length } [x \leftarrow w. x = a] = \text{length } [x \leftarrow w. x = b] + 1”$$

$$“w \in B \iff \text{length } [x \leftarrow w. x = b] = \text{length } [x \leftarrow w. x = a] + 1”$$

nitpick [*timeout* = 60 s]

Nitpick found no counterexample.

The *timeout* option is given to override Nitpick's default timeout delay, which is set to 30 seconds.

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* = $\Lambda \mid N$ “*a::linorder*” *nat* “*'a tree*” “*'a tree*”

primrec *data* **where**

“*data* $\Lambda = \text{undefined}$ ” \mid

“*data* $(N\ x\ _\ _) = x$ ”

primrec *dataset* **where**

“*dataset* $\Lambda = \{\}$ ” \mid

“*dataset* $(N\ x\ t\ u) = \{x\} \cup \text{dataset } t \cup \text{dataset } u$ ”

primrec *level* **where**

“*level* $\Lambda = 0$ ” \mid

“*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 $\Lambda = \text{True}$ " |

"wf ($N_k t u$) =

(if $t = \Lambda$ then

$k = 1 \wedge (u = \Lambda \vee (\text{level } u = 1 \wedge \text{left } u = \Lambda \wedge \text{right } u = \Lambda))$)

else

$\text{wf } t \wedge \text{wf } u \wedge u \neq \Lambda \wedge \text{level } t < k \wedge \text{level } u \leq k \wedge \text{level } (\text{right } u) < k$)"

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 $t \neq \Lambda \wedge k = \text{level } t$ then

$N (\text{data } t) k (\text{left } t) (N x k (\text{right } t) u)$

else

$N x k t u$)"

primrec split where

"split $\Lambda = \Lambda$ " |

"split ($N x k t u$) =

(if $u \neq \Lambda \wedge k = \text{level } (\text{right } u)$ then

$N (\text{data } u) (\text{Suc } k) (N x k t (\text{left } u)) (\text{right } u)$

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.

Furthermore, applying *skew* or *split* to a wellformed tree should not alter the tree:

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

Nitpick ran out of time.

Insertion is implemented recursively. It preserves the sort order:

primrec *insort* **where**
 $"insort\ \Lambda\ x = N\ x\ 1\ \Lambda\ \Lambda"$ |
 $"insort\ (N\ y\ k\ t\ u)\ x =$
 $(*)\ (split\ \circ\ skew)\ (*)\ (N\ y\ k\ (if\ x < y\ then\ insort\ t\ x\ else\ t)$
 $(if\ x > y\ then\ insort\ u\ x\ else\ u))"$

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

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

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

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

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*. This gives

theorem *wf_insort_nat*: $"wf\ t \implies wf\ (insort\ t\ (x::nat))"$
nitpick [*eval* = $"insort\ t\ x"$]

Nitpick found a genuine counterexample:

Free variables:
 $t = N\ 2\ 1\ \Lambda\ \Lambda$

$x = 1$
Evaluated term:
 $\text{insort } t \ x = N \ 2 \ 1 \ (N \ 1 \ 1 \ \wedge \ \wedge) \ \wedge$

Nitpick’s output reveals that the element 1 was added as a left child of 2, and both with 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*: “ $\text{wf } t \implies \text{wf } (\text{insort } t \ x)$ ”
nitpick

Nitpick ran out of time.

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

theorem *dataset_insort*: “ $\text{dataset } (\text{insort } t \ x) = \{x\} \cup \text{dataset } t$ ”
nitpick

Nitpick found no counterexample.

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 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:

- $\langle \textit{string} \rangle$: A string.
- $\langle \textit{int} \rangle$: An integer. Negative integers are prefixed with a hyphen.
- $\langle \textit{nat_range} \rangle$: A nonnegative integer (e.g., 3) or a hyphenated range of non-negative integers (e.g., 1–4).

- $\langle \text{nat_seq} \rangle$: A comma-separated sequence of nonnegative integer ranges (e.g., 1,3,6–8).
- $\langle \text{time} \rangle$: An integer followed by *s* (seconds) or *ms* (milliseconds), or the keyword *none* (∞ years).
- $\langle \text{const} \rangle$: The name of a HOL constant.
- $\langle \text{term_list} \rangle$: A space-separated list of HOL terms (e.g., “*f x*” “*g y*”).
- $\langle \text{type} \rangle$: A HOL type.

Default values are indicated in square brackets.

5.1 Mode of Operation

auto / no_auto [no_auto]

Specifies whether Nitpick should be run automatically on newly entered theorems. For automatic runs, *blocking* (§5.1), *verbose* (§5.3), and *debug* (§5.3) are implicitly 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 / non_blocking [blocking]

Specifies whether the **nitpick** command should block or not. 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.

falsify / satisfy [falsify]

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

5.2 Scope of Search

card $\langle \text{type} \rangle = \langle \text{nat_seq} \rangle$

Specifies the sequence of cardinalities to use for a given type, which may be a free type (*'a*, *'b*, etc.), a type declared using **typedcl** (except *bool*), or the built-in types *nat* or *int*. For *nat* and *int*, the cardinality applies to the fragment used to approximate the type. For example:

<i>card nat</i> = 4	induces the fragment	$\{0, 1, 2, 3\}$
<i>card int</i> = 4	induces the fragment	$\{-1, 0, +1, +2\}$
<i>card int</i> = 5	induces the fragment	$\{-2, -1, 0, +1, +2\}$.

In general:

$card\ nat = K$ induces the fragment $\{0, \dots, K - 1\}$
 $card\ int = K$ induces the fragment $\{-\lceil K/2 \rceil + 1, \dots, +\lfloor K/2 \rfloor\}$.

For free types, and often also for **typed**decl'd types, it usually makes sense to specify cardinalities as a range of the form $1-n$.

card = $\langle nat_seq \rangle$ [1–6]

Specifies the default sequence of cardinalities to use for free types, **typed**decl'd types (except *bool*), and the built-in types *nat* and *int*. This can be overridden on a per-type basis using the *card* $\langle type \rangle$ option described above.

mult $\langle const \rangle = \langle nat_seq \rangle$

Specifies the sequence of multiplicities to use for a given datatype constructor. The multiplicity of a constructor is the number of distinct values that it can construct. Nonsensical values (e.g., *mult* [] = 2) are repaired behind the scenes. Most types declared using **typed**def are considered datatypes, whose single constructor is the corresponding *Abs_* function. The only exceptions are *nat*, *int*, \times (product), and $+$ (sum); the first two can be assigned a cardinality using *card*, and the last two automatically get a cardinality based on their arguments.

mult = $\langle nat_seq \rangle$ [1–6]

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

iter $\langle const \rangle = \langle nat_seq \rangle$

Specifies the sequence of iteration counts to use when unrolling a given inductive predicate. By default, unrolling takes place for inductive predicates that cannot be proved to be wellfounded, but this behavior is influenced by the *inductive_mood* (§5.5) option. Internally, the iteration count is bounded by the cardinality of the predicate's domain.

iter = $\langle nat_seq \rangle$ [1,2,4,8,16,32]

Specifies the sequence of iteration counts to use when unrolling inductive predicates. This can be overridden on a per-predicate basis using the *iter* $\langle const \rangle$ option above.

lockstep / no_lockstep [no_lockstep]

Specifies whether cardinalities of different types progress together or not. Enabling this option can dramatically reduce the number of scopes tried, but it also diminishes the chances of finding a counterexample. Use with care.

5.3 Output Format

verbose / quiet [quiet]

Specifies whether Nitpick should explain what it does. This option is useful to determine which scopes are tried or which SAT solver is used. The output is clearer if *batch_size* (§5.5) is set to 1.

See also *card* (§5.2), *mult* (§5.2), and *iter* (§5.2).

debug / no_debug [no_debug]

Specifies whether Nitpick should display additional debugging information.

show_skolems / hide_skolem [show_skolems]

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 / hide_datatypes [hide_datatypes]

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

show_consts / hide_consts [hide_consts]

Specifies whether the values of constants associated with the original formula (including its axioms) should be displayed as part of counterexamples. These values are sometimes helpful when investigating whether a potential counterexample is genuine or spurious, but they can clutter the output.

max_potential = <int> [1]

Specifies the maximum number of potential counterexamples to display. Setting this option to 0 speeds up the search for a genuine counterexample.

eval = <term_list>

Specifies the list of terms whose values should be displayed as part of 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 = <string>

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

- *genuine*: Nitpick found a 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 Automatic Counterexample Checks

check_potential / trust_potential [*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 / trust_genuine [*trust_genuine*]

Specifies whether 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 therefore easy to install. 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.
- **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 [3]. Alternatively, you can install a standard version of zChaff and set the Isabelle 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, you must ensure that the Isabelle environment variable `RSAT_HOME` is set 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 with Nitpick, you must ensure that the Isabelle environment variable `BERKMIN_HOME` is set to the directory that contains the BerkMin561 executable. 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 with Nitpick, you must ensure that the Isabelle environment variable `BERKMINALLOY_HOME` is set to the directory that contains the `berkmin` executable.
- **MiniSat:** MiniSat is a minimalistic yet 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 [3]. Alternatively, you can install a standard version of MiniSat and set the Isabelle 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 2.0 beta (2007-07-21).
- **Jerusat:** Jerusat 1.3 is an efficient solver written in C. To use Jerusat with Nitpick, you must ensure that the Isabelle environment variable `JERUSAT_HOME` is set to the directory that contains the `Jerusat1.3` executable. The C sources for Jerusat are available at <http://www.cs.tau.ac.il/~ale1/Jerusat1.3.tgz>.
- **smart:** If `sat_solver` is set to *smart*, Nitpick selects the first solver among zChaff, RSat, BerkMin, BerkMinAlloy, MiniSat, and Jerusat that is configured in Isabelle (i.e., whose `_HOME` variable is set). If none is configured, it falls back on SAT4J, which should always be available. If *verbose* is enabled, Nitpick displays which SAT solver was chosen.

batch_size = $\langle int \rangle$ [100]

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.

See also *verbose* (§5.3).

normalize / dont_normalize [*dont_normalize*]

Specifies whether the formula should be normalized-by-evaluation (as done by Isabelle’s **normal_form** command) before Nitpick processes it further. Normalization often transforms formulas in ways that make the search more

efficient; for example, it simplifies $\text{map } f [x]$ to $[f x]$. On the other hand, it sometimes eliminates free variables and renames bound variables, making the resulting counterexample more cryptic.

***inductive_mood* = $\langle \text{string} \rangle$ [realistic]**

Specifies how to handle inductively defined predicates (and sets). The possible values are listed below:

- ***realistic***: Try to prove that the inductive predicate is wellfounded using Isabelle's *lexicographic_order* and *sizechange* tactics. If this succeeds, use an efficient fixed point equation as specification of the predicate; otherwise, unroll the predicates according to the *iter* $\langle \text{const} \rangle$ and *iter* options (§5.2).
- ***optimistic***: Tentatively treat all inductive predicates as if they were wellfounded. This may lead to spurious counterexamples, but Kodkod is then invoked to check whether the example is genuine or not. This approach is typically the most efficient, but it is also theoretically the weakest. When Kodkod detects potentially non-wellfounded predicates, it emits a message listing the offending predicates and suggesting to try again with *inductive_mood* set to *realistic* or *pessimistic*.
- ***pessimistic***: Treat all inductive predicates as if they were not wellfounded. The predicates are then unrolled according to the *iter* $\langle \text{const} \rangle$ and *iter* options (§5.2).

See also *tac_timeout* (§5.6).

***special_depth* = $\langle \text{int} \rangle$ [4]**

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 n th-level axioms are specialized. This optimization often reduces the search space drastically, especially for higher-order functions.

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

***skolem_depth* = $\langle \text{int} \rangle$ [4]**

Specifies the maximum depth at which skolemization takes 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).

peephole_optim / no_peephole_optim [peephole_optim]

Specifies whether Nitpick should perform peephole optimization of the generated Kodkod formulas. 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 = $\langle int \rangle$ [20]

Specifies an upper bound on the number of relations for which Kodkod generates symmetry breaking predicates. According to the Kodkod documentation [4], “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 = $\langle int \rangle$ [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 / dont_flatten_props [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 = $\langle int \rangle$ [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 = $\langle time \rangle$ [30 s]

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

auto_timeout = $\langle time \rangle$ [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 = $\langle time \rangle$ [200 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 an inductive predicate is well-founded. Nitpick tries to honor this constraint as well as it can but offers no guarantees.

See also *check_potential* (§5.4), *check_genuine* (§5.4), and *inductive_mood* (§5.5).

6 Known Bugs and Limitations

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

- Nitpick assumes that the axioms are consistent and might produce spurious results if this is not the case.
- For functions defined using **primrec**, **fun**, or **function**, Nitpick sometimes generates spurious “genuine” counterexamples about 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 genuine counterexample for card nat = 2:
```

```
  Trivial
```

```
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.

- Coinduction is handled by expanding the greatest fixed point definition, which slows down Kodkod horribly. Future versions of Nitpick are expected to unroll coinductive definitions.

Comments and bug reports concerning Nitpick or this manual should be directed to blanchette@in.tum.de.

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 (2007)
- [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] Kodkod: Constraint Solver for Relational Logic, <http://alloy.mit.edu/kodkod/>
- [4] Kodkod API: Class Options, <http://alloy.mit.edu/kodkod/docs/kodkod/engine/config/Options.html>
- [5] Nipkow, T., Paulson, L. C., Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS vol. 2283. Springer, Heidelberg (2002). Updated version <http://isabelle.in.tum.de/doc/tutorial.pdf> (2009)
- [6] The Sledgehammer: Let Automatic Theorem Provers Write Your Isabelle Scripts, <http://www.cl.cam.ac.uk/research/hvg/Isabelle/sledgehammer.html>
- [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] Wikipedia: AA Tree, http://en.wikipedia.org/wiki/AA_tree