Clubbing Cods
A User's Guide to Kodkodi 1.2.1

Jasmin Christian Blanchette
Fakultät für Informatik, Technische Universität München

September 25, 2009

Contents

1 Introduction

Kodkodi is a front-end for the Java library Kodkod [3], a highly-optimized relational model finder developed by the Software Design Group at MIT. Kodkod is based on SAT solving and supports first-order logic with relations, transitive closure, and partial instances. Kodkod forms the basis of version 4 of the Alloy Analyzer [2]. The Kodkodi front-end is designed to make the Kodkod library available to other programming languages than Java.

This manual explains the concrete syntax supported by Kodkodi. It also explains how to install the tool on your workstation. If you use Kodkodi in conjunction with Nitpick for Isabelle/HOL [6], read the installation instructions in the Nitpick manual [1].

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

2 Installing the Tool

To install Kodkodi, download and extract the archive `http://isabelle.in.tum.de/~blanchet/kodkodi-1.2.1.tgz`. The `.jar` files for Kodkodi, the Kodkod library, the portable SAT4J solver, and the ANTLR 3.1.1 runtimes are located in the `jar` subdirectory. Kodkodi requires a Java 1.5 virtual machine, normally called `java`. To run Kodkodi, you must add this directory to the Java classpath and execute

```
java de.tum.in.isabelle.Kodkodi.Kodkodi
```

To recompile Kodkodi, you need a Java compiler and the ANTLR 3.1.1 parser generator tools.

For better performance, it is recommended to install a C or C++ SAT solver. Follow the instructions on Kodkod's home page [4] to install SAT solvers integrated using the JNI, or install a command-line solver of your choice and specify it using the `External` or `ExternalV2` described in §4.2.3. For convenience, Kodkodi includes the C solver PicoSAT [7] in `external/picosat-913`. Run the `build` script to build it. This requires a C compiler. The executable is installed in `bin`.

3 First Steps

Kodkodi takes its input from standard input and writes its output to standard output (on success) or standard error (on failure). Examples are provided in the `examples` directory. When invoked with wrong command-line arguments, Kodkodi displays the usage text:

```
Usage:  java de.tum.in.isabelle.Kodkodi.Kodkodi [options]
options:
  -help               Show usage and exit
  -verbose            Produce more output
  -exit-on-success    Exit on the first successful "solve" directive
  -clean-up-inst      Remove trivial parts of instance from output
  -max-msecs <num>    Maximum running time in milliseconds
  -max-threads <num>  Maximum number of simultaneous threads
                      (default: ⟨machine-dependent-value⟩)
  -server             Run as TCP server
  -port <number>      Listen to specified port (default: 9128)
```

Example input files are provided in the `examples` directory:

```
$ de.tum.in.isabelle.Kodkodi.Kodkodi < examples/pigeonhole.kki
*** PROBLEM 1 ***

--OUTCOME--
UNSATISFIABLE

--STATS--
p cnf 54 68
primary variables: 6
parsing time: 65 ms
translation time: 92 ms
solving time: 0 ms
```

The `picosat.kki` example shows how to use the efficient PicoSAT solver.

4 Input Format

Kodkodi's input format is modeled after the output format of the `toString()` implementations found in the Kodkod library. The operator that are available in Alloy 4 are given the same precedences as they have there.

The grammar is expressed using a variant of Extended Backus-Naur Form. The actual grammar used by Kodkodi is written using ANTLR and can be found in the file `src/Kodkodi.g`.

4.1 Lexical Issues

The grammar is based on the following lexical units, or tokens:

$$
\begin{aligned}
\mathit{WHITESPACE} &::= (\text{␣} \mid \mathbf{\backslash n} \mid \mathbf{\backslash r} \mid \mathbf{\backslash t} \mid \mathbf{\backslash v})^{+} \\
\mathit{COMMENT} &::= \texttt{//} \sim(\mathbf{\backslash n})^{*}\ (\mathbf{\backslash n} \mid \mathbf{eof}) \\
\mathit{NUM} &::= [\texttt{+} \mid \texttt{-}]\ (\texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9})^{+} \\
\mathit{STR\_LITERAL} &::= \texttt{"} \sim(\texttt{"} \mid \mathbf{\backslash n})^{*}\ \texttt{"} \\
\mathit{ATOM\_NAME} &::= \texttt{A}\ \mathit{NAT} \\
\mathit{UNIV\_NAME} &::= \texttt{u}\ \mathit{NAT} \\
\mathit{OFF\_UNIV\_NAME} &::= \texttt{u}\ \mathit{NAT}\ \texttt{@}\ \mathit{NAT} \\
\mathit{TUPLE\_NAME} &::= (\texttt{P} \mid \texttt{T}\ \mathit{NAT}\ \texttt{\_})\ \mathit{NAT} \\
\mathit{RELATION\_NAME} &::= (\texttt{s} \mid \texttt{r} \mid \texttt{m}\ \mathit{NAT}\ \texttt{\_})\ \mathit{NAT} \\
\mathit{VARIABLE\_NAME} &::= (\texttt{S} \mid \texttt{R} \mid \texttt{M}\ \mathit{NAT}\ \texttt{\_})\ \mathit{NAT}\ \texttt{'}^{?} \\
\mathit{TUPLE\_REG} &::= \texttt{\$}\ (\texttt{A} \mid \texttt{P} \mid \texttt{T}\ \mathit{NAT}\ \texttt{\_})\ \mathit{NAT} \\
\mathit{TUPLE\_SET\_REG} &::= \texttt{\$}\ (\texttt{a} \mid \texttt{p} \mid \texttt{t}\ \mathit{NAT}\ \texttt{\_})\ \mathit{NAT} \\
\mathit{FORMULA\_REG} &::= \texttt{\$f}\ \mathit{NAT} \\
\mathit{REL\_EXPR\_REG} &::= \texttt{\$e}\ \mathit{NAT} \\
\mathit{INT\_EXPR\_REG} &::= \texttt{\$i}\ \mathit{NAT}
\end{aligned}
$$

NAT abbreviates $\texttt{0} \mid (\texttt{1} \mid \ldots \mid \texttt{9})\ (\texttt{0} \mid \ldots \mid \texttt{9})^{*}$.

Whitespace and comments are ignored, except as token separators. In addition to the tokens listed above, various keywords and operators are recognized as tokens. These are shown in bold in the grammar.

The table below describes the lexical conventions adopted for naming atoms, tuples, relations, variables, and registers.

Token Name	Syntax	Description
ATOM_NAME	Aj	Atom at index j in the universe
UNIV_NAME	un	Set of atoms $\{\texttt{A0}, \ldots, \texttt{A}(n-1)\}$
OFF_UNIV_NAME	un@j	Set of atoms $\{\texttt{A}j, \ldots, \texttt{A}(j+n-1)\}$
TUPLE_NAME	Pj	Pair at index j in the pair space associated with the universe
	Tn_j	n-tuple at index j in the n-tuple space associated with the universe ($n \geq 3$)
RELATION_NAME	sj	Set number j
	rj	Binary relation number j
	mn_j	n-ary multirelation number j ($n \geq 3$)
VARIABLE_NAME	Sj	Unprimed set variable number j

	`S`*j*`'`	Primed set variable number *j*
	`R`*j*	Unprimed binary relation variable number *j*
	`R`*j*`'`	Primed binary relation variable number *j*
	`M`*n*`_`*j*	Unprimed *n*-ary multirelation variable number *j* ($n \geq 3$)
	`M`*n*`_`*j*`'`	Primed *n*-ary multirelation variable number *j* ($n \geq 3$)
TUPLE_REG	`$A`*j*	One-tuple register number *j*
	`$P`*j*	Pair register number *j*
	`$T`*n*`_`*j*	*n*-tuple register number *j* ($n \geq 3$)
TUPLE_SET_REG	`$a`*j*	One-tuple set register number *j*
	`$p`*j*	Pair set register number *j*
	`$t`*n*`_`*j*	*n*-tuple set register number *j* ($n \geq 3$)
FORMULA_REG	`$f`*j*	Formula register number *j*
REL_EXPR_REG	`$e`*j*	Relational expression register number *j*
INT_EXPR_REG	`$i`*j*	Integer expression register number *j*

4.2 Overall Structure

This section presents the overall structure of Kodkodi input files.

4.2.1 Problems

$$\textit{problems} ::= \textit{problem}^*$$

Kodkodi takes a list of "problems" as input.

4.2.2 Problem

$$\begin{aligned}\textit{problem} ::= {} & \textit{option}^* \ \textit{univ\_spec} \ \textit{tuple\_reg\_directive}^* \ \textit{bound\_spec}^* \ \textit{int\_bound\_spec}^? \\ & \textit{expr\_reg\_directive}^* \ \textit{solve\_directive}\end{aligned}$$

A problem consists of three main parts: a universe specification, a set of bound specifications, and a Kodkod formula to satisfy supplied in a "solve" directive.

Example:

```
univ: u1
bounds s0: {A0}
solve all [S0 : one s0, S1 : one s0] | S0 = S1;
```

4.2.3 Kodkod Options

$$
\begin{aligned}
\textit{option} ::= \ & \texttt{solver} : \textit{STR\_LITERAL}\,(\,\texttt{,}\ \textit{STR\_LITERAL})^* \ \mid \\
& \texttt{symmetry\_breaking} : \textit{NUM} \ \mid \\
& \texttt{sharing} : \textit{NUM} \ \mid \\
& \texttt{bit\_width} : \textit{NUM} \ \mid \\
& \texttt{skolem\_depth} : \textit{NUM} \ \mid \\
& \texttt{flatten} : (\texttt{true} \mid \texttt{false}) \ \mid \\
& \texttt{delay} : \textit{NUM}
\end{aligned}
$$

Kodkod supports various options, documented in the `kodkod.engine.config.Options` class [5]. The following solvers are supported:

```
solver: "DefaultSAT4J"
solver: "LightSAT4J"
solver: "ZChaff"
solver: "zChaff"
solver: "ZChaffMincost"
solver: "zChaffMincost"
solver: "MiniSatProver"
solver: "MiniSat"
solver: "SAT4J" "instance"
solver: "External" "executable" "temp_input" "temp_output"
        "arg_1" ... "arg_n"
solver: "ExternalV2" "executable" "temp_input" "temp_output"
        "sat_marker" "var_marker" "unsat_marker" "arg_1" ... "arg_n"
```

For `"External"`, the optional arguments `"arg_1"`, ..., `"arg_n"` are passed before the input file name. For `"ExternalV2"`, they are passed after.

The `delay` option specifies a delay (expressed in milliseconds) between solving a problem and exiting, if the `-exit-on-success` command-line option is specified. This can be used to grant additional time to other threads so that they have a chance to finish.

4.2.4 Universe Specification

$$
\textit{univ\_spec} ::= \texttt{univ} : \textit{UNIV\_NAME}
$$

The universe specification fixes the universe's uninterpreted atoms. Kodkodi requires that the atoms are numbered consecutively from `A0` to `A`$(n-1)$.

Examples:

```
univ: u2
univ: u100
```

4.2.5 Relation Bound Specifications

bound_spec ::= `bounds` *RELATION_NAME* (`,` *RELATION_NAME*) `:`
(*tuple_set* | `[` *tuple_set* `,` *tuple_set* `]`)

A relational bound specification gives a lower and an upper bound for the given relations. If only one bound is specified, it is taken as both lower and upper bound. The lower bound must be a subset of the upper bound.

Examples:

```
bounds s0: {A0}
bounds r2: [{}, {A0 .. A9} -> {A10 .. A19}]
```

4.2.6 Integer Bound Specification

int_bound_spec ::= `int_bounds` `:` *int_bound_seq* (`,` *int_bound_seq*)*
int_bound_seq ::= [*NUM* `:`] `[` *tuple_set* (`,` *tuple_set*)* `]`

An integer bound specification establishes a correspondence between integers and sets of atoms that represent that integer in relational expressions. The syntax makes it possible to specify the bounds of consecutive integers in sequence.

Example:

```
int_bounds: [{A0}, {A1}], 10: [{A2}, {A3}, {A4}]
```

In the above example, `0` is bounded by `{A0}`, `1` is bounded by `{A1}`, `10` is bounded by `{A2}`, `11` is bounded by `{A11}`, and `12` is bounded by `{A4}`.

4.2.7 Solve Directive

solve_directive ::= `solve` *formula* `;`

The "solve" directive tells Kodkod to try to satisfy the given formula.

Example:

```
solve all [S0 : one s0, S1 : one s0] | ! S0 = S1 => no S0.r0 & S1.r0
```

4.3 Register Directives

Registers make it possible to use a complex syntactic construct several times without duplicating it. They also help reduce Kodkod's memory usage and running time.

4.3.1 Tuple Register Directives

tuple_reg_directive ::= *TUPLE_REG* `:=` *tuple* |
TUPLE_SET_REG `:=` *tuple_set*

A tuple register directive assigns a value to a tuple or tuple set register.

Examples:

```
$P0 := [A0, A0]
$P1 := [A1, A1]
$t4_0 := {$P0, $P1} -> {$P0, $P1}
```

4.3.2 Expression Register Directives

expr_reg_directive ::= *FORMULA_REG* `:=` *formula* |
REL_EXPR_REG `:=` *rel_expr* |
INT_EXPR_REG `:=` *int_expr*

Formulas, relational expressions, and integer expressions can also be assigned to registers using an expression register directive. An alternative is to use the `let` binder inside an expression.

Examples:

```
$f0 := all [S0 : one s0] | s0 in univ
$e5 := (s0 & s1).r1 + (s0 & s2).r2
$i14 := 2 * #($e5) + 1
```

4.4 Tuple Language

Kodkod supports partial solutions in the form of bounds on relations. The bound specifications involve tuples and tuple sets.

4.4.1 Tuples

tuple ::= `[` *ATOM_NAME* (`,` *ATOM_NAME*)* `]` |
ATOM_NAME |
TUPLE_NAME |
TUPLE_REG

An n-tuple is normally specified using the syntax `[A`j_1`, ..., A`j_n`]`. The brackets are optional when $n = 1$. Alternatively, tuples can be specified using an index in

the n-tuple space. For example, given the universe `u10`, the name `P27` refers to the pair `[A2, A7]`.

Examples:

```
[A0, A1, A5, A20]
A0
P5
$P14
```

4.4.2 Tuple Sets

$$
\begin{aligned}
\mathit{tuple\_set} ::= {} & \mathit{tuple\_set}\ (\texttt{+} \mid \texttt{-})\ \mathit{tuple\_set} \mid \\
& \mathit{tuple\_set}\ \texttt{\&}\ \mathit{tuple\_set} \mid \\
& \mathit{tuple\_set}\ \texttt{->}\ \mathit{tuple\_set} \mid \\
& \mathit{tuple\_set}\ \texttt{[}\ \mathit{NUM}\ \texttt{]} \mid \\
& \texttt{\{}\ \mathit{tuple}\ (\texttt{,}\ \mathit{tuple})^{*}\ \texttt{\}} \mid \\
& \texttt{\{}\ \mathit{tuple}\ \texttt{..}\ \mathit{tuple}\ \texttt{\}} \mid \\
& \texttt{\{}\ \mathit{tuple}\ \texttt{\#}\ \mathit{tuple}\ \texttt{\}} \mid \\
& \texttt{none} \mid \\
& \texttt{all} \mid \\
& \mathit{UNIV\_NAME} \mid \\
& \mathit{OFF\_UNIV\_NAME} \mid \\
& \mathit{TUPLE\_SET\_REG} \mid \\
& \texttt{(}\ \mathit{tuple\_set}\ \texttt{)}
\end{aligned}
$$

Tuple sets can be constructed in several ways. The `+`, `-`, and `&` operators denote the union, difference, and intersection of two tuple sets, respectively. The `->` operator denotes the Cartesian product of two tuple sets. The `[]` operator projects the tuple set onto the given dimension. Tuple sets can be specified exhaustively by listing all their tuples. If all the tuples have consecutive indices, the range operator `..` can be used. Alternatively, if all the tuples occupy a rectangular, cubic, etc., area in the tuple space, they can be specified by passing the lowest and highest corner of the area to the `#` operator. Finally, `none` is a synonym for `{}`, and `all` denotes the complete tuple set (whose arity is deduced from the context).

Examples:

```
{A1}
{A1, A2} -> {A3, A4}
{[A1, A2] ..  [A3, A4]}
{[A1, A2] # [A3, A4]}
$p14
```

4.4.3 Tuple Set Operator Precedences and Associativities

The operator precedences and associativities are given in the table below. Fully bracketed operators are not listed.

Level	Operator Class	Arity	Associativity
1	`+ -`	Binary	Left-associative
2	`&`	Binary	Associative
3	`->`	Binary	Associative
4	`[]`	Binary	Left-associative

4.5 Expression Language

Kodkod supports three types of expression: Boolean expressions (formulas), relational expressions, and integer expressions.

4.5.1 Formulas

formula ::= (`all` | `some`) *decls* `|` *formula* |
`let` *assigns* `|` *formula* |
`if` *formula* `then` *formula* `else` *formula* |
formula `||` *formula* |
formula `<=>` *formula* |
formula `=>` *formula* |
formula `&&` *formula* |
`!` *formula* |
`ACYCLIC (` *RELATION_NAME* `)` |
`FUNCTION (` *RELATION_NAME* `,` *rel_expr* `->` (`one` | `lone`) *rel_expr* `)` |
`TOTAL_ORDERING (` *RELATION_NAME* `,`
(*UNIV_NAME* | *OFF_UNIV_NAME* | *RELATION_NAME*) `,`
(*ATOM_NAME* | *RELATION_NAME*) `,`
(*ATOM_NAME* | *RELATION_NAME*) `)` |
rel_expr (`in` | `=`) *rel_expr* |
int_expr (`=` | `<` | `<=` | `>` | `>=`) *int_expr* |
(`no` | `lone` | `one` | `some`) *rel_expr* |
`false` |
`true` |
FORMULA_REG |
(*formula*)

A formula, or Boolean expression, specifies a constraint involving relations and integers.

Example:

```
some [S0 : some s0] | if S0 in s1 then !$f1 else $i0 <= $i1
```

4.5.2 Relational Expressions

rel_expr ::= `let` *assigns* `|` *rel_expr* |
`if` *formula* `then` *rel_expr* `else` *rel_expr* |
rel_expr (`+` | `-`) *rel_expr* |
rel_expr `++` *rel_expr* |
rel_expr `&` *rel_expr* |
rel_expr `->` *rel_expr* |
rel_expr `\` *rel_expr* |
rel_expr `(` *rel_expr* (`,`*rel_expr*)* `)` |
rel_expr `[` *int_expr* (`,` *int_expr*)* `]` |
rel_expr `.` *rel_expr* |
(`^` | $*$ | `~`) *rel_expr* |
`{` *decls* `|` *formula* `}` |
(`Bits` | `Int`) `[` *int_expr* `]` |
`iden` |
`ints` |
`none` |
`univ` |
ATOM_NAME |
UNIV_NAME |
OFF_UNIV_NAME |
RELATION_NAME |
VARIABLE_NAME |
REL_EXPR_REG |
`(` *rel_expr* `)`

A relational expression denotes a relation (set, binary relation, or multirelation). Nearly all operators are identical to those offered by Kodkod, which in turn are modeled after those provided by Alloy. Notable exceptions are the conditional expression `if` ... `then` ... `else` ...; the r `\` s operator, which is a shorthand for `if no` r `then` s `else` r; and finally r`(`s_1`, ..., `s_n`)`, which is equivalent to s_n`.(...(`s_1`.r)...)`.

Example:

```
if #(s0) > 5 then s0.r0 + s1.r1 else none
```

4.5.3 Integer Expressions

int_expr ::= `sum` *decls* `|` *int_expr* |
`let` *assigns* `|` *int_expr* |
`if` *formula* `then` *int_expr* `else` *int_expr* |
int_expr (`<<` | `>>` | `>>>`) *int_expr* |
int_expr (`+` | `-`) *int_expr* |
int_expr (`*` | `/` | `%`) *int_expr* |
(`#` | `sum`) `(` *rel_expr*`)` |
int_expr `|` *int_expr* |
int_expr `^` *int_expr* |
int_expr `&` *int_expr* |
(`~` | `-` | `abs` | `sgn`) *int_expr* |
NUM |
INT_EXPR_REG |
`(` *int_expr* `)`

An integer expression denotes an integer.

Example:

```
(sum [S0 : one s0] | #(S0) * (#(S0) + 1) / 2) % 10
```

4.5.4 Declarations

decls ::= `[` *decl* (`,` *decl*)* `]`
decl ::= *VARIABLE_NAME* `:` (`no` | `lone` | `one` | `some` | `set`) *rel_expr*

The `all`, `some`, and `sum` quantifiers take a list of variable declarations.

Example:

```
[S0 : set s0, S1 : one s1]
```

4.5.5 Assignments

assigns ::= `[` *assign* (`,` *assign*)* `]`
assign ::= *FORMULA_REG* `:=` *formula* |
REL_EXPR_REG `:=` *rel_expr* |
INT_EXPR_REG `:=` *int_expr*

The `let` binder takes a list of register assignments.

Example:

```
[$f0 := all [S0 : one s0] | s0 in univ, $i14 := 2 * #($e5) + 1]
```

4.5.6 Operator Precedences and Associativities

The operator precedences and associativities are given in the table below. Fully bracketed operators are not listed.

Level	Operator Class	Arity	Associativity
1	`all | some | sum | let | if then else`	Binary/Ternary	Right-associative
2	`||`	Binary	Associative
3	`<=>`	Binary	Associative
4	`=>`	Binary	Right-associative
5	`&&`	Binary	Associative
6	`!`	Unary	N/A
7	`in = < <= > >=`	Binary	N/A
8	`no lone one some`	Unary	N/A
9	`<< >> >>>`	Binary	Left-associative
10	`+ -`	Binary	Left-associative
11	`* / %`	Binary	Left-associative
12	`++`	Binary	Associative
13	`| ^ &`	Binary	Associative
14	`->`	Binary	Associative
15	`\`	Binary	Associative
16	`(,)`	Binary	Left-associative
17	`[,]`	Binary	Left-associative
18	`.`	Binary	Left-associative
19	`^ * ~ - abs sgn`	Unary	N/A

4.6 Comments

Comments may be specified as in C++, that is, either as a one line comment starting with `//` or as a block starting with `/*` and ending with `*/`.

Examples:

```
/*
  Copyright 2009 Gnomovision, Inc.
*/

univ: u99999 // Don't panic!
```

5 Case Study: Sorting Using Alloy and Kodkodi

Although Kodkodi's syntax is similar to Alloy's, there are a few important conceptual differences. Consider the following Alloy specification of integer sorting:

```
abstract sig IntSeq {
  ints : seq Int
}

pred isSorted [s : IntSeq] {
  all i : s.ints.inds - s.ints.lastIdx | s.ints[i] <= s.ints[i + 1]
}
pred isPermutation [pre, post : IntSeq] {
  all p : Int | #{pre.ints.p} = #{post.ints.p}
}

one sig Pre extends IntSeq {}
one sig Post extends IntSeq {}

fact {
  Pre.ints[0] = 7 && Pre.ints[1] = 2 &&
  Pre.ints[2] = 4 && Pre.ints[3] = 3 &&
  Pre.ints[4] = 3 && Pre.ints[5] = 8 &&
  Pre.ints[6] = 5 && Pre.ints[7] = 20 &&
  Pre.ints[8] = 18 && Pre.ints[9] = 1 &&
  Pre.ints[10] = 10 && Pre.ints[11] = 5 &&
  Pre.ints[12] = 7 && Pre.ints[13] = 12 &&
  Pre.ints[14] = 2 && Pre.ints[15] = 19 &&
  Pre.ints[16] = 15 && Pre.ints[17] = 13 &&
  Pre.ints[18] = 11 && Pre.ints[19] = 4
}

run { Pre.isPerm[Post] && Post.isSorted } for 20 seq, 6 int
```

There are two main approaches to representing this in Kodkod:

1. *We could tell the Alloy Analyzer to generate Kodkod-based Java code, call `toString()` on the abstract syntax tree, and fiddle a little bit with the output to make it comply with Kodkodi's input syntax.*

   From an Alloy specification, we can generate Java code by chosing "Output Kodkod to file" as the SAT Solver in the Alloy Analyzer's "Options" menu.

   Unfortunately, for the example above, the generated code is too large for the Java compiler, which simply bails out. In general, we would need to

rename the atoms and relations so that they follow Kodkodi's strict naming conventions and change a few syntactic items.

2. *We ignore the Alloy model and start from scratch in Kodkodi.*

   This gives a specification like the following:

```
solver: "MiniSat"
bit_width: 6
univ: u21
bounds r0 /* Pre.ints */:
{[A0, A7], [A1, A2], [A2, A4], [A3, A3], [A4, A3], [A5, A8], [A6,
A5], [A7, A20], [A8, A18], [A9, A1], [A10, A10], [A11, A5], [A12,
A7], [A13, A12], [A14, A2], [A15, A19], [A16, A15], [A17, A13],
[A18, A11], [A19, A4]}
bounds r1 /* Post.ints */: [{}, u20->u21]
int_bounds: [{A0}, {A1}, {A2}, {A3}, {A4}, {A5}, {A6}, {A7}, {A8},
{A9}, {A10}, {A11}, {A12}, {A13}, {A14}, {A15}, {A16}, {A17},
{A18}, {A19}, {A20}]
solve FUNCTION(r1, u20->one u21)
&& (all [S0 : one univ] | #(r1.S0) = #(r0.S0))
&& (all [S0 : one u19] | sum(S0.r1) <= sum(Int[sum(S0) + 1].r1));
```

   The first two lines,

```
solver: "zChaff"
bit_width: 6
```

   are configuration options. Then we specify that the universe should consist of exactly 21 atoms:

```
univ: u21
```

   The atoms are called `A0` to `A20`. Next, we specify the values for the `Pre.ints` relation as a Kodkod bound:

```
bounds r0 /* Pre.ints */:
{[A0, A7], [A1, A2], [A2, A4], [A3, A3], [A4, A3], [A5, A8], [A6,
A5], [A7, A20], [A8, A18], [A9, A1], [A10, A10], [A11, A5], [A12,
A7], [A13, A12], [A14, A2], [A15, A19], [A16, A15], [A17, A13],
[A18, A11], [A19, A4]}
```

   In Kodkodi, all binary relations must be called `r`*j*, where *j* is a natural number. The comment is there to remind us that `r0` corresponds to `Pre.ints` in the Alloy specification.

```
bounds r1 /* Post.ints */: [{}, u20->u21]
```

   For `Post.ints`, we specify the empty set `{}` as the lower bound and the Cartesian product `{A0 .. A19}->{A0 .. A20}` as the upper bound.

```
int_bounds: [{A0}, {A1}, {A2}, {A3}, {A4}, {A5}, {A6}, {A7}, {A8},
{A9}, {A10}, {A11}, {A12}, {A13}, {A14}, {A15}, {A16}, {A17},
{A18}, {A19}, {A20}]
```

Since we need the integers for addition, we must associate atoms with the integers we need. Here we simply let `A0` represent `0`, `A1` represent `1`, and so on.

```
solve FUNCTION(r1, u20->one u21)
&& (all [S0 : one univ] | #(r1.S0) = #(r0.S0))
&& (all [S0 : one u19] | sum(S0.r1) <= sum(Int[sum(S0) + 1].r1));
```

Finally, we specify the formula to solve. The first line ensures that `r1` (i.e., `Post.ints`) is a function rather than an arbitrary relation. The second and third lines are adapted directly from the Alloy specification.

Sorting [7, 2, 4, 3, 3, 8, 5, 20, 18, 1, 10, 5, 7, 12, 2, 19, 15, 13, 11, 4] should give [1, 2, 2, 3, 3, 4, 4, 5, 5, 7, 7, 8, 10, 11, 12, 13, 15, 18, 19, 20]. Let us verify that this is the case by running Kodkodi:

```
*** PROBLEM 1 ***

--OUTCOME--
SATISFIABLE

--INSTANCE--
relations: {r0=[[A0, A7], [A1, A2], [A2, A4], [A3, A3], [A4, A3],
[A5, A8], [A6, A5], [A7, A20], [A8, A18], [A9, A1], [A10, A10],
[A11, A5], [A12, A7], [A13, A12], [A14, A2], [A15, A19], [A16,
A15], [A17, A13], [A18, A11], [A19, A4]], r1=[[A0, A1], [A1, A2],
[A2, A2], [A3, A3], [A4, A3], [A5, A4], [A6, A4], [A7, A5], [A8,
A5], [A9, A7], [A10, A7], [A11, A8], [A12, A10], [A13, A11], [A14,
A12], [A15, A13], [A16, A15], [A17, A18], [A18, A19], [A19, A20]]}

--STATS--
p cnf 8166 29484
primary variables: 420
parsing time: 72 ms
translation time: 359 ms
solving time: 434 ms
```

The result is correct.

6 Known Bugs and Limitations

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

- The `-server` command-line option, which makes Kodkodi run as a TCP server, is limited to a single connection. Furthermore, any error occurring when processing one problem breaks the connection.

References

[1] Blanchette, J. C.: *Picking Nits: A User's Guide to Nitpick 1.2.1 for Isabelle/HOL 2009*. http://isabelle.in.tum.de/~blanchet/nitpick-1.2.1/Nitpick/manual/nitpick.pdf (2009)

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

[3] 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)

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

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

[6] Nitpick: Yet Another Counterexample Generator for Isabelle/HOL, http://isabelle.in.tum.de/~blanchet/nitpick.html

[7] PicoSAT, http://fmv.jku.at/picosat/