The RISC Algorithm Language (RISCAL)

Tutorial and Reference Manual
(Version 1.0.6)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

August 4, 2017

Abstract

This report documents the RISC Algorithm Language (RISCAL). RISCAL is a language and associated software system for describing (potentially nondeterministic) mathematical algorithms over discrete structures, formally specifying their behavior by logical formulas (program annotations in the form of preconditions, postconditions, and loop invariants), and formulating the mathematical theories (by defining functions and predicates and stating theorems) on which these specifications depend. The language is based on a type system that ensures that all variable domains are finite; nevertheless the domain sizes may depend on unspecified numerical constants. By instantiating these constants with values, all algorithms, functions, and predicates become executable, and all formulas become decidable. Indeed the RISCAL software implements a (parallel) model checker that allows to verify the correctness of algorithms and the associated theories with respect to their specifications for all possible input values of the parameter domains.

Contents

1 Introduction

The formal verification of computer programs is a very challenging task. If the program operates on an unbounded domain of values, the only general verification technique is to generate, from the text of the program and its specification, verification conditions, i.e., logical formulas whose validity ensures the correctness of the program with respect to its specification. The generation of such conditions generally requires from the human additional program annotations that express meta-knowledge about the program such as loop invariants and termination measures. Since for more complex programs proofs of these formulas by fully automatic reasoners rarely succeed, typically interactive proving assistants are applied where the human helps the software to construct successful proofs.

This may become a frustrating task, because usually the first verification attempts are doomed to fail: the verification conditions are often *not* valid due to (also subtle) deficiencies in the program, its specification, or annotations. The main problem is then to find out whether a proof fails because of an inadequate proof strategy or because the condition is not valid and, if the later is the case, which errors make the formula invalid. Typically, therefore most time and effort in verification is actually spent in attempts to prove invalid verification conditions, often due to inadequate annotations, in particular due to loop invariants that are too strong or too weak.

For this reason, programs are often restricted to make fully automatic verification feasible that copes without extra program annotations and without manual proving efforts. One possibility is to restrict the domain of a program such that it only operates on a finite number of values, which allows to apply model checkers that investigate all possible executions of a program. Another possibility is to limit the length of executions being considered; then bounded model checkers (typically based on SMT solvers, i.e., automatic decision procedures for combinations of decidable logical theories) are able to check the correctness of a program for a subset of the executions. However, while being fully automatic, (bounded) model checking is time- and memory-consuming, and ultimately does not help to verify the general correctness of a program operating on unbounded domains with executions of unbounded length.

Based on prior expertise with computer-supported program verification, also in educational scenarios [27, 22], we have started the development of RISCAL (RISC Algorithm Language) as an attempt to make program verification less painful. The term "algorithm language" indicates that RISCAL is intended to model, rather than low-level code, high-level algorithms as can be found in textbooks on discrete mathematics [25]. RISCAL thus provides a rich collection of data types (e.g., sets and maps) and operations (e.g., quantified formulas as Boolean conditions and implicit definitions of values by formulas) but only cares about the correctness of execution, not its efficiency. The core idea behind RISCAL is to use automatic techniques to find problems in a program, its specification, or its annotations, that may prevent a successful verification *before* actually attempting to prove verification conditions; we thus aim to start a proof of verification conditions only when we are reasonably confident that they are indeed valid.

As a first step towards this goal, RISCAL restricts in a program P all program variables and mathematical variables to finite types where the number of values of a type T, however, may depend on an unspecified constant $n \in \mathbb{N}$. If we set n to some concrete value c, we get an instance P_c of P where all specifications and annotations can be effectively evaluated during the execution of the program (runtime assertion checking). Furthermore, we can execute a program

and check the annotations for all possible inputs (model checking); only if we do not find errors, the verification of the general program P may be attempted. The current version of the RISCAL software documented in this report thus supports model checking of finite domain instances of programs via the runtime assertion checking of all possible executions, which is based on the executability of all specifications and annotations (further mechanisms will be added in time).

The RISCAL software is freely available as open source under the GNU General Public License, Version 3 at

`https://www.risc.jku.at/research/formal/software/RISCAL`

with this document included in electronic form as the manual for the software.

The remainder of this document is organized as follows: Section 2 represents a tutorial into the practical use of the RISCAL language and system based on a concrete example of a RISCAL specification. Section 3 elaborates more examples to deepen the understanding. Section 4 relates our work to prior research; Section 5 elaborates our plans for the future evolution of RISCAL. Appendix A provides a detailed documentation for the use of the system. Appendix B represents the reference manual for the specification language. Appendix C gives the full source of the specifications used in the tutorial; this source is also included in the distribution.

2 A Quick Start

We start with a quick overview on the RISCAL specification language and associated software system whose graphical user interface is depicted in Figure 1 (an enlarged version is shown in Figure 2 on page 31).

System Overview The system is started from the command line by executing

```
RISCAL &
```

The user interface is divided into two parts. The left part mainly embeds an editor panel with the current specification. The right part is mainly filled by an output panel that shows the output of the system when analyzing the specification that is currently loaded in the editor. The top of both parts contains some interactive elements for controlling the editor respectively the analyzer. Appendix A.3 explains the user interface in more detail.

The system remembers across invocations the last specification file loaded into the editor, i.e., when the software is started, the specification used in the last invocation is automatically loaded. Likewise the options selected in the right panel are remembered across invocations. However, a button  (Reset System) in the right panel allows to reset the system to a clean state (no specification loaded and all options set to their default values).

Specifications are written as plain text files in Unicode format (UTF-8 encoding) with arbitrary file name extension (e.g., `.txt`). The RISCAL language uses several Unicode characters that cannot be found on keyboards, but for each such character there exists an equivalent string in ASCII format that can be typed on a keyboard. While the RISCAL grammar supports both alternatives, the use of the Unicode characters yields much prettier specifications and is thus recommended. The RISCAL editor can be used to translate the ASCII string to the Unicode

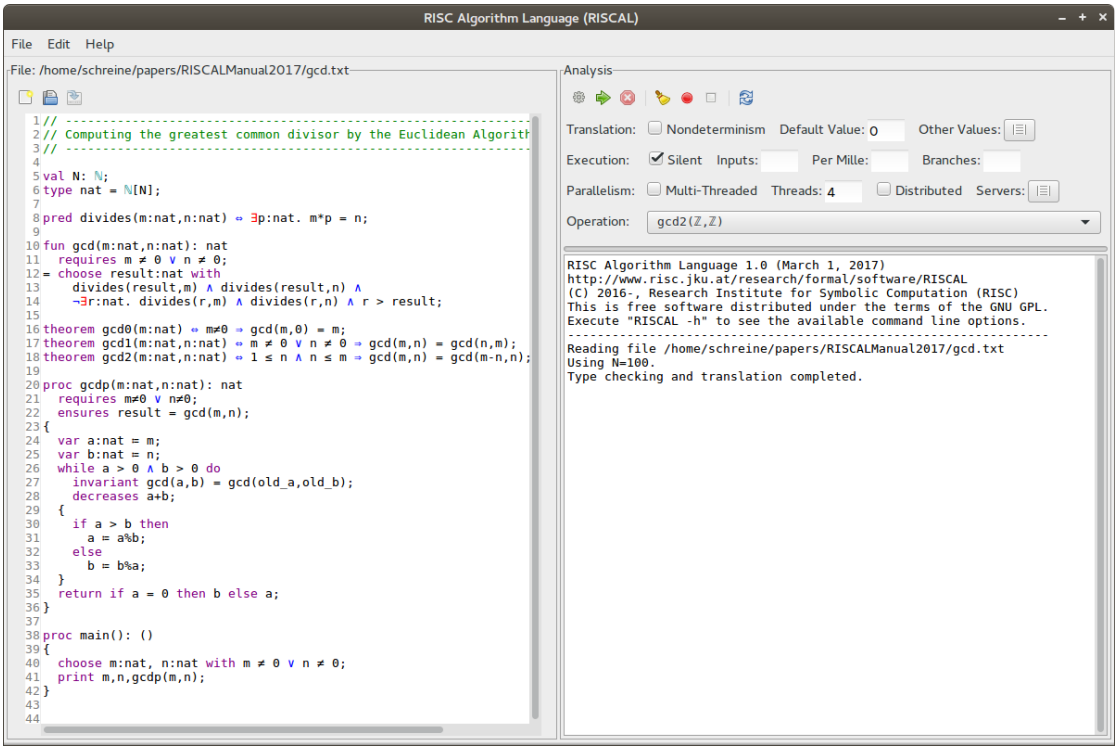

Figure 1: The RISCAL User Interface

character by first typing the string and then (when the editor caret is immediately to the right of this string) pressing <Ctrl>-#, i.e, the Control key and simultaneously the key depicting #. Also later such textual replacements can be performed by positioning the editor caret to the right of the string and pressing <Ctrl>-#. The current table of replacements is given in Appendix B.1.

Whenever a specification is loaded from disk respectively saved to disk after editing, it is immediately processed by a syntax checker and a type checker. Error messages are displayed in the output panel; the positions of errors are displayed by red markers in the editor window; moving the mouse pointer over such a marker also displays the corresponding error message.

Specification Language As a first example of the specification language (which is defined in Appendix B), we write a specification consisting of a mathematical theory of the greatest common divisor and its computation by the Euclidean algorithm. This specification (whose full content is given in Appendix C.1) starts with the declarations

```
val N: ℕ;
type nat = ℕ[N];
```

that introduce a natural number constant N a type *nat* that consists of the values $0, \ldots, N$ (the symbol $\mathbb{N}$ may be entered as `Nat` followed by pressing the keys <Ctrl>-#) which corresponds

to the mathematical definition

$$nat := \{x \in \mathbb{N} \mid x \leq N\}.$$

The value of N is not defined in the specification but in the RISCAL software. We may enter this value either in the field "Default Value" in the right part of the window or we may open with the button "Other Values" a window that allows to set different values for different constants; if there is no entry for a specific constant, the "Default Value" is used.

The specification defines then a predicate

```
pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;
```

which corresponds to the mathematical definition

$$divides \subseteq nat \times nat$$
$$divides(m, n) :\Leftrightarrow \exists p \in nat.\ m \cdot p = n$$

In other words, $divides(m, n)$ means "m divides n" which is typically written as $m|n$.

We then introduce the "greatest common divisor" function as

```
fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose result:nat with
    divides(result,m) ∧ divides(result,n) ∧
    ¬∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;
```

This function is defined by an implicit definition; for any $m \in nat$ and $n \in nat$ with $m \neq 0$ or $n \neq 0$, its result is the largest value $result \in nat$ that divides both m and n.

The function can be used to define some other values, e.g.

```
val g:nat = gcd(N,N-1);
```

which corresponds to the mathematical definition

$$g \in nat,\ g := gcd(N, N-1)$$

This function satisfies certain general properties stated as follows:

```
theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);
```

Each such "theorem" is represented by a named predicate which is implicitly claimed to be true for all possible applications, corresponding to the mathematical propositions

$$\forall m \in nat.\ m \neq 0 \Rightarrow gcd(m, 0) = m$$
$$\forall m \in nat, n \in nat.\ m \neq 0 \vee n \neq 0 \Rightarrow gcd(m, n) = gcd(n, m)$$
$$\forall m \in nat, n \in nat.\ 1 \leq n \wedge n \leq m \Rightarrow gcd(m, n) = gcd(m \bmod n, n)$$

The symbol `%` thus stands for the mathematical "modulus" operator (arithmetic remainder).

Now we write a procedure *gcdp* that implements the Euclidean algorithm:

```
proc gcdp(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
  ensures result = gcd(m,n);
{
  var a:nat := m;
  var b:nat := n;
  while a > 0 ∧ b > 0 do
    invariant gcd(a,b) = gcd(old_a,old_b);
    decreases a+b;
  {
    if a > b then
      a := a%b;
    else
      b := b%a;
  }
  return if a = 0 then b else a;
}
```

The clauses `requires` and `ensures` state that for any arguments m, n with $m \neq 0$ or $n \neq 0$ (i.e., for any argument that satisfies the given precondition), the result of this procedure (denoted by the keyword `result`) is indeed the greatest common divisor of m and n, as specified above (i.e., the result satisfies the given postcondition). The `invariant` states the crucial property for proving the correctness of the algorithm: before and after every iteration of the loop, the greatest common divisor of the current values of program variables a and b equals the greatest common divisor of their initial values (denoted by the keyword prefix `old_`), i.e., the greatest divisor of m and n. The clause `decreases` states the crucial property for the termination of the algorithm: the value $a + b$ decreases by every loop iteration but does not become negative. While all these loop annotations are not necessary for executing the algorithm, they formally express additional knowledge that aid our understanding of the algorithm.

Above procedure demonstrates that RISCAL incorporates an algorithmic language with the usual programming language constructs. However, unlike a classical programming language, this algorithm language allows also nondeterministic executions as in the following procedure:

```
proc main(): ()
{
  choose m:nat, n:nat with m ≠ 0 ∨ n ≠ 0;
  print m,n,gcdp(m,n);
}
```

This procedure does not return a value (indicated by the return type `()`) but just prints the arguments and result of some application of procedure *gcdp*. The argument values m, n for its application are not uniquely determined: the `choose` command selects for m and n some values in *nat* such that at least one of them is not zero.

Language Features As shown above, a RISCAL specification may contain definitions of

- types,
- constants, functions, predicates,
- theorems, and
- procedures

which may depend on (declared but) undefined natural number constants. Functions may be explicitly defined or implicitly specified by predicates, theorems are special predicates that always yield "true", procedures may contain apart from the usual algorithmic constructs also nondeterministic choices, and functions and procedures may be annotated with pre- and postconditions respectively loop invariants and termination measures. The language does not distinguish between logical terms and formulas, a formula is just a term of a type `Bool` and a predicate is a function with that result type.

Furthermore, there is no difference between logical formulas and conditions in program constructs; every logical formula may be in a procedure for example used as a loop condition. Likewise, there is no difference between logical terms and program expressions; every logical term may be for example used on the right hand side of an assignment statement. Procedures have (apart from potential output) no other effect than returning values (also the procedure `main` above implicitly returns a value `()`); they may be thus also used as functions in logical formulas.

The RISCAL language has been designed in such a way that

- every type has only finitely many values, and thus
- every language construct is executable, and thus
- every constant, function, predicate, theorem, procedure can be evaluated.

For instance, the truth value of a formula like

```
∃p:nat. m·p = n
```

can be determined by evaluating, for every possible *nat*-value p, the formula $m \cdot p = n$. If there is at least one value for which this equality holds, the formula is true, otherwise it is false. Likewise, the function

```
fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose result:nat with ...
```

can be evaluated by enumerating all possible candidate values for the result of the function. The function result is then one such candidate for which the condition "..." holds (if this is not the case for any candidate, the execution may abort with an error message). Therefore also all function/predicate/procedure annotations `requires`, `ensures`, `invariant`, and `decreases` are executable.

In summary, every RISCAL specification is completely executable; in particular, all stated claims (theorems and function/predicate/procedure annotations) are checkable.

Execution and Model Checking Whenever a specification file is loaded or saved (or when the button is saved), the corresponding specification is processed,i.e., checked for syntactic and type errors (with graphical markers displaying the locations of the errors) and translated into an executable representation. For this purpose, the value of every constant introduced by a `val` clause on the top level of a specification is determined: if a natural number constant c is just declared, the value of c is taken from the user interface, either from the panel "Other Values"or (if this panel does not list a value for c) from the box "Default Value". If the value of a constant is defined by a term, this value is determined by evaluating the term (which may contain arbitrary computations including the application of user-defined functions). The values of constants may be used as bounds in types; a specification is thus always processed for a specific set of values for the global constants (if the user changes the values, the specification is automatically re-processed before execution). In our example specification, we may thus get the output

```
RISC Algorithm Language 1.0 (March 1, 2017)
http://www.risc.jku.at/research/formal/software/RISCAL
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCAL -h" to see the available command line options.
------------------------------------------------------------------
Reading file /home/schreine/papers/RISCALManual2017/gcd.txt
Using N=3.
Computing the value of g...
Type checking and translation completed.
```

which indicates that the user provided the value 3 for constant N and that the value of g was computed by evaluating the definition.

After the processing of a specification, the menu "Operation" lists all operations together with their argument types (operations with different argument types may have the same name). We may e.g. select the operation `main()` which selects the method

```
proc main(): ()
{
  choose m:nat, n:nat with m ≠ 0 ∨ n ≠ 0;
  print m,n,gcdp(m,n);
}
```

By pressing the button  (Start Execution) the system executes `main()` which produces (assuming that option `Nondeterminism` has not been selected) the output

```
Executing main().
Run of deterministic function main():
0,1,1
Result (34 ms): ()
Execution completed (96 ms).
Not all nondeterministic branches may have been considered.
```

The system has thus chosen the values 0 for m and 1 for n and computed their greatest common divisor 1.

However, if we select the option `Nondeterminism` and then press the button  , the specification is first re-processed and then executed with the following output:

```
Executing main().
Branch 0 of nondeterministic function main():
0,1,1
Result (11 ms): ()
Branch 1 of nondeterministic function main():
0,2,2
Result (24 ms): ()
Branch 2 of nondeterministic function main():
0,3,3
Result (11 ms): ()
...
Branch 13 of nondeterministic function main():
3,2,1
Result (8 ms): ()
Branch 14 of nondeterministic function main():
3,3,3
Result (8 ms): ()
Branch 15 of nondeterministic function main():
No more results (434 ms).
Execution completed (441 ms).
```

Thus the program was executed for all possible choices for m and n subject to the condition $m \neq 0 \vee n \neq 0$ and computed the greatest common divisor. In this execution, all annotations specified in `requires`, `ensures`, `invariant`, and `decreases` have been checked by evaluating the corresponding formulas respectively terms, thus also evaluating the implicitly defined function `gcd` and the predicate `divides`. For instance, changing the postcondition of `gcdp` to

```
ensures result = 1;
```

yields output

```
Executing main().
Branch 0 of nondeterministic function main():
0,1,1
Result (6 ms): ()
Branch 1 of nondeterministic function main():
ERROR in execution of main(): evaluation of
  ensures result = 1;
at line 24 in file gcd.txt:
  postcondition is violated
ERROR encountered in execution.
```

which demonstrates that the second execution of `main` violated the specification. Likewise, setting the termination measure to

```
decreases a;
```

produces the output

```
Executing main().
Branch 0 of nondeterministic function main():
0,1,1
Result (10 ms): ()
...
Branch 4 of nondeterministic function main():
ERROR in execution of main(): evaluation of
  decreases a;
at line 30 in file gcd.txt:
  variant value 1 is not less than old value 1
ERROR encountered in execution.
```

However, rather than `main` in nondeterministic mode, we may also executed `gcdp` for all possible inputs. We thus deselect "Nondeterminism" and select from the menu `Operation` the operation `gcdp(`$\mathbb{Z}$`,`$\mathbb{Z}$`)`, which yields the following execution:

```
Executing gcdp(ℤ,ℤ) with all 16 inputs.
Ignoring inadmissible inputs...
Run 1 of deterministic function gcdp(1,0):
Result (1 ms): 1
Run 2 of deterministic function gcdp(2,0):
Result (0 ms): 2
...
Run 15 of deterministic function gcdp(3,3):
Result (1 ms): 3
Execution completed for ALL inputs (206 ms, 15 checked, 1 inadmissible).
Not all nondeterministic branches may have been considered.
```

The system thus runs `gcdp` with all "admissible" inputs, i.e., with all argument tuples that satisfy the specified precondition

```
requires m≠0 ∨ n≠0;
```

The system thus executes the operation (and checks all annotations) for 15 inputs excluding the inadmissible input $m = 0$ and $n = 0$. The last line of the output reminds us that we have executed the system in deterministic mode which is generally faster but does not consider all possible execution branches resulting from nondeterministic choices such the one performed in the definition of `gcd`.

By switching on the option "Silent", the output is compacted to

```
Executing gcdp(ℤ,ℤ) with all 16 inputs.
Execution completed for ALL inputs (6 ms, 15 checked, 1 inadmissible).
Not all nondeterministic branches may have been considered.
```

which just gives the essential information.

Parallelism If we increase the value of N to say 60, checking all possible inputs may take some time. We thus switch on the option "Multi-Threaded" and set "Threads" to 4. The system thus applies 4 concurrent threads for the application of the operation which is (on a computer with 4 processor cores) much faster and yields output:

```
Executing gcdp(ℤ,ℤ) with all 3721 inputs.
PARALLEL execution with 4 threads (output disabled).
1336 inputs (955 checked, 1 inadmissible, 0 ignored, 380 open)...
2193 inputs (1812 checked, 1 inadmissible, 0 ignored, 380 open)...
3005 inputs (2629 checked, 1 inadmissible, 0 ignored, 375 open)...
3721 inputs (3445 checked, 1 inadmissible, 0 ignored, 275 open)...
Execution completed for ALL inputs (8826 ms, 3720 checked, 1 inadmissible).
Not all nondeterministic branches may have been considered.
```

The statistics output and the growing blue bar displayed on top of the output area displays the progress of a longer running computation. To interrupt such an execution, we may press the button  (Stop Execution).

For even larger inputs, we may also set the option "Distributed" and and thus partially delegate computations to other instances of the RISCAL software, possibly running on other computers (e.g., high performance servers) running in the local network or being sited in other remote networks to which we are connected by the Internet. For this, we also have configure the connection information in menu "Servers" appropriately, see Appendix A.4 for details. The output for $N = 100$ may then look as follows:

```
Executing gcdp(ℤ,ℤ) with all 10201 inputs.
Executing "/software/RISCAL/etc/runssh
  qftquad2.risc.jku.at 4"...
Connecting to qftquad2.risc.uni-linz.ac.at:52387...
Executing "/software/RISCAL/etc/runmach 4"...
Connecting to localhost:9999...
Connected to remote servers.
PARALLEL execution with 4 local threads and 2 remote servers (output disabled).
939 inputs (544 checked, 1 inadmissible, 0 ignored, 394 open)...
2819 inputs (1117 checked, 1 inadmissible, 0 ignored, 1701 open)...
2819 inputs (1424 checked, 1 inadmissible, 0 ignored, 1394 open)...
4605 inputs (2851 checked, 1 inadmissible, 0 ignored, 1753 open)...
5327 inputs (3799 checked, 1 inadmissible, 0 ignored, 1527 open)...
6339 inputs (4779 checked, 1 inadmissible, 0 ignored, 1559 open)...
8035 inputs (6363 checked, 1 inadmissible, 0 ignored, 1671 open)...
```

```
8716 inputs (7408 checked, 1 inadmissible, 0 ignored, 1307 open)...
Execution completed for ALL inputs (18500 ms, 10200 checked, 1 inadmissible).
Not all nondeterministic branches may have been considered.
```

Here, in addition to four local threads, two remote servers are applied, one with Internet name `qftquad2.risc.uni-linz.ac.at`; the other is called `localhost` because it is connected to our process via a local proxy process that bypasses a firewall.

We may not only check procedures but also functions, relations, and especially theorems; in the later case we check whether all applications of the theorem yield value "true". For instance, we may check the theorem `gcd2(ℤ,ℤ)` defined as

```
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);
```

for which the output

```
Executing gcd2(ℤ,ℤ) with all 10201 inputs.
PARALLEL execution with 4 threads (output disabled).
1904 inputs (1704 checked, 0 inadmissible, 0 ignored, 200 open)...
3757 inputs (3673 checked, 0 inadmissible, 0 ignored, 84 open)...
7372 inputs (6436 checked, 0 inadmissible, 0 ignored, 936 open)...
Execution completed for ALL inputs (7127 ms, 10201 checked, 0 inadmissible).
Not all nondeterministic branches may have been considered.
```

validates its correctness.

As demonstrated by above examples, the RISCAL software encompasses the *execution* of specifications with *runtime assertion checking* (checking correctness of a computation for some input) and *model checking* (checking correctness for all/many possible inputs).

3 More Examples

We continue by presenting some more examples of RISCAL specifications.

Insertion Sort We are going to specify the Insertion Sort algorithm for sorting arrays of length N that hold natural numbers up to size M, based on the following declarations (see Appendix C.2 for the full specification):

```
val N:Nat;
val M:Nat;
```

We make use of the following type definitions

```
type nat = Nat[M];
type array = Array[N,nat];
type index = Nat[N-1];
```

where type `array` is the type of all arrays of length N of values of type `nat` to be accessed by indices $0, \ldots, N-1$; type `index` denotes the domain of legal indices.

The insertion sort algorithm is then defined as follows:

```
proc sort(a:array): array
  ensures ∀i:nat. i < N-1 => result[i] ≤ result[i+1];
  ensures ∃p:Array[N,index].
            (∀i:index,j:index. i ≠ j ⇒ p[i] ≠ p[j]) ∧
            (∀i:index. a[i] = result[p[i]]);
{
  var b:array = a;
  for var i:Nat[N]:=1; i<N; i:=i+1 do
    decreases N-i;
  {
    var x:nat := b[i];
    var j:Int[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do
      decreases j+1;
    {
      b[j+1] := b[j];
      j := j-1;
    }
    b[j+1] := x;
  }
  return b;
}
```

The postcondition of this algorithm states that the resulting array is sorted in ascending order and that that it is a permutation of the input array, i.e., that there exists a permutation p of indices such that the result array holds at position $p[i]$ the value of the input array at position i. The loop is annotated with appropriate termination measures (invariants have been omitted).

The specification demonstrates that arrays can be used in a style similar to most imperative programming languages. Semantically, however, arrays in RISCAL differ from programming language arrays in that an array assignment `a[i] := e` does not update the existing array but overwrites the program variable `a` with a new array that is identical to the original one except that it holds at position i value e. RISCAL arrays thus have *value semantics* rather than *pointer semantics*. Correspondingly, above procedure does not update the argument array a; it rather creates a new array b that is returned as the result of the procedure (actually, because of the semantics of the array assignment, the use of a separate variable b is *not* necessary; the program could have just used a and terminated with the statement `return b`).

We can demonstrate a single run of the system by defining the procedure

```
proc main(): Unit
{
  choose a: array;
  print a, sort(a);
}
```

and selecting in menu "Operation" the entry `main()`. Executing this specification for $N = 3$ and in "Deterministic" mode gives output

```
Run of deterministic function main():
[0,0,0,0],[0,0,0,0]
Result (6 ms): ()
Execution completed (46 ms).
Not all nondeterministic branches may have been considered.
```

which however only demonstrates that the array holding 0 everywhere is appropriately "sorted". By setting the option `Nondeterministic`, the output

```
Executing main().
Branch 0 of nondeterministic function main():
[0,0,0],[0,0,0]
Result (8 ms): ()
Branch 1 of nondeterministic function main():
[1,0,0],[0,0,1]
Result (8 ms): ()
Branch 2 of nondeterministic function main():
[2,0,0],[0,0,2]
...
Branch 255 of nondeterministic function main():
[3,3,3,3],[3,3,3,3]
Result (10 ms): ()
Branch 256 of nondeterministic function main():
No more results (5056 ms).
Execution completed (5062 ms).
```

demonstrates that this is the case for all other inputs as well. Setting the option `Silent` and selecting the operation `sort(Map[Array[ℤ]])`, gives with the output

```
Executing sort(Array[ℤ]) with all 256 inputs.
Execution completed for ALL inputs (327 ms, 256 checked, 0 inadmissible).
Not all nondeterministic branches may have been considered.
```

the core information in much shorter time.

DPLL Algorithm As a somewhat bigger example, we present the core of the DPLL (Davis, Putnam, Logemann, Loveland) algorithm for deciding the satisfiability of propositional logic formulas with at most n variables in conjunctive normal form. We start with the following declaration (the full specification is given in Appendix C.3):

```
val n: ℕ;
```

A literal (a propositional variable in positive or negated form) is represented by a positive respectively negative integer; a clause (a conjunction of literals) is represented by a set of literals; a formula (a disjunction of clauses) is represented by a set of clauses. A valuation of a formula (a mapping of propositional variables to truth values) is represented by the set of literals that are mapped to "true". All of this gives rise to the following type definitions:

```
type Literal = ℤ[-n,n];
type Clause  = Set[Literal];
type Formula = Set[Clause];
type Valuation  = Set[Literal];
```

Actually, these definitions only introduce "raw types": not every value of this type is meaningful. Based on the predicate

```
pred consistent(l:Literal,c:Clause) ⇔ ¬(l∈c ∧ -l∈c);
```

we introduce side conditions that all meaningful values of the corresponding types must satisfy:

```
pred literal(l:Literal) ⇔ l≠0;
pred clause(c:Clause) ⇔ ∀l∈c. literal(l) ∧ consistent(l,c);
pred formula(f:Formula) ⇔ ∀c∈f. clause(c);
pred valuation(v:Valuation) ⇔ clause(v);
```

We can define the predicates that state when a valuation satisfies a literal, a clause, and a formula, respectively:

```
pred satisfies(v:Valuation, l:Literal) ⇔ l∈v;
pred satisfies(v:Valuation, c:Clause) ⇔ ∃l∈c. satisfies(v, l);
pred satisfies(v:Valuation, f:Formula) ⇔ ∀c∈f. satisfies(v,c);
```

We thus define the core notion of the *satisfiability* of a formula respectively, it's counterpart, *validity*:

```
pred satisfiable(f:Formula) ⇔
  ∃v:Valuation. valuation(v) ∧ satisfies(v,f);
pred valid(f:Formula) ⇔
  ∀v:Valuation. valuation(v) ⇒ satisfies(v,f);
```

We define the negation of a formula

```
fun not(f: Formula):Formula =
  { c | c:Clause with clause(c) ∧ ∀d∈f. ∃l∈d. -l∈c };
theorem notFormula(f:Formula)
  requires formula(f);
⇔ formula(not(f));
```

and define core relationship between both notions: a formula is valid, if its negation is not satisfiable:

```
theorem notValid(f:Formula)
  requires formula(f);
⇔ valid(f) ⇔ ¬satisfiable(not(f));
```

Having established the basic theory of propositional formulas and their satisfiability, we introduce some auxiliary notions used by the DPLL algorithm, namely the set of all literals of a formula

```
fun literals(f:Formula):Set[Literal] =
  {l | l:Literal with ∃c∈f. l∈c};
```

and the result of setting a literal l in formula f to "true":

```
fun substitute(f:Formula,l:Literal):Formula =
  {c\{-l} | c∈f with ¬(l∈c)};
```

We are now in the position to give the recursive version of the algorithm (omitting for brevity the optimizations that actually make the algorithm efficient):

```
multiple pred DPLL(f:Formula)
  requires formula(f);
  ensures result ⇔ satisfiable(f);
  decreases |literals(f)|;
⇔
  if f = ∅[Clause] then
    ⊤
  else if ∅[Literal] ∈ f then
    ⊥
  else
    choose l∈literals(f) in
    DPLL(substitute(f,l)) ∨ DPLL(substitute(f,-l));
```

The specification of the algorithm states that for every well-formed formula f the algorithm yields "true" if and only if f is satisfiable. If it cannot easily decide the satisfiability of f, the algorithm chooses a literal in that is substituted once by "true" and once by "false" and calls itself recursively on the resulting formulas; if one of them is satisfiable, also f is satisfiable. The algorithm terminates because in every recursive invocation the number of literals in the formula is decreased. The keyword `multiple` in front of the definition is necessary for recursive functions/predicates with nondeterministic semantics, as in the case of this function that applies the `choose` operator.

For asserting the termination the iterative version of the algorithm, we introduce a couple of auxiliary notions

```
fun vars(f:Formula): Set[ℕ[n]] =
  { if l>0 then l else -l | l ∈ literals(f) };
val m = 2^(n+1)-1;
fun size(f:Formula): ℕ[m] = 2^(|vars(f)|+1)-1;
fun size(stack:Array[n+1,Formula], i:ℕ[n+1]): ℕ[m] =
  ∑k:ℕ[n] with k<i. size(stack[k]);
```

which ultimately give a measure for the complexity of the work that is still to be performed for i formulas stored at the beginning of an array *stack* (see the explanations below).

The iterative version of the algorithm can then be formulated and provided with correctness annotations as follows:

```
proc DPLL2(f:Formula): Bool
  requires formula(f);
  ensures result ⇔ satisfiable(f);
{
  var satisfiable: Bool := ⊥;
  var stack: Array[n+1,Formula] := Array[n+1,Formula](∅[Clause]);
  var number: ℕ[n+1] := 0;
  stack[number] := f;
  number := number+1;
  while ¬satisfiable ∧ number>0 do
    invariant 0 ≤ number ∧ number ≤ n+1;
    invariant satisfiable(f) ⇔ satisfiable ∨
      ∃i:ℕ[n+1] with i<number. satisfiable(stack[i]);
    decreases if satisfiable then 0 else size(stack, number);
  {
    number := number-1;
    var g:Formula := stack[number];
    if g = ∅[Clause] then
      satisfiable := ⊤;
    else if ¬∅[Literal]∈g then
    {
      choose l∈literals(g);
      stack[number] := substitute(g,-l);
      number := number+1;
      stack[number] := substitute(g,l);
      number := number+1;
    }
  }
  return satisfiable;
}
```

The algorithm operates on a stack to which it initially pushes the original formula f. It then iteratively pops the top formula g from the stack; if the formula is not trivially satisfiable, it chooses a literal in g that is substituted once by "true" and once by "false"; the resulting formulas are pushed to the stack again. The algorithm terminates when the stack becomes empty (f is then not satisfiable) or if the top formula g is satisfiable (then also f is satisfiable).

In addition to the specification of pre- and postcondition, the algorithm is also annotated with the core invariants from which the correctness of the algorithms can be deduced: the original formula is satisfiable if the variable *satisfiable* is set to "true" or if any of the formulas on the stack is satisfiable. It terminates, because the complexity of the work which remains on the stack (essentially the sum of the number of corresponding applications of the recursive algorithm to these formulas) decreases.

By setting $n = 3$ and defining

```
proc main0(): ()
```

```
{
  val f = {{1,2,3},{-1,2},{-2,3},{-3}};
  val r = DPLL2(f);
  print f,r;
}
```

we can validate the correctness of the (iterative version of the) algorithm for one particular input:

```
Executing main0().
Run of deterministic function main0():
{{1,2,3},{-1,2},{-2,3},{-3}},false
Result (36 ms): ()
Execution completed (100 ms).
Not all nondeterministic branches may have been considered.
```

However, when attempting to check the algorithm for all inputs

```
Executing DPLL2(Set[Set[ℤ]]) with all (at least 2^63) inputs.
PARALLEL execution with 4 threads (output disabled).
434480 inputs (768 checked, 140278 inadmissible, 0 ignored, 293434 open)...
434480 inputs (1792 checked, 429562 inadmissible, 0 ignored, 3126 open)...
711986 inputs (2545 checked, 587520 inadmissible, 0 ignored, 121921 open)...
1217971 inputs (3583 checked, 853627 inadmissible, 0 ignored, 360761 open)...
1500591 inputs (4096 checked, 1055731 inadmissible, 0 ignored, 440764 open)...
1724104 inputs (4096 checked, 1594493 inadmissible, 0 ignored, 125515 open)...
...
```

we first realize that there are extremely many (more than 2^{63}) of these and second that only a small minority of them are well-formed (most sets of sets of integers violate some of the type constraints). Unless we have an overwhelming amount of time (a couple of thousands of years) at our hand, we better restrict our input space. We therefore stop the execution and introduce constants for the maximum number of literals per clause and the maximum number of clauses per formula:

```
val cn: ℕ; // e.g. 2;
val fn: ℕ; // e.g. 20;
```

We then define a function that gives us all formulas with these constraints:

```
fun formulas(): Set[Formula] =
let
  literals = { l | l:Literal with literal(l) },
  clauses = { c | c ∈ Set(literals) with |c| ≤ cn ∧ clause(c) },
  formulas = { f | f ∈ Set(clauses) with |f| ≤ fn ∧ formula(f) }
in formulas;
```

Now we define a test program

```
proc main1(): ()
{
  // apply check to a specific set of formulas
  check DPLL with formulas();
}
```

in which the command check applies the algorithm to the specific set of formulas. By multi-threaded and distributed execution we then may check for $cn = 2$ and $fn = 20$ the selected subset of inputs in a quite limited amount of time:

```
Executing main1().
Executing DPLL(Set[Set[ℤ]]) with selected 524288 inputs.
Executing "/software/RISCAL/etc/runssh qftquad2.risc.jku.at 4"...
Connecting to qftquad2.risc.uni-linz.ac.at:56371...
Executing "/software/RISCAL/etc/runmach 4"...
Connecting to localhost:9999...
Connected to remote servers.
PARALLEL execution with 4 local threads and 2 remote servers (output disabled).
8668 inputs (4674 checked, 0 inadmissible, 0 ignored, 3994 open)...
22126 inputs (10737 checked, 0 inadmissible, 0 ignored, 11389 open)...
...
503297 inputs (477221 checked, 0 inadmissible, 0 ignored, 26076 open)...
Execution completed for SELECTED inputs (61037 ms, 524288 checked, 0 inadmissible).
Execution completed (89457 ms).
Not all nondeterministic branches may have been considered.
```

As this example demonstrates, model checking experiments may have to be planned with care to yield meaningful results with restricted (time and space) resources.

4 Related Work

RISCAL is related to a large body of prior research; we only give a short account of the work that seems most relevant.

Various languages arisen in the context of automated reasoning systems, while being designed for specifying logical theories, have some executable flavor: Theorema [6, 30] has been designed at RISC as a system for computer supported mathematical theorem proving and theory exploration; its PCS (Prove-Compute-Solve) paradigm considers computing as a special kind of proving. Also a compiler for an executable subset of the Theorema language to Java was developed. The language of the formal proof management system Coq [4, 8] allows to write executable algorithms from which functional programs in the programming languages OCaml, Haskell, and Scheme can be extracted; since the correctness of the algorithms can be formulated and verified in Coq, the programs are guaranteed to be correct. Similarly, the higher order logic HOL of the generic proof assistant Isabelle [20, 13] embeds a functional programming language in which algorithms can be defined and verified and converted to programs in OCaml, Haskell, SML, and Scala. In [19], Isabelle is used to define the formal semantics of a simple

imperative semantics from which executable code can be generated. However, all this work is targeted towards generating executable code from verified algorithms; it does not really address the problem stated in Section 1 of validating the correctness of algorithms before verification.

Also the abstract data type specification languages of the OBJ family [12, 11] include a large executable subset, essentially generalizations of functional programming languages. Using the supporting rewriting engines, programs in these languages can be also model-checked. However, the logics of these languages are based on equational logic which is much more restricted than predicate logic by enforcing the formulation of predicates in a low-level executable style.

As for algorithm languages, SETL [29, 28] is an old very high-level programming language based on set theory; it supports set comprehensions and quantified formulas as programming language constructs but not formal specifications. Alloy [14, 2] is a language for describing structures and their relationships, e.g., the configurations of a data structure arising from a sequence of modifying operations. The language is based on a relational logic; the Alloy Analyzer is a satisfiability solver that finds structures certifying constraints. While Alloy can be used to formulate algorithms/programs, this can become quite challenging [24], because the language differs very much from conventional languages. Event-B [1, 10] is a formal method for the modeling and analysis of systems, based on set theory as a modeling notation and the concept of refinement to represent systems at different abstraction levels; the supporting Rodin tool embeds an interactive proving assistant for verifying the correctness of system designs and refinements. The Event-B language is more oriented towards modeling reactive systems than conventional algorithms/programs [24].

RISCAL has been more directly influenced by the temporal logic of actions (TLA) [16, 31] which has evolved into a specification language TLA+ for describing concurrent systems. It also supports a the more conventional algorithm language PlusCal by translation to TLA+ specifications; PlusCal can be used to describe iterative algorithms but does not support recursion. TLA+/PlusCal is based on classical first order logic and set theory and supported by the TLC model checker and the TLA+ proof system. The RISCAL use of externally defined constants to restrict domains has been inspired by the corresponding use of constants by TLA+/PlusCal to restrict the sizes of sets. However, while RISCAL is statically typed, TLA+/PlusCal has no static type system; indeed all values are ultimately sets. PlusCal demonstrates its heritage from TLA+ in that it has no direct means of specifying an algorithm's pre- or post-conditions, invariants, and termination measures; such properties have to be expressed via assertions or via temporal formulas that refer to the value of the program counter.

The algorithm language with probably the longest tradition is VDM [17, 21] that supports in a typed framework with a rich set of types an expressive language for modeling both recursive and iterative algorithms with algorithms specified in terms of pre- and post-conditions. The supporting software Overture also provides an execution-based model checker similar to RISCAL (while a supporting proof tool is still in its infancy). However, there are some language glitches which make the use of the system somewhat cumbersome [24]: for instance, it is not possible to introduce named predicates in invariants; furthermore, invariants can be only used to constrain global state changes but not individual loops.

The language WhyML of the program verification environment Why3 environment [5, 32], while being a real programming language, can due its high-level also be considered as an algorithm language supporting pre- and postconditions, assertions, loop invariants, and termination

measures. However, due to its actual nature as a programming language, WhyML does not support nondeterministic constructions like TLA+/PlusCal, VDM, or RISCAL. WhyML programs can be executed via translation to the language OCaml and verified by various external theorem provers; runtime assertion checking and model checking are not supported. Similarly Dafny [18, 9] is a high-level programming language developed at Microsoft with built-in specification constructs. A program can be compiled to executable .NET code and verified via the SMT solver Z3. Also Dafny does not support nondeterministic constructions, runtime assertion checking or model checking.

Also for various more wide-spread programming languages such as C, Ada, Java, C#, extensions for specifications do exist. Considering only the Java world, around the Java Modeling Language (JML) [7, 15] an ecosystem of supporting tools have been developed, including runtime assertion checkers, extended static checkers, and full-fledged verifiers. However, all of these tools have to struggle with the complex semantics of an "industrial" programming language which is only partially covered by JML respectively the corresponding tools, partially also with the complexity of JML itself. For instance, the runtime assertion checking supported by the old "Common JML Tools" or the newer "OpenJML" toolset has to deal with the fact that not all quantified formulas expressible in JML are easily executable such that not all parts of a specification are necessarily considered in checks.

The thesis [24] has compared some of the languages/tools mentioned above (notably JML, TLA+/PlusCal, Alloy, VDM/Overture, Event-B/Rodin) and their suitability for modeling and verifying mathematical algorithms; in a nutshell, while none was considered as ideal, the system TLA+/PlusCal was judged as the best on for model checking (with VDM as an alternative for recursive algorithms, which are not supported by PlusCal).

5 Future Work

The current version of the RISCAL software allows to validate the correctness of mathematical algorithms and their formal annotations by executing respectively evaluating them on finite subsets of the generally infinite domains. Thus it can be e.g. detected that a loop invariant is too strong, i.e., does not hold for all inputs and loop iterations. However, this is only a first step towards an environment for the general verification of mathematical algorithms.

As a next step, we will develop a verification condition generator for the specification language. The conditions are parameterized over the unspecified domain bounds; for concrete values of these bounds, the conditions are decidable: they can be verified by evaluation (which is presumably slow) and by application of SMT solvers (which can be expected to be reasonably efficient). If such concrete instances are invalid, also the general condition is invalid and a proof need not be attempted. Thus we will also be able to detect that a loop invariant is too weak, i.e., that it does not describe the value space of the loop variables accurately enough to prove that the invariant holds in the post-state of the loop, even if it holds in the pre-state.

The expectation is that thus the formal annotations can be further validated to carry a subsequent proof-based verification of the algorithms for domains of arbitrary size. Ultimately, we will therefore connect the RISCAL software to a computer-aided interactive proof assistant such as the RISC ProofNavigator [26, 23] in order to perform general verifications.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B — System and Software Engineering*. Cambridge University Press, Cambridge, UK, May 2010. http://www.cambridge.org/at/academic/subjects/computer-science/programming-languages-and-applied-logic/modeling-event-b-system-and-software-engineering?format=HB.

[2] Alloy: a Language & Tool for Relational Models, March 2016. http://alloy.mit.edu/alloy.

[3] ANTLR, December 2016. http://www.antlr.org.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Germany, 2016. https://doi.org/10.1007/978-3-662-07964-5.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. http://proval.lri.fr/publications/boogie11final.pdf.

[6] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016. https://doi.org/10.6092/issn.1972-5787/4568.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, Berlin, Germany, 2006. Springer. https://doi.org/10.1007/11804192_16.

[8] The Coq Proof Assistant, January 2017. https://coq.inria.fr.

[9] Dafny: A Language and Program Verifier for Functional Correctness, January 2017. https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness.

[10] Event-B and the Rodin Platform, November 2015. http://www.event-b.org.

[11] Kokichi Futatsugi et al. CafeOBJ. Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan, January 2017. https://cafeobj.org.

[12] Joseph A. Goguen and Grant Malcom, editors. *Software Engineering with OBJ — Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*. Springer US, New York, NY, USA, 2000. https://doi.org/10.1007/978-1-4757-6541-0.

[13] Isabelle, November 2016. https://isabelle.in.tum.de.

[14] Daniel Jackson. *Software Abstractions — Logic, Language, and Analysis*. MIT Press, Cambridge, MA, USA, revised edition, November 2011. https://mitpress.mit.edu/books/software-abstractions.

[15] The Java Modeling Language (JML) Home Page, February 2013. http://www.jmlspecs.org.

[16] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. http://research.microsoft.com/users/lamport/tla/book.html.

[17] Peter Gorm Larsen et al. VDM-10 Language Manual. Overture Technical Report TR-001, Overture Tool, September 2016. http://raw.github.com/overturetool/documentation/master/documentation/VDM10LangMan/VDM10_lang_man.pdf.

[18] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic Programming and Automated Reasoning (LPAR-16), Dakar, Senegal, April 25–May 1, 2010*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, Berlin, Germany, 2010. https://www.microsoft.com/en-us/research/wp-content/uploads/2008/12/dafny_krml203.pdf.

[19] Tobias Nipkow and Gerwin Klein. *Concrete Semantics — With Isabelle/HOL*. Springer, Heidelberg, Germany, 2014. https://doi.org/10.1007/978-3-319-10542-0.

[20] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Germany, December 2016. https://isabelle.in.tum.de/dist/Isabelle2016-1/doc/tutorial.pdf.

[21] Overture Tool — Formal Modelling in VDM, January 2017. http://overturetool.org.

[22] The RISC ProgramExplorer, June 2016. https://www.risc.jku.at/research/formal/software/ProgramExplorer.

[23] The RISC ProofNavigator, September 2011. https://www.risc.jku.at/research/formal/software/ProofNavigator.

[24] Daniela Ritirc. Formally Modeling and Analyzing Mathematical Algorithms with Software Specification Languages & Tools. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, January 2016. https://www.risc.jku.at/publications/download/risc_5224/Master_Thesis.pdf.

[25] Kenneth Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, Columbus, OH, USA, 7th edition, 2012. http://highered.mheducation.com/sites/0073383090/information_center_view0/index.html.

[26] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, March 2009. https://doi.org/10.1007/s00165-008-0069-4 and https://www.risc.jku.at/people/schreine/papers/fac2008.pdf.

[27] Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 79:124–142, February 2012. Pedro Quaresma and Ralph-Johan Back (eds), Proceedings of the First Workshop on CTP Components for Educational Software (THedu'11), Wrocław, Poland, July 31, 2011, https://doi.org/10.4204/EPTCS.79.8 and https://www.risc.jku.at/research/formal/software/ProgramExplorer/papers/THeduPaper-2011.pdf.

[28] About SETL, January 2015. http://setl.org/setl.

[29] W. Kirk Snyder. The SETL2 Programming Language. Technical Report 490, Courant Institute of Mathematical Sciences, Computer Science Department, New York University, New York, NY, USA, 1990. https://archive.org/details/setl2programming00snyd.

[30] The Theorema System, January 2017. https://www.risc.jku.at/research/theorema/software.

[31] The TLA Home Page, November 2016. https://research.microsoft.com/en-us/um/people/lamport/tla/tla.html.

[32] Why3 — Where Programs Meet Provers, January 2017. http://why3.lri.fr/.

A The Software System

In the following sections, we describe the software that implements the RISCAL language.

A.1 Installing the Software

The README file of the installation is included below.

```
--------------------------------------------------------------------------------
README
Information on RISCAL.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Copyright (C) 2016-, Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria, http://www.risc.jku.at

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
--------------------------------------------------------------------------------

RISC Algorithm Language (RISCAL)
--------------------------------
http://www.risc.jku.at/research/formal/software/RISCAL

This is the RISC Algorithm Language (RISCAL), a specification language and
associated software system for describing mathematical algorithms, formally
specifying their behavior based on mathematical theories, and validating the
correctness of algorithms, specifications, and theories by execution/evaluation.

This software has been developed at the Research Institute for Symbolic
Computation (RISC) of the Johannes Kepler University (JKU) in Linz, Austria. It
is freely available under the terms of the GNU General Public License, see file
COPYING. RISCAL runs on computers with x86-compatible processors supporting Java
8 and the Standard Widget Toolkit (SWT); it has been developed and tested on a
computer with the GNU/Linux operating system and a x86-64 processor. For
learning how to use the software, see the file "main.pdf" in the directory
"manual"; examples can be found in the directory "spec".

Please send bug reports to the author of this software:

  Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
  http://www.risc.jku.at/home/schreine
  Research Institute for Symbolic Computation (RISC)
```

```
  Johannes Kepler University
  A-4040 Linz, Austria

A Virtual Machine with RISCAL
-----------------------------
On the RISCAL web site, you can find a virtual GNU/Linux machine that has RISCAL
preinstalled. This virtual machine can be executed with the free virtualization
software VirtualBox (http://www.virtualbox.org) on any computer with an
x86-compatible processor running under Linux, MS Windows, or MacOS. You just
need to install VirtualBox, download the virtual machine, and import the virtual
machine into VirtualBox.

This may be for you the easiest option to run the software; if you choose this
option, see the web site for further instructions.

The Distribution
----------------
The distribution has the following contents:

  README     ... this file
  COPYING    ... the GNU General Public Licence Version 3
  CHANGES    ... the version history of the software
  etc/
    RISCAL   ... the execution script
    run*     ... examples of server execution scripts
  lib/
    *.jar          ... the Java compiled libraries
    swt32/swt.jar ... the SWT library for GNU/Linux and x86-32 processors
    swt64/swt.jar ... the SWT library for GNU/Linux and x86-64 processors
  doc/
    main.pdf ... the manual
  spec/
    *.txt    ... sample specifications
  src/
    */*.java ... the Java source code

Installation
------------
Copy the file etc/RISCAL to a directory in your PATH and adapt the variable JAVA
to point to the Java executable "java". Adapt LIB to point to the directory
"lib" of the distribution and adapt $LIB/swt64 to point to the directory with
the SWT library for your operating system and processor.

You should then be able to execute

  RISCAL &

Third Party Software
--------------------
RISCAL uses the following open source programs and libraries. Most of this is
already included in the distribution, but if you want or need, you can download
the source code from the denoted locations (local copies are available on the
RISCAL web site) and compile it on your own. Many thanks to the respective
developers for making this great software freely available!
```

```
Java Development Kit 8 (or higher)
http://www.oracle.com/technetwork/java/javase/downloads/index.html
-----------------------------------
Go to the "Downloads" section to download the JDK.

ANTLR 4.5
http://www.antlr.org
--------------------
This is a framework for constructing parsers and lexical analyzers used for
processing the programming/specification language of the RISC ProgramExplorer.
Go to the "Download" section to download the latest 4.* version of the library.

On a Debian 9 "stretch" GNU/Linux distribution, just install the package "antlr4"
by executing (as superuser) the command

  apt-get install antlr4

The Eclipse Standard Widget Toolkit 4.7
http://www.eclipse.org/swt
--------------------------------------
This is a widget set for developing GUIs in Java.

Go to section "Stable" and download the version "Linux (x86/GTK2)" (if you use
a 32bit x86 processor) or "Linux (x86_64/GTK 2)" (if you use a 64bit x86
processor).

For the builtin "Help" to work properly, WebKitGTK 1.2.0 or newer must
be installed; e.g. on a Debian 9 "stretch" GNU/Linux distribution, just install
the package "libwebkitgtk-3.0-0" by executing (as superuser) the command

  apt-get install libwebkitgtk-3.0-0

Tango Icon Library 0.8.90
http://tango.freedesktop.org
----------------------------
Go to the section "Base Icon Library", subsection "Download", to download
the icons used in RISCAL.

------------------------------------------------------------------------------
End of README.
------------------------------------------------------------------------------
```

A.2 Running the Software

The RISCAL software is intended to be used in interactive mode by executing the shell script

```
RISCAL &
```

which prints out the copyright message

```
RISC Algorithm Language 1.0 (March 1, 2017)
http://www.risc.jku.at/research/formal/software/RISCAL
```

```
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCAL -h" to see the available command line options.
-----------------------------------------------------------------
```

However, if we execute (as indicated in this message)

```
RISCAL -h
```

we get the following output which displays the following options:

```
RISCAL [ <options> ] [ <path> ]
<path>: path to a specification file
<options>: the following command line options
-h: print this message and exit
-s <T>: run in server mode with T threads
-nogui: do not use graphical user interface
-p: print the parsed specification
-t: print the typed specification with symbols
-v <I>: use integer <I> for all external values
-nd: nondeterministic execution is allowed
-e <F>: execute parameter-less function/procedure F
```

Most of these options were used in the initial development of the software and are preserved mainly for historical reasons. The main exception is the option `-s` by which it is possible to execute the software in "server mode", e.g. as

```
RISCAL -s 4
```

which indicates that the software shall run as a server with 4 threads. It the prints a line such as

```
amir.risc.jku.at 41459 27pn3agrgjc5c1mcu14r4rcr8n
```

where the first string represents the Internet name of the machine running the software, the second word represents the number of the port where the server is listening for a connection request and the last word represents a one-time password for authenticating the connection request. This information can be used by another RISCAL process that runs in "Distributed" mode to connect to the server process and forward computations to the server. See Appendix A.4 for more details.

A.3 The User Interface

Main Window The user interface depicted in Figure 2 is divided into two parts. The left part mainly embeds an editor panel with the current specification. The right part is mainly filled by an output panel that shows the output of the system when analyzing the specification that is currently loaded in the editor. The top of both parts contains some interactive elements for controlling the editor respectively the analyzer.

Figure 2: The RISCAL User Interface (Enlarged)

Menu Bar The bar at the top of the window holds three menus:

File Option "New" starts the editing of a new specification; while "Open. . . " opens an existing one. "Save" saves the current specification to disk; "Save As. . . " saves it under a new name to be chosen. "Quit" terminates the software.

Edit Option "Undo" reverts the last editing operation while "Redo" performs it again. Options "Bigger Font" and "Smaller Font" allow to resize the fonts used for the display of text in the editor and in the output panel.

Help Option "Online Manual" opens a web browser with an online version of this manual; "About" opens a window with a copyright message.

Most menu entries have keyboard shortcuts which are displayed in the corresponding menus. The file actions "New", "Open. . . " and "Save" are also bound to three buttons  ,  , and  displayed at the top of the editor panel.

The translation respectively execution of a specification is controlled by various buttons, check boxes, and input fields on the top of the right panel; moving the mouse cursor over a button displays a corresponding description.

Executing a Specification We have the following buttons:

 **[Process Specification]** This triggers the re-processing of a specification. However, since a specification is automatically processed when it is loaded or saved after editing or before executing the specification after some of the "Translation" options below have been changed, there is usually no reason to explicitly trigger the processing.

 **[Start Execution]** This starts execution respectively model checking of that operation that has been selected in the "Operation" menu described below.

 **[Stop Execution]** This stops any ongoing execution triggered by the "Start Execution" button.

 **[Clear Output]** This clears the output panel. The output displayed in that panel is automatically truncated when it gets too long; to avoid truncation, the panel may me explicitly cleared by this button.

 **[Start Logging]** This opens a file selection dialog by which the name and directory of a file may be specified to which the content of the output panel is logged for later investigation.

 **[Stop Logging]** This stops logging the content of the output panel triggered by the "Start Logging" button.

 **[Reset System]** This resets the software to the initial state; it clears the editor window and resets all options to their defaults.

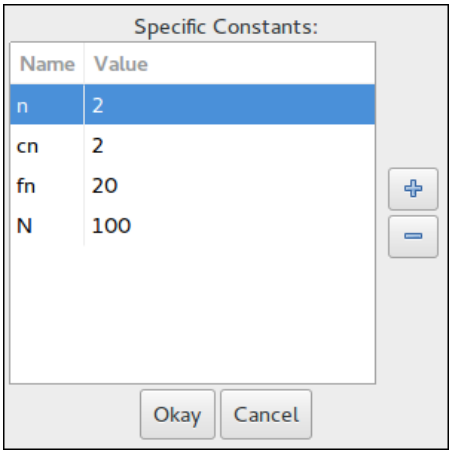

Figure 3: The "Other Values" Window

Configuring the Translation The label "Translation" displays all options/values that affect the processing of the specification to an executable representation (whenever one of these is changed, the specification is automatically re-processed before execution):

Nondeterminism If this option is *not* selected, the specification is executed in a deterministic mode where value choices performed by a nondeterministic language construct such as `choose` are resolved by choosing a single eligible value, respectively by aborting, if no such value exists. However, if this option is selected, the specification is executed in a nondeterministic mode where each choice splits the execution into multiple branches each of which is executed in turn. If an execution contains multiple subsequent choices, this yields exponentially many execution branches whose execution takes exponentially more time than executing the single branch chosen in deterministic mode. Since the deterministic mode also produces a more efficient executable version of the specification, this option should be only used with care.

Default Value In this field, the user can define the value (a non-negative integer in decimal representation) that is given to every unspecified natural number constant in the specification. Actually, this value is used for a constant c only if the "Other Values" table explained below does not give a specific value for c.

Other Values If this button is pressed, the window displayed in Figure 3 pops up where values for specific natural number constants can be given. If for a constant c used in the specification no value is provided, the "Default Value" explained above is chosen.

Configuring the Execution The label "Execution" displays all options/values that affect the execution of a specification (it is not necessary to re-process a specification whenever one of these is changed):

Silent If this option is *not* selected, every application of the operation selected in the "Operation" menu explained below yields some output (the value of the function and some execution

statistics). If this option is selected, only infrequently (every 2s or so) statistics about the applications executed so far is printed.

Inputs If this field is empty, the operation selected in the "Operation" menu is applied to all argument tuples from the domain of the operation. If this field is not empty, it must contain a natural number N (a non-negative integer in decimal representation). Then the operation is applied to at most N argument tuples (if "Parallelism" is applied as explained below, N is not a sharp bound but only a guideline; actually some more applications are possible).

Per Mille If this field is not empty, it must contain a natural number N (a non-negative integer in decimal representation) with $0 \leq N \leq 1000$. Then every possible argument tuple for the operation selected in the "Operation" menu is chosen with a probability of $N/1000$; as a consequence the operation is applied to only a fraction of approximately $N/1000$ of all possible inputs.

Branches If this field is not empty, it must contain a natural number N (a non-negative integer in decimal representation). Then, if the option "Nondeterminism" explained above is selected, at most N values are chosen in a nondeterministic language construct such as `choose`, i.e., the execution splits into at most N branches at each choice point.

Configuring Parallelism The label "Parallelism" groups all options/values that speed up the model checker by multi-threaded and/or distributed parallel execution:

Multi-Threaded If this option is selected, the applications of the operation selected in the "Operation" menu are performed by multiple (at least 2) threads in parallel, which may significantly speed up the execution.

Threads If this field is empty, multi-threaded execution proceeds with 2 threads. If this field is not empty, it must contain a natural number N (a non-negative integer in decimal representation). Then multithreaded execution proceeds with $\max\{N, 2\}$ threads.

Distributed If this option is selected, the model checker applies (possibly in addition to multiple threads as described above) additional RISCAL processes which may be also executed on remote computers. The creation of these processes is configured in the "Servers" menu:

Servers By pressing this button, the window depicted in Figure 4 pops up. The text fields in this window contains a list of commands, one command (possibly with arguments) per line. Each of the listed commands is executed by the software when the "Distributed" option explained above is selected to startup another RISCAL process; the output of this command gives the current process the information it needs to connect to the other process which may be executed on another computer, e.g., a high performance compute server. More information on this topic is given in Appendix A.4.

A.4 Distributed Execution

The RISCAL software may be executed in a "Distributed" mode where the "client process" running with the graphical user interface on the user's local computer connects to one more

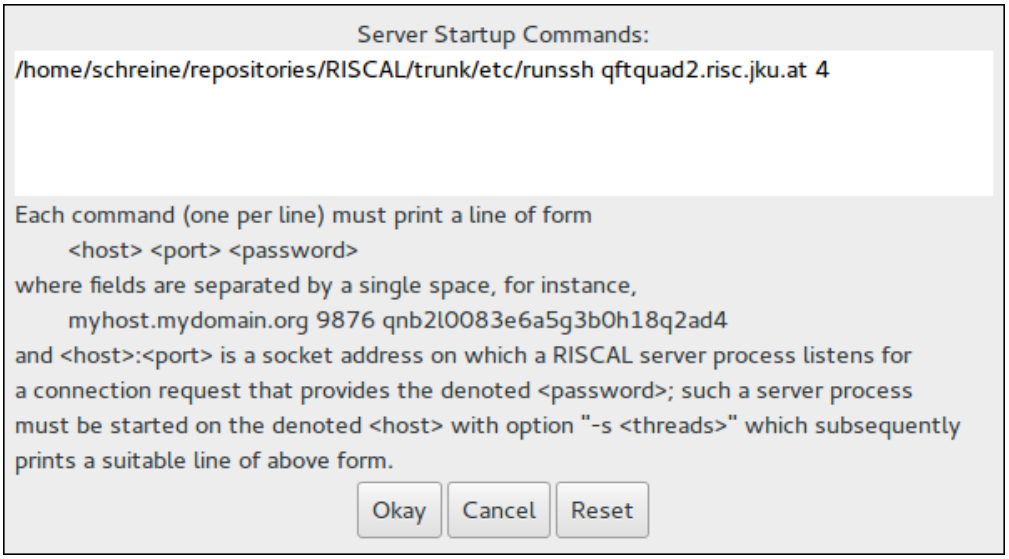

Figure 4: The "Servers" Window

"server processes" that potentially run on remote computers (e.g., high-performance servers); the client process then forwards part of the model checking work to the server processes.

For this purpose, every command listed in the "Server" window depicted in Figure 4 on page 35 must start an instance of the RISCAL software with the option `-s` T which indicates that the software is executed in "server" mode with T threads. A possible such command is

```
java -cp LIB/* riscal.Main -s 4
```

where `LIB` is to be replaced by the absolute path of the directory that contains the `.jar` files of the RISCAL software on the computer that runs the server process; actually from this library only the files `antlr4.jar` and `riscal.jar` are required to run the server. The process then prints to the standard output a line of form *host port password* e.g.

```
host.mydomain.org 9876 qnb2l0083e6a5g3b0h18q2ad4
```

where the first string is the internet name/address of the host where the process is executed, the second string denotes the number of the port on which the process listens for connection requests and the last string is a randomly generated one-time password. The current process uses this information to connect to the remote process on that host via the denoted port and provides the password to prove that it is entitled to the connection.

If the server process is to run on the same computer under the GNU/Linux operating system, the command may be wrapped into a shell script `runsh` that may be e.g. invoked as

```
/PATH/runsh 4
```

and whose content is as follows:

```
#!/bin/bash
# uses bash-specific process substitution below
if [ $# -ne 1 ] ; then
  echo "usage: runsh <threads>"
  exit
fi
THREADS=$1
JAVA=/software/java8/bin/java
LIB=/home/schreine/repositories/RISCAL/trunk/lib
head -1 <( $JAVA -cp "$LIB/*" -Xmx2G -Xms1G riscal.Main -s $THREADS )
```

where the variable `JAVA` has to be replaced by the absolute path of the `java` runtime engine and the variable `LIB` has to be replaced by the path of the RISCAL installation.

If the server process is to run on another computer under the GNU/Linux operating system to which we may connect by the "Secure Shell" (SSH) software, the command may be wrapped into a shell script `runssh` that may be e.g. invoked as

```
/PATH/runssh host.mydomain.org 4
```

and whose content is as follows:

```
#!/bin/bash
# uses bash-specific process substitution below
if [ $# -ne 2 ] ; then
  echo "usage: runssh <host> <threads>"
  exit
fi
HOST=$1
THREADS=$2
JAVA=/zvol/formal/java8_64/bin/java
LIB=/home/schreine/repositories/RISCAL/trunk/lib
DIR=/home/schreine/tmp/riscal
head -1 <( ssh $HOST $JAVA -cp \"$DIR/*\" -Xmx2G -Xms1G riscal.Main -s $THREADS )
```

where again the variable `JAVA` has to be replaced by the absolute path of the `java` runtime engine and the variable `LIB` has to be replaced by the path of the RISCAL installation. The SSH software must be then configured (by the use of a certificate) such that remote login is possible without a password, e.g.

```
ssh host.mydomain.org echo hi
```

must print "hi" without asking for a password.

B The Specification Language

In the following sections, we describe the specification language.

B.1 Lexical and Syntactic Structure

On the lowest level, a RISCAL specification is a file encoded in UTF-8 format. RISCAL uses several Unicode characters that cannot be found on keyboards, but for each such character there exists an equivalent string in ASCII format that can be typed on a keyboard. While the RISCAL grammar supports both alternatives, the use of the Unicode characters yields much prettier specifications and is thus recommended.

Fortunately the RISCAL editor can be used to translate the ASCII string to the Unicode character by first typing the string and then (when the editor caret is immediately to the right of this string) pressing <Ctrl>-#, i.e, the Control key and simultaneously the key depicting #. Also later such textual replacements can be performed by positioning the editor caret to the right of the string and pressing <Ctrl>-#. The current table of replacements is as depicted in Figure 5.

A specification file may include two kinds of comments which are ignored when processing the file:

- Comments starting with // and ranging till the end of the file.
- Comments starting with /* and ending with */ (such comments must not be nested).

Likewise white space characters (blanks, tabulators, new lines, returns, form feeds) are ignored.

The syntactical grammar of RISCAL uses the following kinds of terminal symbols:

- An *identifier* ⟨*ident*⟩ is a non-empty sequence of (lower and upper case) ASCII letters, decimal digits, and the underscore character _ starting with a letter, e.g. `pos0`.
- A *decimal number literal* ⟨*decimal*⟩ is a non-empty sequence of decimal digits, e.g. `123`.
- A *string literal* ⟨*string*⟩ is a sequence of characters of form `"..."` where the characters between the double quotes may include the escape sequences \\ (backslash), \" (double quote), and \n (new line), e.g. `"output: \"{1}\"\n\"{2}\""`.

The RISCAL grammar (for both lexical analysis and syntax analysis) is formally defined in Appendix B.7 as an ANTLR4 grammar file. The following sections explain the various kinds of phrases in a form that is modeled after that syntax but edited for readability.

In the subsequent presentation, a rule

⟨*domain*⟩ ::= ⟨*alternative1*⟩ | ... | ⟨*alternativeN*⟩

introduces a syntactic domain with *N* construction alternatives. Within each alternative, we have the following meta-syntax:

- A phrase in teletype letters like `Int` or `[` denotes a literal token.
- Also special Unicode characters like ℤ are literals that stand for themselves.
- (⟨*phrase1*⟩ | ... | ⟨*phraseN*⟩) denotes one of the phrases ⟨*phrase1*⟩, . . . , ⟨*phraseN*⟩.
- (⟨*phrase*⟩)? denotes 0 or 1 occurrence of ⟨*phrase*⟩.

ASCII String	Unicode Character
`Int`	$\mathbb{Z}$
`Nat`	$\mathbb{N}$
`:=`	$:=$
`true`	$\top$
`false`	$\bot$
`~`	$\neg$
`/\`	$\wedge$
`\/`	$\vee$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`forall`	$\forall$
`exists`	$\exists$
`sum`	$\sum$
`product`	$\prod$
`~=`	$\neq$
`<=`	$\leq$
`>=`	$\geq$
`*`	$\cdot$
`times`	$\times$
`{}`	$\emptyset$
`intersect`	$\cap$
`union`	$\cup$
`Intersect`	$\bigcap$
`Union`	$\bigcup$
`isin`	$\in$
`subseteq`	$\subseteq$
`<<`	$\langle$
`>>`	$\rangle$

Figure 5: ASCII Strings and their Equivalent Unicode Characters

- (⟨*phrase*⟩)* denotes 0 or more occurrences of ⟨*phrase*⟩.
- (⟨*phrase*⟩)+ denotes 1 or more occurrences of ⟨*phrase*⟩.

Furthermore, parentheses (. . .) may be used to group phrases.

B.2 Specifications and Declarations

A specification is a sequence of declarations:

⟨*specification*⟩ ::= (⟨*declaration*⟩)*

A declaration introduces at least one name into the environment; the name may be subsequently referenced. The name space is divided into three categories of names:

- Types
- Values (constants, parameter-less predicates and theorems)
- Functions (functions, parameterized predicates and theorems, procedures)

It is an error to declare in the same category two entities with the same name (but it is okay, if entities in different categories have the same name). As an exception, different functions may have the same name, if they differ in the number or types of their parameters (i.e., function names may be "overloaded").

In the following we describe the domain

⟨*declaration*⟩ ::= . . .

of declarations.

B.2.1 Types

Grammar

`type` ⟨*ident*⟩ = ⟨*type*⟩ `;`
`rectype` `(` ⟨*exp*⟩ `)` ⟨*ritem*⟩ (`and` ⟨*ritem*⟩)* `;`
`enumtype` ⟨*ritem*⟩ `;`

⟨*ritem*⟩ ::= ⟨*ident*⟩ = ⟨*rident*⟩ (`|` ⟨*rident*⟩)*
⟨*rident*⟩ ::= ⟨*ident*⟩ (`(` ⟨*type*⟩ (`,` ⟨*type*⟩)* `)`)?

Description We have the following definitions of types:

- A type definition `type` *id* = *T* `;` introduces a name *id* as a synonym for type *T*.

- A recursive type defintion `rectype(`n`)` id = c | f`(`T_1`,...,`T_n`)` | ... `;` introduces a new type id with constant id`!`c and constructor id`!`f, a function with parameters of types $T_1, \ldots, T_n$ and a result of type id. Different constants denote different values, applications of different constructors yield different results, and applications of the same constructor to different arguments yield different results.

  The type name id may appear as a (subtype of) parameter type of a constructor; thus arbitrarily deep nestings of constructor applications are possible. However, the number of nested constructor applications must not exceed the bound set by the constant expression $n \geq 0$. In particular, if $n = 0$, then only constant values may be used; if $n = 1$, only applications of constructors to constants are allowed. This constraint is checked at runtime such that execution is aborted, if it is violated.

  A recursive type definition `rectype(`n`)` id_1 = ... `and` ... `and` id_n = ... `;` introduces n types $id_1, \ldots, id_n$ that may mutually recursively depend on each other, i.e., one type may appear as a (subtype of) a parameter type of a constructor of another type. The bound n applies to all types in common: not more than n nested constructor applications are allowed, even if these are applications of constructors of different types in the recursive type definition.

- An enumerated type definition `enumtype` id = c_1 | ... | c_n `;` is an abbreviation for the recursive type definition `rectype(0)` id = c_1 | ... | c_n `;`.

Pragmatics The bound n in a recursive type definition `rectype(`n`)` id = ... ensures that (like all other types) also a recursive type has only finitely many values.

B.2.2 Values

Grammar

`val` ⟨*ident*⟩ `:` (ℕ | `Nat`) `;`
`val` ⟨*ident*⟩ (`:` ⟨*type*⟩)? = ⟨*exp*⟩ `;`
`pred` ⟨*ident*⟩ (⇔ | <=>) ⟨*exp*⟩ `;`
`theorem` ⟨*ident*⟩ (⇔ | <=>) ⟨*exp*⟩ `;`

Description We have the following definitions of "values" (which also encompass "predicates" and "theorems"):

- `val` n`:` ℕ (alternatively, `val` n`:` `Nat`) introduces a new natural number constant n. The value of this constant is not defined in the specification itself but chosen externally (before the specification is processed). The remainder of the specification is thus processed for one particular choice of the constant value.

- `val` c`:` T = e`;` introduces a new constant c and binds it to the value of e; the type of id is the type of e. If the optional type T is given, e must be of type T.

- `pred` $p \Leftrightarrow b$`;` (where the symbol ⇔ can be alternatively written as <=>) defines a "predicate" p, i.e., a constant of type `Bool` and binds it to the value of formula b (an expression of type `Bool`).

- `theorem` $t \Leftrightarrow b$`;` (where the symbol ⇔ can be alternatively written as <=>) introduces a "theorem" t, i.e., a predicate whose value is "true". Here b must be a formula with value "true"; if its value is "false", execution aborts.

Pragmatics External constants may serve as bounds in type definitions respectively may be used to compute such bounds in constant expressions.

Value (also predicate or theorem) definitions are evaluated by the type checker, which may considerably delay the checking. An alternative to the definition of a value is the definition of a corresponding function (respectively parameterized predicate or theorem) with argument type `()` (i.e, type `Unit`); such a function is only evaluated when it is applied.

B.2.3 Functions

(`multiple`)? `fun` ⟨*ident*⟩ `(` (⟨*ident*⟩`:`⟨*type*⟩ (`,` ⟨*ident*⟩`:`⟨*type*⟩)*)? `)` `:` ⟨*type*⟩
((⟨*funspec*⟩)* '=' ⟨*exp*⟩)? `;`

(`multiple`)? `pred` ⟨*ident*⟩ `(` (⟨*ident*⟩`:`⟨*type*⟩ (`,` ⟨*ident*⟩`:`⟨*type*⟩)*)? `)`
((⟨*funspec*⟩)* ('⇔' | '⟨=⟩') ⟨*exp*⟩)? `;`

(`multiple`)? `theorem` ⟨*ident*⟩ `(` (⟨*ident*⟩`:`⟨*type*⟩ (`,` ⟨*ident*⟩`:`⟨*type*⟩)*)? `)`
((⟨*funspec*⟩)* ('⇔' | '⟨=⟩') ⟨*exp*⟩)? `;`

(`multiple`)? `proc` ⟨*ident*⟩ `(` (⟨*ident*⟩`:`⟨*type*⟩ (`,` ⟨*ident*⟩`:`⟨*type*⟩)*)? `)` `:` ⟨*type*⟩
((⟨*funspec*⟩)* `{` (⟨*command*⟩)* (`return` ⟨*exp*⟩ `;`)? `}`)?

⟨*funspec*⟩ ::= `requires` ⟨*exp*⟩ `;` | `ensures` ⟨*exp*⟩ `;` | `decreases` ⟨*exp*⟩ (`,` ⟨*exp*⟩)* `;`

Description We have the following definitions of "functions" (which encompass "parameterized predicates", "parameterized theorems", and "procedures"):

- `fun` f`(`p_1`:`T_1`,`$\ldots$`,`p_n`:`T_n`)`$:T = e$ `;` introduces a function f with n parameters $p_1, \ldots, p_n$ of types $T_1, \ldots, T_n$, respectively; the value of the function is defined by the expression e of type T.

- `pred` p`(`p_1`:`T_1`,`$\ldots$`,`p_n`:`T_n`)` $\Leftrightarrow b$ `;` (where the symbol ⇔ can be alternatively written as <=>) introduces a predicate p with n parameters $p_1, \ldots, p_n$ of types $T_1, \ldots, T_n$, respectively; the value of the predicate is defined by the formula b (an expression of type `Bool`).

- `theorem` t`(`p_1`:`T_1`,`$\ldots$`,`p_n`:`T_n`)` $\Leftrightarrow b$ `;` (where the symbol ⇔ can be alternatively written as <=>) introduces a theorem t, i.e., a predicate for which we claim that all applications yield truth value "true". If an application yields "false", this application aborts.

- `proc` p`(`p_1`:`T_1`,...,`p_n`:`T_n`):`T `{` c_1`;` ... `;` c_n`; return` e`; }` introduces a procedure p with n parameters $p_1, \ldots, p_n$ of types $T_1, \ldots, T_n$, respectively; the value of the procedure is computed by executing the commands $c_1, \ldots, c_n$ in sequence and evaluating the expression e of type T in the resulting store; the value of e is the value of the procedure. Parameters are local constants, their values thus cannot be changed by the command execution.

  `proc` p`(`p_1`:`T_1`,...,`p_n`:`T_n`):() {` c_1`;` ... `;` c_n`; }` introduces a procedure p with result type `()` (i.e., type `Unit`); this procedure executes commands $c_1, \ldots, c_n$ in sequence and then returns the value `()`.

A function definition yields an error, if a function with the same name, the same number of arguments, and the same argument types has been already defined (but it is okay to define multiple functions with the same name, if they differ in the number or types of arguments, i.e. function names may be "overloaded").

A function is visible from the point of the definition on, including its defining expression respectively command sequence; thus a function may apply itself recursively.

A function may be first declared (by omitting the defining expression respectively command sequence) and later defined. From the point of the declaration on, the function is known and may be applied by other functions. Thus functions may apply themselves mutually recursively.

If the value of a (mutually) recursive function is not uniquely determined from its arguments (because the function makes choices to compute its result), the function must be tagged with the keyword `multiple`; if this is omitted, the processing of the specification reports an error.

A function definition may be annotated by multiple "preconditions", i.e., clauses of form `requires` b`;` where b is a formula that may refer to the parameters of the function. The function may be only applied to arguments for which the evaluation of all preconditions (where the formal parameters are substituted by the actual arguments) yields "true"; if some precondition yields "false", execution is aborted.

A function definition may be annotated by multiple "postconditions", i.e., clauses of form `ensures` b`;` where b is a formula that may refer to the parameters of the function and to the special constant `result` whose type is the result type of the function. The function may only return values for which the evaluation of all postconditions (where the formal parameters are substituted by the actual arguments and `result` is bound to the result value of the function) yields "true"; if some precondition yields "false", execution is aborted.

A function definition may be annotated by multiple "termination measures", i.e., clauses of form `decreases` e with integer expression e. This expression is evaluated after before/after every (directly or indirectly) recursive application of the function; if the value of e becomes negative or is not less than the value in the last application, execution aborts, otherwise the clause has no effect. The existence of at least one such clause thus ensures that a recursive function eventually terminates. In general, every clause may have the form `decreases` e_1`,...,`e_n with $n \geq 1$ in which case no e_i with $1 \leq i \leq n$ may become negative and the sequence of values must be decreased by every recursive function application with respect to the "lexicographical order" of integer sequences (there must exist some position i with $1 \leq i \leq n$ in the sequence such that e_i is decreased and all e_j with $1 \leq j < i$ remain the same).

Pragmatics The externally visible behavior of a procedure is that of a function in that it returns a value but otherwise has no side effect (apart from potentially printing output which however cannot affect the computation). This is because the store of a procedure is local to the procedure; there is no "global store" that might be affected by the execution of the procedure; also the values of variables passed as arguments to a procedure cannot be changed by the procedure.

The keyword `multiple` simplifies the translation process; it could be made superfluous by a more powerful static analysis than currently implemented.

The clause `return` *e* is not a general command may may only appear at the end of a procedure definition; this simplifies the later development of a verification calculus.

B.3 Commands

In this section, we are going to describe the domain

⟨*command*⟩ ::= ... | ⟨*exp*⟩ | `{` (⟨*command*⟩)* `}` | `;`

of *commands*, i.e., syntactic phrases that cause effects but have no values. However, also every expression of type `()` (i.e., type `Unit`) can serve as a command, in particular applications of functions with return type `()`. Furthermore, a sequence of commands may be grouped by curly braces `{...}` into a single command. Finally, a command can be empty (indicated by the sole occurrence of the command terminator `;`), which allows redundant occurrences of `;`.

B.3.1 Declarations and Assignments

Grammar

`val` ⟨*ident*⟩ (`:` ⟨*type*⟩)? (:= | `:=` | =) ⟨*exp*⟩ `;`
`var` ⟨*ident*⟩ `:` ⟨*type*⟩ ((:= | `:=` | =) ⟨*exp*⟩)? `;`
⟨*ident*⟩ (⟨*sel*⟩)* (:= | `:=` | =) ⟨*exp*⟩ `;`

⟨*sel*⟩ ::= `[` ⟨*exp*⟩ `]` | `.` ⟨*decimal*⟩ | `.` ⟨*ident*⟩

Description

- A declaration `val` *id*`:`*T* := *e* (where the definition symbol := may be alternatively written as the two characters `:=` or the single character =) introduces a local constant of name *id* and gives it the value of expression *e*. If the optional type *T* is given, the type of *e* must be *T*. It is an error, if another local constant (or variable) with name *id* has been already declared (but it is okay, if the declaration overshadows a global value declaration).
- A declaration `var` *id*`:`*T* := *e* (where := may be alternatively written as `:=` or =) introduces a variable of name *id* and type *T*. If the optional expression *e* is given, the type of *e* must be *T* and the variable is initialized with that value; otherwise, the value of the variable is undefined. It is an error, if another variable (or local constant) with name *id* has been already declared (but it is okay, if a global value declaration is overshadowed).

- A variable assignment *id* := *e* (where := may be alternatively written as `:=` or =) assigns to a previously declared variable *id* the value of expression *e*. It is an error, if no variable of name *id* has been declared or if the type of *e* is not the type given in the declaration.

  An array/map assignment *a*`[`*i*`]` := *e* is just an abbreviation for the plain variable assignment *a* := *a* `with [`*i*`] =` *e* which assigns to variable *a* the new array denoted by the array update expression *a* `with [`*i*`] =` *e*, i.e., an array that is a duplicate of *a* except that it holds at *i* the value *e*. Likewise, a tuple assignment *t*`.`*d* := *e* is an abbreviation for the plain assignment *t* := *t* `with .`*d* `=` *e* and a record assignment *r*`.`*id* := *e* is an abbreviation for the plain assignment *r* := *r* `with .`*id* `=` *e*. The component selectors for the various kinds of data structures may be also combined, e.g. *a*`[`*i*`].`*id* := *e* is an abbreviation for:

  a `:=` *a* `with [`*i*`] :=` `(`*a*`[`*i*`] with .`*id* `=` *e*`)`

Pragmatics Arrays/maps, tuples, and records have "value semantics", i.e. an element assignment like *a*`[`*i*`]` := *e* creates a new array and assigns it to variable *a*; the original array is not modified in any way. Thus the code

```
var a:Array[10,ℕ[3]] := Array[10,ℕ[3]](0);
var b:Array[10,ℕ[3]] := a;
a[0] := 1;
print b[0];
```

prints `0`, not `1`.

B.3.2 Choices

Grammar

`choose` ⟨*qvar*⟩ `;`
`choose` ⟨*qvar*⟩ `then` ⟨*command*⟩ `else` ⟨*command*⟩

Description These commands introduce new constants whose values are chosen from a finite set of possibilities. If RISCAL is executed in "deterministic" mode, for each constant an arbitrary value is chosen; if RISCAL is executed in "nondeterministic" mode, all values are chosen in turn (resulting in multiple computation branches that are executed in turn).

`choose` *q* `;` introduces new local constants whose names are those of the quantified variables in *q* and binds them to chosen values. In deterministic execution, if no choice is possible, the computation is aborted.

`choose` *q* `then` c_1 `else` c_2 either executes command c_1 or it executes command c_2. The execution of c_1 must take place in a context that contains new local constants whose names are those of the quantified variables in *q* and which are bound to chosen values. Thus, if no choice is possible, command c_2 must be executed.

Pragmatics The constants introduced by `choose` *q* `;` are visible in the subsequent commands. The constants introduced by `choose` *q* `then` c_1 `else` c_2 are only visible in command c_1; they are not visible afterwards.

B.3.3 Conditionals

Grammar

`if` ⟨*exp*⟩ `then` ⟨*command*⟩
`if` ⟨*exp*⟩ `then` ⟨*command*⟩ `else` ⟨*command*⟩

`match` ⟨*exp*⟩ `with` `{` (⟨*pattern*⟩ `->` ⟨*command*⟩)+ `}`
⟨*pattern*⟩ ::= ⟨*ident*⟩ | ⟨*ident*⟩ `(` ⟨*param*⟩ (`,` ⟨*param*⟩)* `)` | `_`

Description We have the following kinds of conditional statements:

- The one-sided conditional statement `if` b `then` c evaluates formula b (an expression of type `Bool`). If this evaluation yields "true", the statement executes command c, otherwise it has no effect.
- The two-sided conditional statement `if` b `then` c_1 `else` c_2 evaluates formula b. If this yields "true", the statement executes command c_1, otherwise it executes command c_2.
- The matching command `match` e `with` `{` p_1`->`c_1`;` ... `;` p_n`->`c_n`;}` attempts to "match" the value of e (which must be of some recursive type T) to the patterns $p_1, \ldots, p_n$. Each pattern can be either the name *id* of a constant of type T or an application *id*`(`p_1`,` ... `,`p_n`)` of a constructor *id* of type T or the "wildcard" pattern `_`. A match succeeds if the value of e is the denoted constant or the result of an application of the denoted constructor or if the pattern is the wildcard `_`.

  The matches are attempted in the stated order of patterns; the first successful match of the value of e to some pattern p_i determines the effect of the whole command in that the command c_i is executed. If the match succeeds for a pattern p_i = *id*`(`p_1`,` ... `,`p_n`)`, the parameters $p_1, \ldots, p_n$ receive the arguments to which constructor *id* was applied to yield the value of e; these parameters can be consequently referenced in c_i. If there is no successful match, the effect of the command is undefined (i.e. the computation aborts).

B.3.4 Loops

Grammar

`while` ⟨*exp*⟩ `do` (⟨*loopspec*⟩)* ⟨*command*⟩
`do` (⟨*loopspec*⟩)* ⟨*command*⟩ `while` ⟨*exp*⟩ `;`
`for` ⟨*command*⟩ `;` ⟨*exp*⟩ `;` ⟨*command*⟩ `do` (⟨*loopspec*⟩)* ⟨*command*⟩

`for` ⟨*qvar*⟩ `do` (⟨*loopspec*⟩)* ⟨*command*⟩
`choose` ⟨*qvar*⟩ `do` (⟨*loopspec*⟩)* ⟨*command*⟩

⟨*loopspec*⟩ ::= `invariant` ⟨*exp*⟩ `;` | `decreases` ⟨*exp*⟩ (`,` ⟨*exp*⟩)* `;`

Description We have the following kinds of loops:

- `while` *b* `do` *c* evaluates formula *b* (an expression of type `Bool`); if its value is "false", the loop terminates. Otherwise it executes command *c* and repeats its behavior with the next evaluation of *b*).

  Variables and constants introduced in *c* are only visible in *c*.

- `do` *c* `while` *b* executes command *c* and then evaluates formula *b*; if its value is "false", the loop terminates. Otherwise it repeats its behavior with the next execution of *c*.

  Variables and constants introduced in *c* are only visible in *c*.

- `for` c_1`;` *b*`;` c_2 `do` c_3 first executes command c_1. It then evaluates formula *b*; if its value is "false", the loop terminates. Otherwise it executes command c_3 and then command c_2 and then repeats its behavior with the next evaluation of *b*.

  Variables and constants introduced in c_1 are visible in the whole command (but not outside the command). Variables and constants introduced in c_2 are only visible in c_2. Variables and constants introduced in c_3 are only visible in c_3.

- `for` *q* `do` *c* executes command *c* in the contexts arising from all possible choices of values for the quantified variables in *q*, i.e., in contexts that contain constants for the quantified variables to which chosen values are assigned. If *n* choices are possible, *c* is therefore executed *n* times.

  However, the order of choices is arbitrary; therefore $n!$ such executions are possible, one for each permutation of the *n* choices. If executed in "deterministic" mode, one of these executions is performed; if executed in "nondeterministic" mode, the execution yields $n!$ branches each of which proceeds according to one permutation.

- `choose` *q* `do` *c* attempts to choose a value for the quantified variables in *q*. If no such choice is possible, the loop terminates. Otherwise it executes *c* in a context arising from this choice (i.e., in a context that contains constants for the quantified variables to which the chosen values are assigned). It then repeats its behavior with the next attempt to perform a choice.

  When executed in "deterministic" mode, the loop repeatedly make a choice until no more choice is possible. When executed in "nondeterministic" choice, each choice with *n* possibilities yields *n* execution branches. By the repeated choice in each branch, ultimately the execution may ultimately yield super-exponentially many branches.

Every kind of loop may be annotated by multiple "loop invariants", i.e., clauses of form `invariant` *b* where *b* is a formula that is evaluated before/after every loop iteration. If the value of *b* is "false" for any evaluation, execution aborts, otherwise the clause has no effect. Formula *b* may refer to identifiers of form `old_`*id*, the values of such an identifiers is the value of the program variable *id* immediately before the loop started execution. In a loop `for` *q* `do` *c*, the invariant may refer to the identifier `forSet` which denotes the set of all values chosen in the previous iterations of the loop (initially this set is empty, ultimately it holds all choices).

Furthermore, every loop may be annotated by multiple "termination measures", i.e., clauses of form `decreases` e with integer expression e. This expression is evaluated after before/after every loop iteration; if the value of e becomes negative or is not less than the value before the iteration, execution aborts, otherwise the clause has no effect. The existence of at least one such clause thus ensures that the loop eventually terminates. In general, every clause may have the form `decreases` e_1 `,` $\ldots$ `,` e_n with $n \geq 1$ in which case no e_i with $1 \leq i \leq n$ may become negative and the sequence of values must be decreased by every iteration of the loop with respect to the "lexicographical order" of integer sequences (there must exist some position i with $1 \leq i \leq n$ in the sequence such that e_i is decreased and all e_j with $1 \leq j < i$ remain the same).

Pragmatics In a loop `for` c_1 `;` b `;` c_2 `do` c_3, command c_2 is for syntactic reasons restricted to some special commands (see Appendix B.7); typically c_2 is an assignment.

The difference between the two loops `for` q `do` c and `choose` q `do` c is that in the `for` loop the values of the program variables occurring in q are considered when the loop starts execution: this determines once and for all possible choices and permutations of these choices. In the `choose` loop, after every iteration of the loop a new choice is performed for the new values of the program variables. Thus to print the elements of a set `S` of type `Set[T]`, we may apply either the first form

```
for x∈S do
  print x;
```

or the second form

```
var X:Set[T] := S;
choose x∈X do
{
  print x;
  X := X\{x};
}
```

The later form is more verbose but also more flexible: it enables the loop to modify in every iteration the set of possibilities for performing the next choice.

B.3.5 Miscellaneous

Grammar

`assert` ⟨*exp*⟩ `;`
`print` `(` ⟨*string*⟩ `,` `)?` ⟨*exp*⟩ `(` `,` ⟨*exp*⟩ `)*` `;`
`print` ⟨*string*⟩ `;`
`check` ⟨*ident*⟩ `(` `with` ⟨*exp*⟩ `)?` `;`

Description We are now going to describe those commands that do not fit the previously listed categories:

- The assertion command `assert` b`;` first evaluates formula b; if the result is "false", the computation aborts, otherwise the command has no effect.
- The print command `print` e_1`,...,`e_n`;` prints the values of $e_1, \ldots, e_n$. A command like `print "...{`i`}...",` e_1`,...,`e_n`;` prints this values in a context defined by the given string. This string is printed literally, except that every occurrence of a token `{`i`}` with $1 \leq i \leq n$ is replaced by the value of e_i.
- The print command `print "..."`; prints the given literal string.
- The formula `check` f applies the function (predicate, theorem, procedure) f to all values of its parameter domain; execution aborts, if some of the executions resulted in an error, and continues normally, otherwise. The command `check` f `with` S applies f to all values of S. If f has one parameter of type T, S must have type `Set[`T`]`. If f has multiple parameters of types $T_1, \ldots, T_n$, S must have type `Set[`$T_1 \times \ldots \times T_n$`]`.

Pragmatics The `assert` and `print` commands are convenient for debugging a specification. The `check` command allows to write model checking scripts whose control flow guides the checks; the `with` clause allows to restrict the domain of the check to some computed values.

B.4 Types

Grammar

⟨*type*⟩ ::=
    `Bool`
  | (ℤ | `Int`) `[` ⟨*exp*⟩ `,` ⟨*exp*⟩ `]`
  | (ℕ | `Nat`) `[` ⟨*exp*⟩ `]`
  | `Set [` ⟨*type*⟩ `]`
  | `Tuple [` ⟨*type*⟩ (`,` ⟨*type*⟩)* `]`
  | `Record [` ⟨*ident*⟩ `:` ⟨*type*⟩ (`,` ⟨*ident*⟩ `:` ⟨*type*⟩)* `]`
  | `Array [` ⟨*exp*⟩ `,` ⟨*type*⟩ `]`
  | `Map [` ⟨*type*⟩ `,` ⟨*type*⟩ `]`
  | `()` | `Unit`
  | ⟨*ident*⟩

Description Every constant or variable has a type that constrains the values to which the constant can be bound respectively that the variable can hold. Types may depend on the values of certain integer-valued expressions which we subsequently call "constant expressions". Constant expressions may be of arbitrary form (i.e., they may contain arithmetic operations and function calls) but their values must depend only on constants that are declared on the top-level of a specification (i.e., they must not depend on function parameters or variables/constants that are locally defined in a function).

We have the following types:

- `Bool` denotes the type of the two values $\top$ (alternatively, `true`) and $\bot$ (alternatively, `false`).

  We denote by the term "truth value" a value of this type. Expressions of this type are also called "formulas", functions with this result type are also called "predicates" or "theorems" (if the predicate always returns $\top$).

- $\mathbb{Z}$`[`*min*`,`*max*`]` (alternatively, `Int[`*min*`,`*max*`]`) denotes the type of every integer number i with $min \leq i \leq max$ where *min* and *max* are constant expressions.

  In the following, we denote by the term "integer (number)" a value of such a type.

- $\mathbb{N}$`[`*max*`]` (alternatively, `Nat[`*max*`]`) is a synonym for $\mathbb{Z}$`[0,`*max*`]` where $max \geq 0$ is a constant expression.

  In the following, we denote by the term "natural number" a value of such a type.

- `Set[`T`]` denotes the type of all sets whose elements have type T.

  In the following, we denote by the term "set" a value of such a type.

- `Tuple[`T_1`,...,`T_n`]` denotes the type of all tuples with $n \geq 1$ components that have types $T_1, \ldots, T_n$; the components are numbered $1, \ldots, n$.

  In the following, we denote by the term "tuple" a value of such a type.

- `Record[`id_1`:`T_1`,...,`id_n`:`T_n`]` denotes the type of all records with $n \geq 1$ components that have types $T_1, \ldots, T_n$; the components are identified by names $id_n, \ldots, id_n$.

  In the following, we denote by the term "record" a value of such a type.

- `Array[`n`,`T`]` denotes the type of all arrays with n elements that have type T where $n \geq 0$ is a constant expression; the elements are identified by indices $0, \ldots, n-1$.

  In the following, we denote by the term "array" a value of such a type.

- `Map[`K`,`E`]` denotes the type of all values that map values of type K (the "key type") to values of type E (the "element type"); the elements are identified by the keys.

  In the following, we denote by the term "map" a value of such a type.

- `()` (alternatively, `Unit`) denotes the type that has a single value `()` which we call "unit".

- Identifier *id* denotes a type that has been defined on the top level of the specification by a `type` or `rectype` definition; the later kind of definition introduces "recursive types" that are described in the section on type declarations.

Pragmatics The type system is deliberately designed in such a way that every type (with evaluated constant expressions) has only finitely many values, which makes all formulas involving variables of such types decidable.

The type system also ensures that every type has at least one value such that quantified formulas over variables of an empty type are not trivial.

`Array[`n`,`T`]` is essentially a synonym for `Map[`$\mathbb{N}$`[`n`-1],`T`]`. However, arrays and maps have different runtime representations, which makes the use of arrays more efficient.

Type `()` may serve as the return type of procedures that produce output but do not return meaningful result values.

Types are partially checked via static analysis by a type checker, partially by runtime assertions during execution:

- The static analysis checks that the value assigned to a variable has the same base type as the variable and rejects a specification, if this is not the case. Thus e.g. a specification that tries to assign a `Bool` value to a variable of type $\mathbb{N}$`[1]` is rejected. However, a specification is not rejected, if it assigns a value of type $\mathbb{N}$`[2]` to a variable of type $\mathbb{N}$`[1]`, i.e., the static analysis does not consider the values of the constant expressions in a type.
- Runtime assertions check that the value assigned to a variable is within the range determined by the constant expressions of a type. Thus e.g. the execution of a specification that tries to assign the value 2 to a variable of type $\mathbb{N}$`[1]` aborts with an error message.

B.5 Expressions

In this section, we are going to describe the domain

⟨*exp*⟩ ::= ... | `(` ⟨*exp*⟩ `)`

of *expressions*, i.e., syntactic phrases that denote *values*, including truth values (i.e., expressions encompass both *terms* and *formulas* of classical logic). All possible values of an expression have the same type, see Section B.4.

In the following grammar snippets, the various kinds of expressions are listed in the order of decreasing binding power; as usual, an expression `(` ⟨*exp*⟩ `)` with parentheses may be used to indicate the intended parsing structure.

B.5.1 Constants and Applications

Grammar

⟨*ident*⟩
⟨*ident*⟩ `(` `(` ⟨*exp*⟩ `(` `,` ⟨*exp*⟩ `)`* `)`? `)`

Description An identifier *id* denotes the value to which the name *id* has been bound in the current environment. This binding may arise from the definition of a global constant or theorem (a constant whose value is a truth value), from the value assigned to a parameter of a function, predicate, theorem, or procedure, from the definition of a local constant or variable in a procedure, from the definition of a local constant by a binder in an expression, or from the value assigned to a variable by a quantifier.

An application *id*`(`e_1`,...,`e_n`)` denotes the value of the application of the "parameterized entity" denoted by *id* to the values of expressions $e_1, \ldots, e_n$ whose types must be those given to the parameters of the entity. This parameterized entity may be a function, a predicate, a theorem, or a procedure that has been previously declared on the top level of the specification.

B.5.2 Formulas

Grammar

⊤ | `true`
⊥ | `false`
(¬ | ~) ⟨*exp*⟩
⟨*exp*⟩ (∧ | /\ ) ⟨*exp*⟩
⟨*exp*⟩ (∨ | \/) ⟨*exp*⟩
⟨*exp*⟩ (⇒ | =>) ⟨*exp*⟩
⟨*exp*⟩ (⇔ | <=>) ⟨*exp*⟩
(∀ | `forall`) ⟨*qvar*⟩ `.` ⟨*exp*⟩
(∃ | `exists`) ⟨*qvar*⟩ `.` ⟨*exp*⟩

Description By a "formula" we mean every expression of type `Bool`, i.e., every expression that denotes a truth value. Formulas can be constructed by the usual operators of predicate logic:

- The literal ⊤ (respectively `true`) denotes the truth value "true"; likewise the literal ⊥ (respectively `false`) denotes "false".
- The logical connectives that combine formulas to bigger formulas are represented as follows: the unary operator ¬ (respectively ~) denotes logical negation, while the binary operators ∧ (respectively /\), ∨ (respectively \/), ⇒ (respectively =>), ⇔ (respectively <=>) denote logical conjunction, disjunction, implication, and equivalence, respectively. The operators are listed in the order of decreasing binding power, i.e. `a` ∨ `¬b` ∧ `c` is parsed as `a` ∨ `((¬b)` ∧ `c)`.
- The logical quantifiers that bind a variable in a formula are represented as follows: the quantifier ∀ respectively `forall` denotes universal quantification, the quantifier ∃ respectively `exists` denotes existential quantification.

In addition, we have various atomic predicates which are listed in the subsequent sections; by an "atomic predicate" we mean every operator that takes non-truth values as arguments and returns a truth value as a result.

B.5.3 Equalities and Inequalities

Grammar

⟨*exp*⟩ = ⟨*exp*⟩
⟨*exp*⟩ (≠ | ~=) ⟨*exp*⟩
⟨*exp*⟩ < ⟨*exp*⟩
⟨*exp*⟩ (≤ | <=) ⟨*exp*⟩
⟨*exp*⟩ > ⟨*exp*⟩
⟨*exp*⟩ (≥ | >=) ⟨*exp*⟩

Description Two values of the same arbitrary (also compound) type may be compared by application of the atomic predicates = denoting "equals" and (respectively ~=) denoting "not equals"; the result of the comparison is of type `Bool`.

Furthermore, two integers, i.e., values of an integer type (not necessarily with the same type bounds) may be compared by application of the atomic predicates < denoting "less than", ≤ (respectively <=) denoting "less than or equal", > denoting "greater than", or ≥ (respectively >=) denoting "greater than or equal".

B.5.4 Integers

Grammar

⟨*decimal*⟩
⟨*exp*⟩ `!`
`-` ⟨*exp*⟩
⟨*exp*⟩ (· | `*`) ⟨*exp*⟩
⟨*exp*⟩ (· | `*`) `..` (· | `*`) ⟨*exp*⟩
⟨*exp*⟩ `^` ⟨*exp*⟩
⟨*exp*⟩ `/` ⟨*exp*⟩
⟨*exp*⟩ `%` ⟨*exp*⟩
⟨*exp*⟩ `-` ⟨*exp*⟩
⟨*exp*⟩ `+` ⟨*exp*⟩
⟨*exp*⟩ `+ .. +` ⟨*exp*⟩
(∑ | Σ | `sum`) ⟨*qvar*⟩ `.` ⟨*exp*⟩
(∏ | Π | `product`) ⟨*qvar*⟩ `.` ⟨*exp*⟩
`min` ⟨*qvar*⟩ `.` ⟨*exp*⟩
`max` ⟨*qvar*⟩ `.` ⟨*exp*⟩
`#` ⟨*qvar*⟩

Description The following kinds of expressions denote integers:

- A sequence $d_1 \ldots d_n$ of $n \geq 1$ decimal digits denotes an integer in decimal representation.
- Application of the unary prefix operator `-` denotes arithmetic negation. Application of the unary postfix operator `!` denotes the computation of the factorial, i.e., $n!$ denotes the product $\prod_{i=1}^{n} i$ (whose value is 1, if $n < 1$).
- Applications of the binary operators `+`, `-`, · (center dot, alternatively `*`), `/`, `%`, and `^` denote addition, subtraction, multiplication, truncated division, the remainder of truncated division, and the exponentiation of two integers, respectively.

  The sign of the remainder denoted by `%` is the sign of the dividend, i.e., of the first argument. The result of `/` respectively `%` is undefined (i.e., the computation of the value aborts), if the divisor, i.e., the second argument, is 0. The result of `^` is undefined (i.e., the computation of the value aborts), if the second argument is negative. The operators are listed in above grammar in the order of decreasing binding power, i.e., `-a+b·c` is parsed as `(-a)+(b·c)`.

- The expression a + `..` + b denotes the sum $\sum_{i=a}^{b} i$; its value is 0, if $a > b$. Likewise, $a \cdot \texttt{..} \cdot b$ (alternatively, a `*` `..` `*` b) denotes the product $\prod_{i=a}^{b} i$; its value is 1, if $a > b$.
- The numerical quantifier $\sum$ (alternatively Σ, i.e., capital *Sigma*, or `sum`) computes the sum of the values of the quantified expression while the quantifier $\prod$ (alternatively Π, i.e., capital *Pi*, or `product`) computes the product of these values; the result is 0 respectively 1, if the quantification does not yield any value. The quantifiers `min` and `max` compute the smallest respectively the largest value; the result is undefined (i.e., the computation of the value aborts), if the quantification does not yield any value. The quantifier `#` computes the number of values that the quantification yields.

B.5.5 Sets

Grammar

(∅ | `{` `}`) `[` ⟨*type*⟩ `]`
(⋂ | `Intersect`) ⟨*exp*⟩
(⋃ | `Union`) ⟨*exp*⟩
`{` ⟨*exp*⟩ (`,` ⟨*exp*⟩)* `}`
⟨*exp*⟩ `..` ⟨*exp*⟩
`|` ⟨*exp*⟩ `|`
⟨*exp*⟩ (∩ | `intersect`) ⟨*exp*⟩
⟨*exp*⟩ (∪ | `union`) ⟨*exp*⟩
⟨*exp*⟩ \ ⟨*exp*⟩
`{` ⟨*exp*⟩ `|` ⟨*qvar*⟩ `}`
⟨*exp*⟩ ((× | `times`) ⟨*exp*⟩)+
`Set` `(` ⟨*exp*⟩ `)`
`Set` `(` ⟨*exp*⟩ `,` ⟨*exp*⟩ `)`
`Set` `(` ⟨*exp*⟩ `,` ⟨*exp*⟩ `,` ⟨*exp*⟩ `)`
⟨*exp*⟩ (∈ | `isin`) ⟨*exp*⟩
⟨*exp*⟩ (⊆ | `subseteq`) ⟨*exp*⟩

Description We have the following operations on sets:

- The literal ∅`[`T`]` (alternatively `{}[`T`]`) denotes the empty set of type `Set[`T`]`.
- `{`e_1`,...,`e_n`}` denotes the set that consists of the values $e_1, \ldots, e_n$ (which must have all the same type).
- a`..`b denotes the set of all integers i with $a \leq i \leq b$. If $a > b$, this set is empty.
- `|`S`|` denotes the cardinality (the number of elements) contained in set S.
- The operator ∩ (alternatively `intersect`) denotes the intersection of two sets of the same type, the operator ∪ (alternatively `union`) denotes their union, the operator \ denotes their difference (which consists of all elements of the first set that are not contained in

the second one). The operators are listed in the order of decreasing binding power, thus `S1 ∪ S2 ∩ S3` is parsed as `S1 ∪ (S2 ∩ S3)`.

- $\bigcap S$ (alternatively `Intersect` S) denotes the intersection of all sets contained in set S while $\bigcup S$ (alternatively `Union` S) denotes their union.
- The set builder `{` e `|` q `}` denotes the set of all values of e that result from the values of the quantified variables in q.
- The Cartesian product $S_1 \times \ldots \times S_n$ (alternatively, S_1 `times` $\ldots$ `times` S_n) denotes the set of all tuples whose component i is an element of set S_i. If this type serves as the component type of another Cartesian product, it has to be written with parentheses as `(`$S_1 \times \ldots \times S_n$`)`.
- The power set `Set(`S`)` denotes the set of all subsets of set S. `Set(`S`,`n`)` denotes the set of all subsets of S whose cardinality (number of elements) is n. `Set(`S`,`a`,`b`)` denotes the set of all subsets of S whose cardinality n satisfies $a \leq n \leq b$.
- The atomic formula $e \in S$ (alternatively e `isin` S) is true if e is an element of set S (where the type of e must be the element type of the type of S). $S_1 \subseteq S_2$ (alternatively S_1 `subseteq` S_2) is true if every element of set S_1 is an element of set S_2 (where S_1 and S_2 must have the same types).

Pragmatics The computations of `Set(S,n)` respectively `Set(S,a,b)` are more memory-efficient than the computation of the same sets by the expressions

```
{s | s∈Set(S) with |s|=n}
{s | s∈Set(S) with a≤|s| ∧ |s|≤b}
```

because in the later all elements of `Set(S)` are simultaneously stored in memory, which is not in the case for the computation of the former expressions.

B.5.6 Tuples

Grammar

(⟨ | <<) ⟨*exp*⟩ (',' ⟨*exp*⟩)* (⟩ | >>)
⟨*exp*⟩ `.` ⟨*decimal*⟩
⟨*exp*⟩ `with .` ⟨*decimal*⟩ = ⟨*exp*⟩

Description We have the following operations on tuples:

- $\langle e_1, \ldots, e_n \rangle$ (alternatively, $\ll e_1, \ldots, e_n \gg$) denotes a tuple of type `Tuple[`$T_1, \ldots, T_n$`]` whose components are the values $e_1, \ldots, e_n$ of types $T_1, \ldots, T_n$, respectively.
- t`.`d denotes the value that tuple t holds in the component with number d. Components are numbered from 1, i.e., if t holds n components, these are denoted by t`.1`,$\ldots$,t`.n`.
- t `with .`d = e denotes the tuple that is identical to t except that it holds in the component with number d value e (whose type must be the corresponding component type of t).

B.5.7 Records

Grammar

(⟨ | <<) ⟨*ident*⟩ : ⟨*exp*⟩ (',' ⟨*ident*⟩ : ⟨*exp*⟩)* (⟩ | >>)
⟨*exp*⟩ . ⟨*ident*⟩
⟨*exp*⟩ `with` . ⟨*ident*⟩ = ⟨*exp*⟩

Description We have the following operations on records:

- ⟨id_1 : e_1 , . . . , id_n : e_n⟩ (alternatively, <<id_1 : e_1 , . . . , id_n : e_n>>) denotes a record of type `Record[`id_1 : T_1 , . . . , id_n : T_n`]` whose components are values $e_1, \ldots, e_n$ of types $T_1, \ldots, T_n$, respectively.
- r . id denotes the value that record r holds in the component denoted by identifier id.
- r `with .`id = e denotes the record that is identical to r except that it holds in the component with identifier id value e (whose type is the corresponding component type of r).

B.5.8 Arrays

Grammar

`Array [` ⟨*exp*⟩ `,` ⟨*type*⟩ `] (` exp `)`
⟨*exp*⟩ `[` ⟨*exp*⟩ `]`
⟨*exp*⟩ `with [` ⟨*exp*⟩ `]` = ⟨*exp*⟩

Description We have the following operations on arrays:

- `Array[`n`,`T`](`e`)` denotes an array of type `Array[`n`,`T`]` (i.e., an array with n elements of type T) that holds at all indices the value e (whose type must be T).
- a`[`i`]` denotes the element that array a holds at index i. Indices are counted from 0, i.e., if a has length n, its elements are $a[0], \ldots, a[n-1]$. The value at any other index is undefined, i.e., the computation of such a value aborts.
- a `with [`i`]` = e denotes the array that is identical to array a except that it holds at index i value e (whose type must be the element type of a). The resulting array is undefined, if i is not a valid index in a, i.e., the computation of such an array aborts.

B.5.9 Maps

Grammar

`Map [` ⟨*type*⟩ `,` ⟨*type*⟩ `] (` exp `)`
⟨*exp*⟩ `[` ⟨*exp*⟩ `]`
⟨*exp*⟩ `with [` ⟨*exp*⟩ `]` = ⟨*exp*⟩

Description We have the following operations on maps:

- `Map[`K`,`E`](`e`)` denotes a map of type `Map[`K`,`E`]` (i.e., whose keys are of type K and whose elements are of type E) that maps all keys to the value e (whose type must be E).
- m`[`k`]` denotes the element to which map m maps key k (whose type is the key type of m).
- m `with [`k`] =` e denotes the map that is identical to m except that it maps key k (whose type must be the key type of m) to value e (whose type must be the element type of m).

Pragmatics RISCAL implements maps by hash tables that map hash values of the keys to the corresponding elements which in average gives constant time access similar to arrays (but with a higher overhead factor).

B.5.10 Recursive Values

Grammar

⟨*ident*⟩ `!` ⟨*ident*⟩
⟨*ident*⟩ `!` ⟨*ident*⟩ `(` ⟨*exp*⟩ `(` `,` ⟨*exp*⟩ `)*` `)`

Description A recursive value is a value whose type T has been introduced by a definition `rectype(`n`)` T `=` ... where constant expression $n \geq 0$ denotes an upper bound on the number of nested constructor applications. We have the following operations on recursive values:

- T`!`*id* denotes a constant *id* of the recursive type T whose definition must introduce such a constant.
- T`!`*id*`(`e_1`,`...`,`e_n`)` denotes the application of a constructor *id* of recursive type T. The definition of this type must introduce such a constructor whose parameter types are the types of $e_1, \ldots, e_n$. The result of the application is undefined (i.e, the computation aborts) if some argument v_i has been already constructed by n nested applications of some constructor of T (where n is the bound introduced in the definition of T).

B.5.11 Units

Grammar

`()`

Description The literal `()` denotes the only value of type `()` (i.e., type `Unit`).

Pragmatics The value `()` is returned by every procedure with result type `()` (also implicitly, i.e., if the procedure omits the `return` statement).

B.5.12 Conditionals

Grammar

`if` ⟨*exp*⟩ `then` ⟨*exp*⟩ `else` ⟨*exp*⟩

`match` ⟨*exp*⟩ `with` `{` (⟨*pattern*⟩ `->` ⟨*exp*⟩ `;`)+ `}`
⟨*pattern*⟩ ::= ⟨*ident*⟩ | ⟨*ident*⟩ `(` ⟨*param*⟩ (`,` ⟨*param*⟩)* `)` | `_`

Description

- The conditional expression `if` *b* `then` *t* `else` *f* first evaluates the formula *b*; if this yields the value "true", the result is the value of *t*, otherwise it is the value of *f* (both *t* and *f* must have the same type).
- The matching expression `match` *e* `with` `{` p_1`->`e_1`;` ... `;` p_n`->`e_n`;}` attempts to "match" the value of *e* (which must be of some recursive type *T*) to the patterns $p_1, \ldots, p_n$. Each pattern can be either the name *id* of a constant of type *T* or an application *id*`(`p_1`,` ... `,`p_n`)` of a constructor *id* of type *T* or the "wildcard" pattern `_`. A match succeeds if the value of *e* is the denoted constant or the result of an application of the denoted constructor or if the pattern is the wildcard `_`.

  The matches are attempted in the stated order of patterns; the first successful match of the value of *e* to some pattern p_i determines the result of the whole expression as the value of the expression e_i. If the match succeeds for a pattern p_i = *id*`(`p_1`,` ... `,`p_n`)`, the parameters $p_1, \ldots, p_n$ receive the arguments to which constructor *id* was applied to yield the value of *e*; these parameters can be consequently referenced in e_i. If there is no successful match, the result is undefined (i.e. the computation aborts).

B.5.13 Binders

Grammar

`let` ⟨*ident*⟩ `=` ⟨*exp*⟩ (`,` ⟨*ident*⟩ `=` ⟨*exp*⟩)* `in` ⟨*exp*⟩

Description The binder expression `let` id_1`=`e_1`,` ... `,`id_n`=`e_n `in` *e* binds in turn each constant id_i to the value of e_i (each subsequent binding may already refer to the previously introduced ones) and then returns the value of *e* when evaluated in this environment.

B.5.14 Choices

Grammar

`choose` ⟨*qvar*⟩
`choose` ⟨*qvar*⟩ `in` ⟨*exp*⟩
`choose` ⟨*qvar*⟩ `in` ⟨*exp*⟩ `else` ⟨*exp*⟩

Description The values of these expressions can be chosen from a finite set of possibilities. If RISCAL is executed in "deterministic" mode, an arbitrary value is chosen; if RISCAL is executed in "nondeterministic" mode, all values are chosen in turn (resulting in multiple computation branches that are executed in turn).

- `choose` q chooses a value that has been assigned to the quantified variable in q (if q introduces more than one variable, the result is a tuple of the variable values). The result is undefined, if no choice is possible. In deterministic execution, if no choice is possible, the computation is aborted.
- `choose` q `in` e chooses a value of expression e when evaluated for some values that have been assigned to the quantified variables in q (which may introduce multiple variables). The result is undefined, if no choice is possible. In deterministic execution, if no choice is possible, the computation is aborted.
- `choose` q `in` e_1 `else` e_2 either chooses a value of e_1 when evaluated for some values that have been assigned to the quantified variables in q (which may introduce multiple variables) or it chooses the value of e_2 (which must not refer to the quantified variables). Thus there is always a choice possible such that deterministic execution is never aborted.

Pragmatics The expression `choose x:T` is equivalent to `choose x:T in x`. While the other variants of choice are more general, it has been introduced due to its prominence in classical logic (Hilbert's ε-operator). The variant `choose` q `in` e_1 `else` e_2 has been introduced since it allows to simplify the typical pattern where first the existence of a choice is checked, then, if such a choice exists, it is performed, and, if not, another value is taken.

If a choice is performed in deterministic mode, RISCAL always chooses the same value (i.e., it does not perform a random choice).

B.5.15 Miscellaneous

Grammar

`assert` ⟨*exp*⟩ `in` ⟨*exp*⟩
`print` (⟨*string*⟩ `,`)? ⟨*exp*⟩
`check` ⟨*ident*⟩ (`with` ⟨*exp*⟩)?

Description We are now going to describe those expressions that do not fit the previously listed categories:

- The assertion expression `assert` b `in` e first evaluates formula b; if the result is "false", the value of the expression is undefined (i.e., the computation aborts). Otherwise, its value is the value of e.
- The print expression `print` e prints the value of e and returns it as a result. An expression like `print "...{1}...", ` e prints this value in a context defined by the given string. This string is printed literally, except that every occurrence of the token `{1}` is replaced by the value of e.

- The formula `check` f applies the function (predicate, theorem, procedure) f to all values of its parameter domain; its result is "true" if none of the executions resulted in an error, and "false", otherwise. The formula `check` f `with` S applies f to all values of S. If f has one parameter of type T, S must have type `Set[`T`]`. If f has multiple parameters of types $T_1, \ldots, T_n$, S must have type `Set[`$T_1 \times \ldots \times T_n$`]`.

Pragmatics The `assert` and `print` expressions are convenient for debugging a specification. The `check` expression allows to write model checking scripts whose control flow is guarded by the results of the checks.

B.6 Quantified Variables

Grammar

⟨*qvar*⟩ ::= ⟨*qvcore*⟩ (`,` ⟨*qvcore*⟩)* (`with` ⟨*exp*⟩)?

⟨*qvcore*⟩ ::= ⟨*ident*⟩ `:` ⟨*type*⟩ | ⟨*ident*⟩ (∈ | `in`) ⟨*exp*⟩

Description

- A phrase x`:`T `with` b introduces a quantified variable x of type T. If the optional formula b is given (which may refer to x), b yields for the value of this variable "true".
- A phrase x∈S `with` b (alternatively, x `in` S `with` b) introduces a quantified variable x whose value is an element of set S; consequently, if S has type `Set[`T`]`, then x has type T. If the optional formula b is given (which may refer to x), b yields for the value of this variable "true".
- A phrase x_1`:`T_1`,` ... `,`x_n`:`T_n `with` b introduces n quantified variables $x_1, \ldots, x_n$ of types $T_1, \ldots, T_n$. If the optional formula b is given (which may refer to $x_1, \ldots, x_n$), b yields for the value of these variables "true". Analogously, each clause x_i`:`T_i can be replaced by a clause x_i∈S_i where the value of variable x_i must be an element of set S_i.

Pragmatics The particular syntax of quantified variables makes them usable for multiple kinds of quantified constructs, for instance in the set expression

```
{ x/y | x∈S, y:T with y ≠ 0 ∧ x > y }
```

in the formula

```
∀x∈S, y:T with y ≠ 0 ∧ x > y. x/y ≥ 1
```

or in the following choice statement:

```
choose x∈S, y:T with y ≠ 0 ∧ x > y;
```

B.7 ANTLR 4 Grammar

In the following, we list the grammar used by the parser generator ANTLR 4 [3] to generate the lexical and syntactic analyzer for the language.

```
// ------------------------------------------------------------------------
// RISCAL.g4
// RISC Algorithm Language ANTLR 4 Grammar
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Copyright (C) 2016-, Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria, http://www.risc.jku.at
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
// ------------------------------------------------------------------------

grammar RISCAL;

options
{
  language=Java;
}

@header
{
  package riscal.parser;
}
@members {boolean cartesian = true;}

// ------------------------------------------------------------------------
// modules, declarations, commands
// ------------------------------------------------------------------------

specification: ( declaration )* EOF ;

declaration:
// declarations (value externally defined, others forward defined)
  'val' ident ':' ( 'Nat' | 'ℕ' ) EOS                                  #ValueDeclaration
| multiple 'fun' ident '(' ( param ( ',' param)* )? ')' ':' type EOS #FunctionDeclaration
| multiple 'pred' ident '(' ( param ( ',' param)* )? ')' EOS         #PredicateDeclaration
| multiple 'proc' ident '(' ( param ( ',' param)* )? ')' ':' type    #ProcedureDeclaration

// definitions
```

```
| 'type' ident '=' type EOS                                    #TypeDefinition
| 'rectype' '(' exp ')' ritem ( 'and' ritem)* EOS              #RecTypeDefinition
| 'enumtype' ritem EOS                                         #EnumTypeDefinition
| 'val' ident ( ':' type )? '=' exp EOS                        #ValueDefinition
| 'pred' ident  ( '⇔' | '<=>' ) exp EOS                        #PredicateValueDefinition
| 'theorem' ident ( '⇔' | '<=>' ) exp EOS                      #TheoremDefinition
| multiple 'fun' ident '(' ( param ( ',' param)* )? ')' ':' type
  ( funspec )* '=' exp EOS                                     #FunctionDefinition
| multiple 'pred' ident '(' ( param ( ',' param)* )? ')'
  ( funspec )* ( '⇔' | '<=>' ) exp EOS                        #PredicateDefinition
| multiple 'proc' ident '(' ( param ( ',' param)* )? ')' ':' type
  ( funspec )* '{' ( command )* ( 'return' exp EOS )? '}'     #ProcedureDefinition
| multiple 'theorem' ident '(' ( param ( ',' param)* )? ')'
  ( funspec )* ( '⇔' | '<=>' ) exp EOS                        #TheoremParamDefinition
;

funspec :
  'requires' exp EOS                  #RequiresSpec
| 'ensures' exp  EOS                  #EnsuresSpec
| 'decreases' exp ( ',' exp)* EOS     #DecreasesSpec
;

// commands terminated by a semicolon
scommand:
                                                               #EmptyCommand
| ident ( sel )* ( ':=' | ':=' | '=' ) exp                     #AssignmentCommand
| 'choose' qvar                                                #ChooseCommand
| 'do' ( loopspec )* command 'while' exp                       #DoWhileCommand
| 'var' ident ':' type ( ( ':=' | ':=' | '=' ) exp )?          #VarCommand
| 'val' ident ( ':' type )? ( ':=' | ':=' | '=' ) exp          #ValCommand
| 'assert' exp                                                 #AssertCommand
| 'print' ( STRING ',' )? exp ( ',' exp)*                      #PrintCommand
| 'print' STRING                                               #Print2Command
| 'check' ident ( 'with' exp )?                                #CheckCommand
| exp                                                          #ExpCommand
;

command:
  scommand EOS                                                 #SemicolonCommand
| 'choose' qvar 'then' command 'else' command                  #ChooseElseCommand
| 'if' exp 'then' command                                      #IfThenCommand
| 'if' exp 'then' command 'else' command                       #IfThenElseCommand
| 'match' exp 'with' '{' ( pcommand )+ '}'                     #MatchCommand
| 'while' exp 'do' ( loopspec )* command                       #WhileCommand
| 'for' scommand EOS exp EOS scommand 'do' ( loopspec )* command #ForCommand
| 'for' qvar 'do' ( loopspec )* command                        #ForInCommand
| 'choose' qvar 'do' ( loopspec )* command                     #ChooseDoCommand
| '{' ( command )* '}'                                         #CommandSequence
;

loopspec :
  'invariant' exp   EOS             #InvariantLoopSpec
| 'decreases' exp ( ',' exp)* EOS #DecreasesLoopSpec
;
```

```
// ----------------------------------------------------------------------------
// expressions, types
// ----------------------------------------------------------------------------

exp  :
// literals
  '(' ')'                                  #UnitExp
| ( '⊤' | 'true'    )                      #TrueExp
| ( '⊥' | 'false'   )                      #FalseExp
| decimal                                  #IntLiteralExp
| ( '∅' | '{' '}' ) '[' type ']'           #EmptySetExp

// identifier-like
| ident '!' ident                           #RecIdentifierExp
| ident '!' ident '(' exp ( ',' exp)* ')'  #RecApplicationExp
| ident                                     #IdentifierExp
| ident '(' ( exp ( ',' exp)* )? ')'       #ApplicationExp
| 'Array' '[' exp ',' type ']' '(' exp ')' #ArrayBuilderExp
| 'Map' '[' type ',' type ']' '(' exp ')'  #MapBuilderExp
| 'Set' '(' exp ')'                         #PowerSetExp
| 'Set' '(' exp ',' exp ')'                 #PowerSet1Exp
| 'Set' '(' exp ',' exp ',' exp ')'         #PowerSet2Exp

// selections
| exp '[' exp ']'                          #ArraySelectionExp
| exp '.' decimal                          #TupleSelectionExp
| exp '.' ident                            #RecordSelectionExp

// postfix-term-like
| exp '!'                                   #FactorialExp

// prefix-term-like
| '-' exp                                   #NegationExp
| ('⋂' | 'Intersect') exp                   #BigIntersectExp
| ('⋃' | 'Union') exp                       #BigUnionExp

// infix-term-like
| exp '^' exp                               #PowerExp
| exp ( '*' | '·' ) exp                     #TimesExp
| exp ( '*' | '·' ) '..' ( '*' | '·' ) exp #TimesExpMult
| exp '/' exp                               #DividesExp
| exp '%' exp                               #RemainderExp
| exp '-' exp                               #MinusExp
| exp '+' exp                               #PlusExp
| exp '+' '..' '+' exp                      #PlusExpMult
| '{' exp ( ',' exp)* '}'                   #EnumeratedSetExp
| exp '..' exp                              #IntervalExp
| ( '⟨' | '⟨' | '⟨' | '<<' ) exp ( ',' exp )* ( '⟩' | '⟩' | '⟩' | '>>' )       #TupleExp
| ( '⟨' | '⟨' | '⟨' | '<<' ) eident ( ',' eident )* ( '⟩' | '⟩' | '⟩' | '>>' ) #RecordExp
| '|' exp '|'                               #SetSizeExp
| exp ( '∩' | 'intersect') exp              #IntersectExp
| exp ( '∪' | 'union' ) exp                 #UnionExp
| exp '\\' exp                              #WithoutExp
```

```
// n-ary infix-term-like (hack allows non-associative parsing without parentheses)
| '(' exp ( ( '×' | 'times' ) exp  )+ ')' #CartesianExp
| exp ( {cartesian}? ( '×' | 'times' ) {cartesian=false;} exp {cartesian=true;} )+ #CartesianExp

// term-quantifier like
| '#' qvar                                #NumberExp
| '{' exp '|' qvar '}'                    #SetBuilderExp
| 'choose' qvar                           #ChooseExp
| ( '∑' | 'Σ' | 'sum' ) qvar '.' exp       #SumExp
| ( '∏' | 'Π' | 'product' ) qvar '.' exp   #ProductExp
| 'min' qvar '.' exp                      #MinExp
| 'max' qvar '.' exp                      #MaxExp

// updates
| exp 'with' '[' exp ']' '=' exp          #ArrayUpdateExp
| exp 'with' '.' decimal '=' exp          #TupleUpdateExp
| exp 'with' '.' ident '=' exp            #RecordUpdateExp

// relations
| exp '=' exp                             #EqualsExp
| exp ( '≠' | '~=' ) exp                  #NotEqualsExp
| exp '<' exp                             #LessExp
| exp ( '≤' | '<=' ) exp                  #LessEqualExp
| exp '>' exp                             #GreaterExp
| exp ( '≥' | '>=' ) exp                  #GreaterEqualExp
| exp ( '∈' | 'isin' ) exp                #InSetExp
| exp ( '⊆' | 'subseteq' ) exp            #SubsetExp

// propositions
| ( '¬' | '~' ) exp                       #NotExp
| exp ( '∧' | '/\\' ) exp                 #AndExp
| exp ( '∨' | '\\/' ) exp                 #OrExp
| exp ( '⇒' | '=>' ) exp                  #ImpliesExp
| exp ( '⇔' | '<=>' ) exp                 #EquivExp

// conditionals
| 'if' exp 'then' exp 'else' exp             #IfThenElseExp
| 'match' exp 'with' '{' ( pexp ';' )+ '}'   #MatchExp

// quantified formulas
| ( '∀' | 'forall' ) qvar '.' exp            #ForallExp
| ( '∃' | 'exists' ) qvar '.' exp            #ExistsExp

// binders
| 'let' binder ( ',' binder )* 'in' exp   #LetExp
| 'choose' qvar 'in' exp                  #ChooseInExp
| 'choose' qvar 'in' exp 'else' exp       #ChooseInElseExp

// assertions
| 'assert' exp 'in' exp                   #AssertExp

// print expression
| 'print' ( STRING ',' )? exp             #PrintExp
```

```
// check operation
| 'check' ident ( 'with' exp )?           #CheckExp

// parenthesized expressions
| '(' exp ')'                             #ParenthesizedExp
;

// types
type :
  ( 'Unit' | '(' ')' )                    #UnitType
| 'Bool'                                  #BoolType
| ( 'Int' | 'ℤ' )  '[' exp ',' exp ']'    #IntType
| 'Map' '[' type ',' type ']'             #MapType
| 'Tuple' '[' type ( ',' type)* ']'       #TupleType
| 'Record' '[' param ( ',' param)* ']'    #RecordType
| ( 'Nat' | 'ℕ' ) '[' exp ']'             #NatType
| 'Set' '[' type ']'                      #SetType
| 'Array' '[' exp ',' type ']'            #ArrayType
| ident                                   #IdentifierType
;

// ----------------------------------------------------------------------------
// auxiliaries
// ----------------------------------------------------------------------------

qvar :
  qvcore ( ',' qvcore )* ( 'with' exp )?    #QuantifiedVariable
;

qvcore :
  ident ':' type            #IdentifierTypeQuantifiedVar
| ident ( '∈' | 'in' ) exp  #IdentifierSetQuantifiedVar
;

binder : ident '=' exp ;

pcommand :
  ident'->' command                              #IdentifierPatternCommand
| ident'(' param ( ',' param)* ')' '->' command  #ApplicationPatternCommand
| '_' '->' command                               #DefaultPatternCommand
;

pexp :
  ident'->' exp                                  #IdentifierPatternExp
| ident'(' param ( ',' param)* ')' '->' exp       #ApplicationPatternExp
| '_' '->' exp                                   #DefaultPatternExp
;

param : ident ':' type ;

ritem: ident '=' rident ( '|' rident)* ;

eident : ident ':' exp ;
```

```
rident :
  ident                              #RecIdentifier
| ident '(' type ( ',' type)* ')'   #RecApplication
;

sel :
  '[' exp ']'       #MapSelector
| '.' decimal       #TupleSelector
| '.' ident         #RecordSelector
;

multiple :
             #IsNotMultiple
| 'multiple' #IsMultiple
;

ident   : IDENT ;
decimal : DECIMAL ;

// ------------------------------------------------------------------------
// lexical rules
// ------------------------------------------------------------------------

// reserve leading underscore for internal identifiers
IDENT   : [a-zA-Z][a-zA-Z_0-9]* ;
DECIMAL : [0-9]+ ;
EOS     : ';' ;

// format string literals
STRING : '"' (ESC|.)*? '"' ;
fragment ESC : '\\"' | '\\n' | '\\%' | '\\\\';

WHITESPACE  : [ \t\r\n\f]+ -> skip ;
LINECOMMENT : '//' .*? '\r'? ('\n' | EOF) -> skip ;
COMMENT     : '/*' .*? '*/' -> skip ;

// matches any other character
ERROR : . ;

// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

C Example Specifications

In the following, we list the example specifications used in the tutorial.

C.1 Euclidean Algorithm

```
// ------------------------------------------------------------------------
// Computing the greatest common divisor by the Euclidean Algorithm
```

```
// ---------------------------------------------------------------------------

val N: ℕ;
type nat = ℕ[N];

pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;

fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose result:nat with
    divides(result,m) ∧ divides(result,n) ∧
    ¬∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;

val g:nat = gcd(N,N-1);

theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);

proc gcdp(m:nat,n:nat): nat
  requires m≠0 ∨ n≠0;
  ensures result = gcd(m,n);
{
  var a:nat := m;
  var b:nat := n;
  while a > 0 ∧ b > 0 do
    invariant gcd(a,b) = gcd(old_a,old_b);
    decreases a+b;
  {
    if a > b then
      a := a%b;
    else
      b := b%a;
  }
  return if a = 0 then b else a;
}

proc main(): ()
{
  choose m:nat, n:nat with m ≠ 0 ∨ n ≠ 0;
  print m,n,gcdp(m,n);
}
```

C.2 Insertion Sort

```
// ---------------------------------------------------------------------------
// Sorting arrays by the Insertion Sort Algorithm
// ---------------------------------------------------------------------------

val N:Nat;
val M:Nat;
type nat = Nat[M];
```

```
type array = Array[N,nat];
type index = Nat[N-1];

proc sort(a:array): array
  ensures ∀i:nat. i < N-1 => result[i] ≤ result[i+1];
  ensures ∃p:Array[N,index].
            (∀i:index,j:index. i ≠ j ⇒ p[i] ≠ p[j]) ∧
            (∀i:index. a[i] = result[p[i]]);
{
  var b:array = a;
  for var i:Nat[N]:=1; i<N; i:=i+1 do
    decreases N-i;
  {
    var x:nat := b[i];
    var j:Int[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do
      decreases j+1;
    {
      b[j+1] := b[j];
      j := j-1;
    }
    b[j+1] := x;
  }
  return b;
}

proc main(): Unit
{
  choose a: array;
  print a, sort(a);
}
```

C.3 DPLL Algorithm

```
// ---------------------------------------------------------------------------
// SAT solving by the DPLL Algorithm
// ---------------------------------------------------------------------------

// the number of literals
val n: ℕ; // e.g. 3;

// the raw types
type Literal = ℤ[-n,n];
type Clause  = Set[Literal];
type Formula = Set[Clause];
type Valuation  = Set[Literal];

// a consistency condition
pred consistent(l:Literal,c:Clause) ⇔ ¬(l∈c ∧ -l∈c);

// the type restrictions
pred literal(l:Literal) ⇔ l≠0;
```

```
pred clause(c:Clause) ⇔ ∀l∈c. literal(l) ∧ consistent(l,c);
pred formula(f:Formula) ⇔ ∀c∈f. clause(c);
pred valuation(v:Valuation) ⇔ clause(v);

// the satisfaction relation
pred satisfies(v:Valuation, l:Literal) ⇔ l∈v;
pred satisfies(v:Valuation, c:Clause) ⇔ ∃l∈c. satisfies(v, l);
pred satisfies(v:Valuation, f:Formula) ⇔ ∀c∈f. satisfies(v,c);

// the satisfiability and the validity of a relation
pred satisfiable(f:Formula) ⇔
  ∃v:Valuation. valuation(v) ∧ satisfies(v,f);
pred valid(f:Formula) ⇔
  ∀v:Valuation. valuation(v) ⇒ satisfies(v,f);

// the negation of a formula
fun not(f: Formula):Formula =
  { c | c:Clause with clause(c) ∧ ∀d∈f. ∃l∈d. -l∈c };
theorem notFormula(f:Formula)
  requires formula(f);
⇔ formula(not(f));
theorem notValid(f:Formula)
  requires formula(f);
⇔ valid(f) ⇔ ¬satisfiable(not(f));

// the literals of a formula
fun literals(f:Formula):Set[Literal] =
  {l | l:Literal with ∃c∈f. l∈c};

// the result of setting a literal l in formula f to true
fun substitute(f:Formula,l:Literal):Formula =
  {c\{-l} | c∈f with ¬(l∈c)};

// the recursive DPLL algorithm (without optimizations)
multiple pred DPLL(f:Formula)
  requires formula(f);
  ensures result ⇔ satisfiable(f);
  decreases |literals(f)|;
⇔
  if f = ∅[Clause] then
    ⊤
  else if ∅[Literal] ∈ f then
    ⊥
  else
    choose l∈literals(f) in
    DPLL(substitute(f,l)) ∨ DPLL(substitute(f,-l));

// the variables in a formula
fun vars(f:Formula): Set[ℕ[n]] =
  { if l>0 then l else -l | l ∈ literals(f) };

// the maximum number of nodes in the search tree
val m = 2^(n+1)-1;
```

```
// the number of nodes in the search tree for f
fun size(f:Formula): ℕ[m] = 2^(|vars(f)|+1)-1;

// the number of nodes in the search tree for the formulas in stack[0..i-1]
fun size(stack:Array[n+1,Formula], i:ℕ[n+1]): ℕ[m] =
  ∑k:ℕ[n] with k<i. size(stack[k]);

// the iterative DPLL algorithm (without optimizations)
proc DPLL2(f:Formula): Bool
  requires formula(f);
  ensures result ⇔ satisfiable(f);
{
  var satisfiable: Bool := ⊥;
  var stack: Array[n+1,Formula] := Array[n+1,Formula](∅[Clause]);
  var number: ℕ[n+1] := 0;
  stack[number] := f;
  number := number+1;
  while ¬satisfiable ∧ number>0 do
    invariant 0 ≤ number ∧ number ≤ n+1;
    invariant satisfiable(f) ⇔ satisfiable ∨
      ∃i:ℕ[n+1] with i<number. satisfiable(stack[i]);
    decreases if satisfiable then 0 else size(stack, number);
  {
    number := number-1;
    var g:Formula := stack[number];
    if g = ∅[Clause] then
      satisfiable := ⊤;
    else if ¬∅[Literal]∈g then
    {
      choose l∈literals(g);
      stack[number] := substitute(g,-l);
      number := number+1;
      stack[number] := substitute(g,l);
      number := number+1;
    }
  }
  return satisfiable;
}

proc main0(): ()
{
  val f = {{1,2,3},{-1,2},{-2,3},{-3}};
  val r = DPLL2(f);
  print f,r;
}

// maximal sizes of clauses and formulas
val cn: ℕ; // e.g. 2;
val fn: ℕ; // e.g. 20;

// get set of formulas satisfying above restrictions
fun formulas(): Set[Formula] =
let
  literals = { l | l:Literal with literal(l) },
```

```
  clauses = { c | c ∈ Set(literals) with |c| ≤ cn ∧ clause(c) },
  formulas = { f | f ∈ Set(clauses) with |f| ≤ fn ∧ formula(f) }
in formulas;

proc main1(): ()
{
  // apply check to a specific set of formulas
  check DPLL with formulas();
}

proc main2(): ()
{
  // apply non-determinism to checking
  val formulas: Set[Formula] = formulas();
  print "number: {1}", |formulas|;
  choose f ∈ formulas;
  val r = DPLL(f);
  print f, r;
}

proc main3(): ()
{
  // check formulas deterministically
  val formulas: Set[Formula] = formulas();
  print "number: {1}", |formulas|;
  var i:ℕ[2^20] := 0;
  for f ∈ formulas do
  {
    val r = DPLL(f);
    // print i, f, r;
    if (i%100 = 0) then print i;
    i := i+1;
  }
}
```