

The RISC Algorithm Language (RISCAL)

Tutorial and Reference Manual
(Version 2.7.*)

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

March 11, 2019

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, we determine a finite model in which all algorithms, functions, and predicates are executable, and all formulas are 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. Furthermore, a verification condition generator allows to validate, by generating verification conditions and checking them in some model, whether the program annotations are strong enough to subsequently perform a proof-based verification that ensures the general correctness of the algorithm for infinitely many models.

Contents

1	Introduction	4
2	A Quick Start	5
2.1	System Overview	5
2.2	Specification Language	7
2.3	Language Features	9
2.4	Execution and Model Checking	10
2.5	Parallelism	13
2.6	Validating Specifications	15
2.7	Verifying Implementations	19
2.8	Visualizing Execution Traces	26
2.9	Visualizing Evaluation Trees	29
3	More Examples	37
3.1	Linear and Binary Search	37
3.2	Insertion Sort	41
3.3	DPLL Algorithm	46
3.4	DPLL Algorithm with Subtypes	51
4	Related Work	53
5	Conclusions and Future Work	56
A	The Software System	61
A.1	Installing the Software	61
A.2	Running the Software	65
A.3	The User Interface	66
A.4	Distributed Execution	73
A.5	Visualization	74
B	The Specification Language	75
B.1	Lexical and Syntactic Structure	75
B.2	Specifications and Declarations	77
B.2.1	Types	78
B.2.2	Values	79
B.2.3	Functions	80
B.3	Commands	83
B.3.1	Declarations and Assignments	83
B.3.2	Choices	84
B.3.3	Conditionals	84
B.3.4	Loops	85
B.3.5	Miscellaneous	87
B.4	Types	88

B.5	Expressions	90
B.5.1	Constants and Applications	90
B.5.2	Formulas	91
B.5.3	Equalities and Inequalities	91
B.5.4	Integers	92
B.5.5	Sets	93
B.5.6	Tuples	94
B.5.7	Records	95
B.5.8	Arrays	95
B.5.9	Maps	95
B.5.10	Recursive Values	96
B.5.11	Units	97
B.5.12	Conditionals	97
B.5.13	Binders	98
B.5.14	Choices	98
B.5.15	Miscellaneous	99
B.6	Quantified Variables	100
B.7	ANTLR 4 Grammar	101
C	Example Specifications	107
C.1	Euclidean Algorithm	107
C.2	Bubble Sort	108
C.3	Linear and Binary Search	109
C.4	Insertion Sort	110
C.5	DPLL Algorithm	112
C.6	DPLL Algorithm with Subtypes	115

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 [31, 25], 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 [28]. 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. Thus since Version 1 the RISCAL software has supported 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).

However, ensuring the correctness of formal annotations such as loop invariants by model checking program executions only ensures that the annotations are not too strong, i.e., that they are not violated by any execution. Checking program executions does not rule out that the annotations are too weak to derive from them valid verification conditions that ensure the correctness of the program by a proof-based verification (which is required, if the general correctness of the program for models of arbitrary size is to be ensured); in particular, a loop invariant may not be inductive, i.e., it may be too weak to ensure its preservation across loop iterations. Therefore Version 2 of the RISCAL software documented in this report also includes a verification condition generator that produces from the annotations verification conditions whose validity ensures the correctness of the program; since these conditions are formulas, they may be also checked in a finite model by the RISCAL model checker. A successful check of these conditions demonstrates that with these annotations a proof-based verification in that model is possible; this increases our confidence that with these annotations also a proof-based verification for models of arbitrary size may succeed (conversely, if the check fails, this demonstrates that such a proof-based verification need not be attempted yet).

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. Further examples on the use of this software is provided by some publications that may be also downloaded from this web site, e.g. [32].

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 19 on page 67).

2.1 System Overview

The system is started from the command line by executing

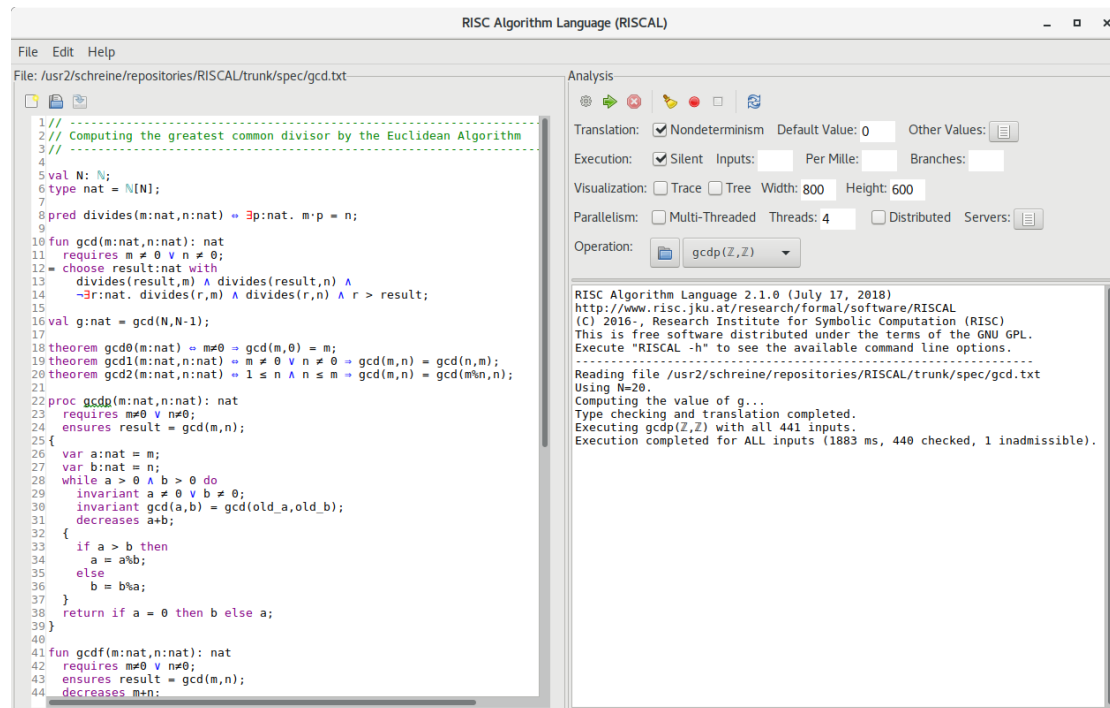



Figure 1: The RISCAL User Interface

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 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.

2.2 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 $0, \dots, N$ (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

$$\begin{aligned} divides &\subseteq nat \times nat \\ divides(m, n) &:\Leftrightarrow \exists p \in nat. m \cdot p = n \end{aligned}$$

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

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

This function satisfies certain general properties stated as follows:

```
theorem gcd0(m:nat)  $\Leftrightarrow$  m  $\neq$  0  $\Rightarrow$  gcd(m,0) = m;
theorem gcd1(m:nat,n:nat)  $\Leftrightarrow$  m  $\neq$  0  $\vee$  n  $\neq$  0  $\Rightarrow$  gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat)  $\Leftrightarrow$  1  $\leq$  n  $\wedge$  n  $\leq$  m  $\Rightarrow$  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

$$\begin{aligned} \forall m \in \text{nat}. m \neq 0 &\Rightarrow \text{gcd}(m, 0) = m \\ \forall m \in \text{nat}, n \in \text{nat}. m \neq 0 \vee n \neq 0 &\Rightarrow \text{gcd}(m, n) = \text{gcd}(n, m) \\ \forall m \in \text{nat}, n \in \text{nat}. 1 \leq n \wedge n \leq m &\Rightarrow \text{gcd}(m, n) = \text{gcd}(m \bmod n, n) \end{aligned}$$

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  $\neq$  0  $\vee$  n  $\neq$  0;
  ensures result = gcd(m,n);
{
  var a:nat := m;
  var b:nat := n;
  while a > 0  $\wedge$  b > 0 do
    invariant a  $\neq$  0  $\vee$  b  $\neq$  0;
    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** clauses state 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 (of which at least one is not zero) 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. Furthermore, RISCAL generates from these clauses verification conditions whose validity implies the correctness of the program with respect to its specification (see Section 2.7).

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.

2.3 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

$$\exists p:\text{nat}. m \cdot p = n$$

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.

2.4 Execution and Model Checking

Whenever a specification file is loaded or saved, the corresponding specification is processed, i.e., checked for syntactic and type errors (with graphical markers indicating 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 these values, the specification is automatically re-processed before execution). In our example, we may thus get 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  $m \neq 0 \vee n \neq 0$ ;  
  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).  
WARNING: not all nondeterministic branches 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( $\mathbb{Z}, \mathbb{Z}$ ) 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).
WARNING: not all nondeterministic branches have been considered.
```

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

```
requires  $m \neq 0 \vee n \neq 0$ ;
```

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( $\mathbb{Z}, \mathbb{Z}$ ) with all 16 inputs.
Execution completed for ALL inputs (6 ms, 15 checked, 1 inadmissible).
WARNING: not all nondeterministic branches have been considered.
```


which just gives the essential information.

2.5 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( $\mathbb{Z}, \mathbb{Z}$ ) 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).
 WARNING: not all nondeterministic branches 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 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( $\mathbb{Z}, \mathbb{Z}$ ) 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).
WARNING: not all nondeterministic branches 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 $\text{gcd2}(\mathbb{Z}, \mathbb{Z})$ defined as

$$\text{theorem gcd2}(m:\text{nat}, n:\text{nat}) \Leftrightarrow 1 \leq n \wedge n \leq m \Rightarrow \text{gcd}(m, n) = \text{gcd}(m \% n, n);$$

for which the output

```
Executing gcd2( $\mathbb{Z}, \mathbb{Z}$ ) with all 10201 inputs.
PARALLEL execution with 4 threads (output disabled).
1904 inputs (1704 checked, 0 inadmissible, 0 ignored, 200 open)...
```

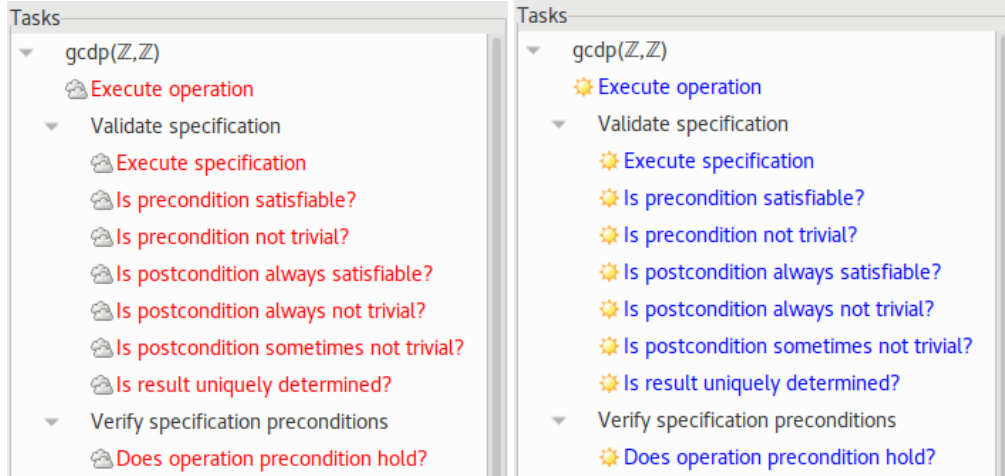


Figure 2: Operation-Related Tasks


```
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).
WARNING: not all nondeterministic branches have been considered.
```

validates its correctness.

As illustrated by these 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).

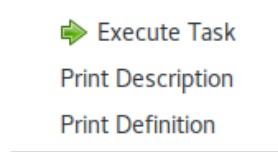
2.6 Validating Specifications


As demonstrated above, we can validate the correctness of an operation by checking whether it satisfies its specification. However, it is by no means sure that the formulas in the specification indeed appropriately express our intentions of how the operation shall behave: for instance, by a slight logical error, the postcondition of a specification may be trivially satisfied by every output. This kind of error can be apparently not detected by just checking the operation.

Nevertheless RISCAL provides some means to validate the adequacy of a specification according to various criteria. If we press the button  “Show/Hide Tasks”, the user interface is extended on the right side by a view of the folder depicted in the left part of Figure 2; this folder lists a couple of tasks related to the currently selected operation. These tasks are initially colored red (and marked with a cloud symbol) to indicate that they are still open; once they have been performed, they become blue (and are marked with a sun symbol) as shown in the right part of Figure 2. Our focus is now on the tasks generated for the greatest common divisor function `gcdp` introduced in Section 2.2 and checked in Section 2.4.

For instance, the task “Execute operation” denotes the checking of the operation itself, i.e., its application to all inputs allowed by the precondition. The execution of a task may be triggered

by a double click on the item; alternatively, by a right-click a menu



pops up in which the entry  “Execute Task” may be selected (by a right-click on a folder, a corresponding button “Execute All Tasks” is displayed that executes all tasks in the folder and in its subfolders). Likewise the menu entry “Print Description” prints a verbal description of the task while “Print Definition” prints its definition (i.e. the definition of an executable operation in the RISCAL language).

Furthermore, the folder “Validate specification” lists the following tasks:

Execute specification This task executes an automatically generated function whose result is implicitly defined by the postcondition of the selected operation. For instance, in case of the procedure `gcdp`, the menu entry “Print Definition” shows the following operation:

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

If we execute this task for $N = 6$ in non-deterministic and non-silent mode, we get the following output which demonstrates that the postcondition indeed determines the greatest common divisor:

```
Executing gcdp( $\mathbb{Z}, \mathbb{Z}$ ) with all 49 inputs.
Ignoring inadmissible inputs...
Run 1 of deterministic function gcdp(1,0):
Result (2 ms): 1
..
Run 17 of deterministic function gcdp(3,2):
Result (1 ms): 1
...
Run 34 of deterministic function gcdp(6,4):
Result (1 ms): 2
...
Run 48 of deterministic function gcdp(6,6):
Result (1 ms): 6
Execution completed for ALL inputs (419 ms, 48 checked, 1 inadmissible).
```

The execution of this task should proceed in non-deterministic mode to ensure that, if the postcondition allows multiple outputs (which may or may be not intended, see also the discussion concerning the last task below), all of the outputs allowed by the postcondition of the operation are indeed displayed.

Is precondition satisfiable/not trivial? These tasks demonstrate that there is at least one input that satisfies the precondition respectively that there is at least one output that violates the precondition. In more detail, the tasks denote the following theorems:


```

theorem _gcdpPreSat()  $\Leftrightarrow \exists m:\text{nat}, n:\text{nat}. ((m \neq 0) \vee (n \neq 0));$ 
theorem _gcdpPreNotTrivial()  $\Leftrightarrow \exists m:\text{nat}, n:\text{nat}. (\neg((m \neq 0) \vee (n \neq 0)));$ 

```

These theorems are apparently true in our concrete case:

```

Executing _gcdpPreSat().
Execution completed (0 ms).
Executing _gcdpPreNotTrivial().
Execution completed (0 ms).

```

If the first condition were violated, gcdp would not accept any input; if the second condition were violated, gcdp would accept every input (both cases would make the precondition pointless).

Is postcondition always satisfiable? This task demonstrates that, for every input that satisfies the precondition, there exists at least one output that satisfies the postcondition:

```

theorem _gcdpPostSat(m:nat, n:nat)
requires (m  $\neq$  0)  $\vee$  (n  $\neq$  0);
 $\Leftrightarrow \exists \text{result}:\text{nat}. (\text{result} = \text{gcd}(m, n));$ 

```

Again, this theorem is true in our case:

```

Executing _gcdpPostSat( $\mathbb{Z}$ ,  $\mathbb{Z}$ ) with all 49 inputs.
Execution completed for ALL inputs (14 ms, 48 checked, 1 inadmissible).

```

If the theorem were violated, the specification could not be correctly implemented: for some legal input, gcdp could not return any legal output.

Is postcondition always/sometimes not trivial? These tasks demonstrate that the postcondition indeed rules out some illegal outputs (otherwise the postcondition would be pointless). In more detail, the first (stronger) theorem states that for every input some outputs are prohibited:

```

theorem _gcdpPostNotTrivialAll(m:nat, n:nat)
requires (m  $\neq$  0)  $\vee$  (n  $\neq$  0);
 $\Leftrightarrow \exists \text{result}:\text{nat}. (\neg(\text{result} = \text{gcd}(m, n)));$ 

```

However, sometimes an operation might indeed allow for some special cases of inputs arbitrary outputs. Therefore we provide a second (weaker) theorem that states that at least for some input not all outputs are allowed:

```

theorem _gcdpPostNotTrivialSome()
 $\Leftrightarrow \exists m:\text{nat}, n:\text{nat} \text{ with } (m \neq 0) \vee (n \neq 0). (\exists \text{result}:\text{nat}. (\neg(\text{result} = \text{gcd}(m, n))));$ 

```

In our case, both theorems hold:

```

Executing gcdp( $\mathbb{Z}$ ,  $\mathbb{Z}$ ) with all 49 inputs.
Execution completed for ALL inputs (65 ms, 48 checked, 1 inadmissible).
Executing _gcdpPostNotTrivialSome().
Execution completed (0 ms).

```

```

10 fun gcd(m:nat,n:nat): nat
11   requires m ≠ 0 ∨ n ≠ 0;
12 = choose result:nat with
13   divides(result,m) ∧ divides(result,n) ∧
14   ¬∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;
15
16 val g:nat = gcd(N,N-1);
17
18 theorem gcd0(m:nat) ⇔ m ≠ 0 ⇒ gcd(m,0) = m;
19 theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);
20 theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);
21
22 proc gcdp(m:nat,n:nat): nat
23   requires m ≠ 0 ∨ n ≠ 0;
24   ensures result = gcd(m,n);
25 {
26   var a:nat = m;

```

Figure 3: Does Operation Precondition Hold?

Is output uniquely determined? This theorem allows to detect unintentionally underspecified operations, i.e., operations that by a logical error in the postcondition allow multiple outputs even when this is not desired. In more detail, the corresponding theorem states that for every legal input there exists at most one legal output:

```

theorem _gcdpPostUnique(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ ∀result:nat with result = gcd(m, n).
  (∀_result:nat with let result = _result in (result = gcd(m, n)).
   (result = _result));

```

In our example, this theorem indeed holds:

Executing `_gcdpPostUnique(\mathbb{Z} , \mathbb{Z})` with all 49 inputs.

Execution completed for ALL inputs (52 ms, 48 checked, 1 inadmissible).

This theorem need not generally hold, because a function might be intentionally underspecified such that multiple outputs are acceptable. However, if the theorem holds, the procedure has no choice in which value it may return.

Furthermore, the folder “Verify specification preconditions” contains tasks that verify that the specification is actually well-formed, in particular, that all operations in it are applied only to arguments that satisfy the operation’s precondition

Does operation precondition hold? If we select in our example this task, the editor window indicates by a green squiggly line the expression `gcd(m,n)` in the specification from which this precondition was generated and by a grey squiggly line the corresponding precondition

```
requires m ≠ 0 ∨ n ≠ 0;
```

in the definition of that operation (see Figure 3). The task is defined as

```

theorem _gcdp_PreEns0(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ letpar m = m, n = n in ((m ≠ 0) ∨ (n ≠ 0));

```

The validity of this theorem demonstrates that for the arguments of `gcdp` (which satisfy the operation's precondition) this property indeed holds.

2.7 Verifying Implementations

To further validate the correctness of algorithms (and as a prerequisite of subsequent general correctness proofs), RISCAL provides a *verification condition generator (VCG)*. This VCG generates from the definition of an operation (including its specification and other formal annotations) a set of verification conditions, i.e., formulas whose validity implies that, for all arguments that satisfy the precondition of the operation, its execution terminates and produces a result that satisfies its postcondition. The validity of these theorems can be checked within RISCAL itself.

Generating and checking verification conditions in RISCAL serves a particular purpose: it not only ensures that the algorithm works as expected (this has been demonstrated already by checking the algorithm) but also establishes that all annotations (specifications, loop invariants, termination terms) are strong enough to verify the correctness of the algorithm by formal proof. If checking the verification conditions succeeds, this indeed demonstrates that such a proof is possible in the particular model (determined by the concrete values assigned to the type bounds) in which the check has taken place. However, the success of such a check gives also reason to believe (at least increases our confidence) that such a proof may be possible for the infinite class of *all* possible models (i.e., for all infinitely many possible values of the type bounds). At least, if a check fails, we know that there is no point in attempting such a general proof before the error exhibited in the particular model has been fixed.

Our default assumption is that (due to some problem in the definition of the operation respectively in its specification or annotations) a correctness proof will fail. Therefore RISCAL tries to help the user to find the reason of such a failure by the following strategy:

- Rather than generating a single big verification condition whose validity ensures the overall correctness of the algorithm, RISCAL generates a lot of smaller verification conditions each of which demonstrates some aspect of correctness. Thus, if a particular condition fails, we do not only know that the overall correctness of the algorithm cannot be established but we can focus on that aspect of correctness expressed by the corresponding condition. Thus we hope that by gradual and diligent work more and more aspects of correctness can be established until ultimately the full correctness of the algorithm has been shown.
- Each verification condition is internally linked to those parts of the algorithm from which the condition has been generated and that are therefore relevant for its validity. These parts are visualized in the editor such that the user may get a quick intuition on which parts of the algorithm she has to concentrate.

This strategy will be illustrated in an example below.

In the following, we will mainly focus on the correctness of imperative procedures whose execution consists of a sequence of basic commands (the verification calculus also covers the

$$\begin{aligned}
wp(x:=E, Q) &= \text{let } x = E \text{ in } Q \\
wp(C_1; C_2, Q) &= wp(C_1, wp(C_2, Q)) \\
wp(\text{if } E \text{ then } C, Q) &= (E \Rightarrow wp(C, Q)) \wedge (\neg E \Rightarrow Q) \\
wp(\text{if } E \text{ then } C_1 \text{ else } C_2, Q) &= (E \Rightarrow wp(C_1, Q)) \wedge (\neg E \Rightarrow wp(C_2, Q)) \\
wp(\text{while } E \text{ do invariant } I; \text{ decreases } M; C, Q) &= \\
&\quad \text{letpar } old_x = x, \dots \text{ in} \\
&\quad (\forall x, \dots I \wedge \neg E \Rightarrow Q) \wedge \tag{1} \\
&\quad I \wedge (\forall x, \dots I \wedge E \Rightarrow wp(C, I)) \wedge \tag{2a,2b} \\
&\quad (\forall x, \dots I \Rightarrow M \geq 0) \wedge (\forall x, \dots I \wedge E \Rightarrow \text{let } m = M \text{ in } wp(C, M < m)) \tag{3a,3b}
\end{aligned}$$

Figure 4: The Weakest Precondition $wp(C, Q)$

correctness of declarative functions, but the principles outlined above have been in RISCAL best elaborated for procedures). We start by outlines some basic principles of verification condition generation as implemented in RISCAL. The VCG of RISCAL is fundamentally based on Dijkstra’s weakest precondition (wp) calculus. In order to prove the correctness of a procedure

```
proc p(x:T1):T2 requires P(x); ensures Q(x,result) { C; return r; }
```

we generate a theorem

```
theorem _p_Corr(x:T1) requires P(x);  $\Leftrightarrow$  wp(C, let result = r in Q);
```

where $wp(C, Q)$ is a formula generated from the body C of the procedure and its postcondition Q (called “the weakest precondition of C with respect to Q ”). In a nutshell, the theorem states that every argument x that satisfies the precondition P of the procedure also satisfies that weakest precondition (as discussed above, we generate not only one such big formula but a lot of smaller ones; however, in the following discussion we will stick to the basic calculus which produces a single weakest precondition).

Figure 4 shows how the weakest precondition $wp(C, Q)$ of command C with respect to postcondition Q is generated for some basic commands C . For most commands, the computation of the weakest precondition proceeds by processing the structure of the command recursively giving rise to simple propositional formulas. However, the situation is fundamentally different in case of a loop `while E do C` which we assume to be annotated by an invariant I (a formula) and termination measure M (an integer term). Here the weakest precondition first captures for all program variables x, \dots visible in the loop in the logical variables old_x, \dots the values of the program variables (the invariant I may thus refer to these values). The remainder of the weakest precondition is a conjunction of the following formulas:

- Condition (1) $(\forall x, \dots I \wedge \neg E \Rightarrow Q)$ states that the postcondition Q must follow from invariant I and the negation of the loop condition E for all possible values of the program variables x, \dots . All essential information for proving Q thus must come from I which

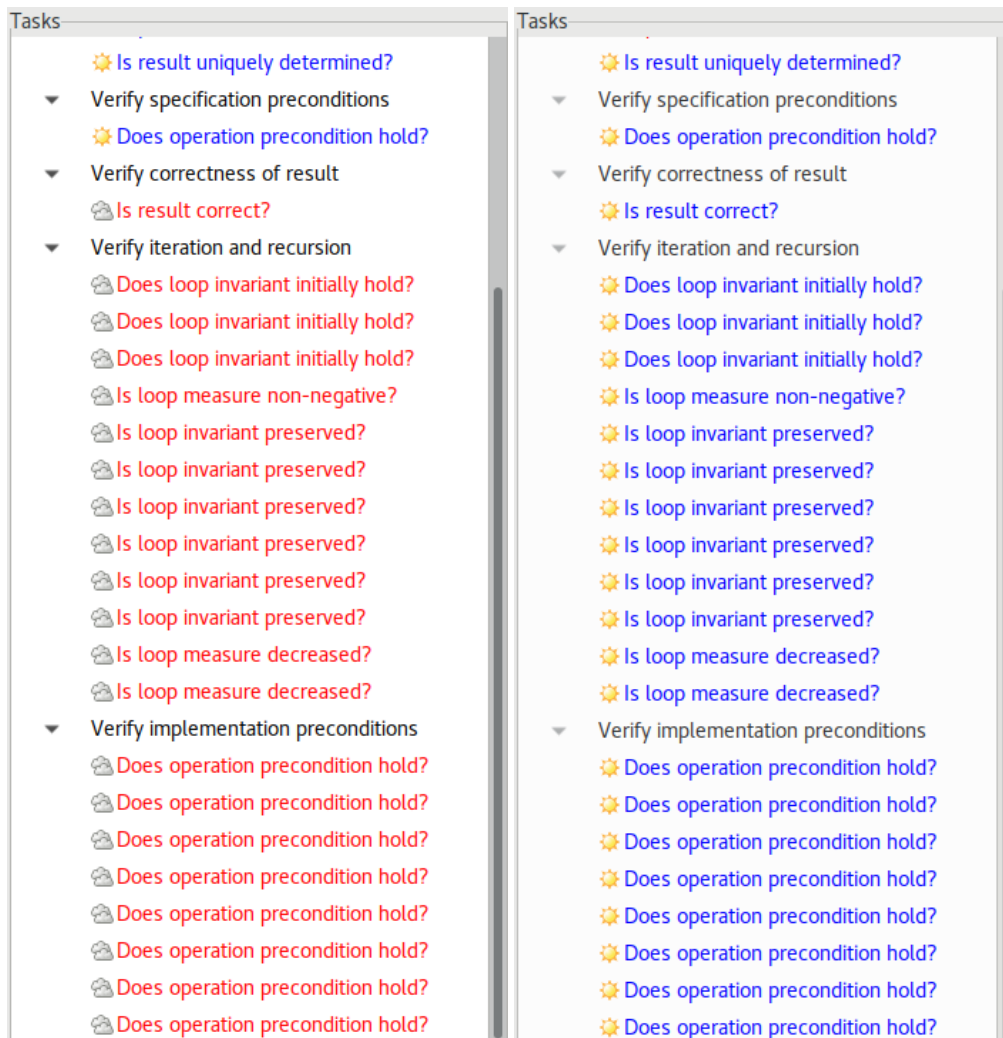



Figure 5: The Correctness of the Algorithm

can be considered as the “specification” of the loop. If the loop does not modify some program variable x declared in the body of the procedure, it is advisable to include in I the formula $x = \text{old}_x$ to preserve in the verification condition all information that the context provides for this variable.

- Conditions (2a) I and (2b) $(\forall x, \dots I \wedge E \Rightarrow wp(C, I))$ show by a kind of induction proof that formula I is indeed an invariant, i.e., that that holds after the termination of the loop no matter how often the loop has been iterated. Condition (2a) represents the induction base that demonstrates that the invariant holds initially, i.e., after 0 iterations. Condition (2b) shows that under the assumption that I holds (after an arbitrary number of iterations, the induction base) and that E holds (the loop body is entered), I holds after the execution of the loop body again (i.e., after one more iteration, the induction step).
- Conditions (3a) $(\forall x, \dots I \Rightarrow M \geq 0)$ and (3b) $(\forall x, \dots I \wedge E \Rightarrow \text{let } m = M \text{ in } wp(C, M < m))$ prove that the loop terminates. Condition (3a) shows that the termination measure M remains non-negative after arbitrary many iterations of the loop (i.e., whenever invariant I holds). Condition (3b) shows that, if the loop body C is executed, the new value of M is smaller than the old value m of M before the execution.

In RISCAL, these five parts of the weakest precondition of a loop are indeed generated as distinct verification conditions. Furthermore, RISCAL also generates conditions (not shown above) that demonstrate that the evaluation of each expression E produces a valid result, i.e., that all operations in E are only applied to arguments that satisfy the precondition of the operation.

In case of the greatest common divisor procedure `gcdp` that was introduced in Section 2.2 and whose specification was validated in Section 2.6, RISCAL generates the verification conditions listed in Figure 5. These conditions are initially indicated as red tasks; having successfully checked them in RISCAL (by double clicking the tasks or right-clicking the task and selecting in the pop-up menu the entry  “Execute Task”), the tasks turn blue.

In more detail, we have the following verification conditions:

Is result correct? This task checks that the precondition of the procedure implies (the core of) the weakest precondition of its body with respect to its postcondition (excluding loop-oriented tasks and operation preconditions, see below). In case of a procedure body with a single loop, this condition essentially stems from Condition (1) in Figure 4. Its definition (displayed by right-clicking the condition and selecting in the pop-up menu the entry “Print Definition”) is as follows:

```
theorem _gcdp_5_CorrOp0(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in
  (letpar old_a = a, old_b = b in
    (∀a:nat, b:nat. (((((a ≠ 0) ∨ (b ≠ 0)) ∧
                      (gcd(a, b) = gcd(old_a, old_b))) ∧
                      (¬((a > 0) ∧ (b > 0))))) ⇒
      (let result = if a = 0 then b else a in
        (result = gcd(m, n))))))));
```

```

22 proc gcdp(m:nat,n:nat): nat
23   requires m≠0 ∨ n≠0;
24   ensures result = gcd(m,n);
25 {
26   var a:nat = m;
27   var b:nat = n;
28   while a > 0 ∧ b > 0 do
29     invariant a ≠ 0 ∨ b ≠ 0;
30     invariant gcd(a,b) = gcd(old_a,old_b);
31     decreases a+b;
32   {
33     if a > b then
34       a = a%b;
35     else
36       b = b%a;
37   }
38   return if a = 0 then b else a;
39 }

```

Figure 6: Is Result Correct?

Furthermore, by selecting the condition with a single click, the editor displays the view shown in Figure 6. The postcondition to be proved is underlined in green while grey underlines indicate those parts of the program that contributed to the derivation of the weakest precondition (in case of a loop, only the loop expression is underlined; actually, most of the verification condition comes from the invariant of the loop). It should be noted that the body of the loop does here play no role, because for the derivation of the weakest precondition only the invariant of the loop is considered.

If a procedure lists multiple postconditions, a separate task is generated for each. It is recommended to split a postcondition into multiple conditions, whenever possible, in order to investigate the reason why a verification condition of this kind is invalid.

Does loop invariant initially hold? Furthermore, two tasks are generated to show that each of the listed invariants holds initially, i.e., before the loop body is executed for the first time (Condition (2a) in Figure 4). In case of the first invariant, its definition is

```

theorem _gcdp_5_LoopOp0(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in ((a ≠ 0) ∨ (b ≠ 0)));

```

where again green and gray underlines in the editor mark the relevant parts of the program.

Is loop measure non-negative? Likewise, a task is generated to show that the listed termination measure is always non-negative (Condition (3a) in Figure 4). Its definition is

```

theorem _gcdp_5_LoopOp2(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in
  (letpar old_a = a, old_b = b in
    (∀a:nat, b:nat. (((a ≠ 0) ∨ (b ≠ 0)) ∧

```

<pre> 22 proc gcdp(m:nat,n:nat): nat 23 requires m≠0 ∨ n≠0; 24 ensures result = gcd(m,n); 25 { 26 var a:nat = m; 27 var b:nat = n; 28 while a > 0 ∧ b > 0 do 29 invariant a ≠ 0 ∨ b ≠ 0; 30 invariant gcd(a,b) = gcd(old_a,old_b); 31 decreases a+b; 32 { 33 if a > b then 34 a = a%b; 35 else 36 b = b%a; 37 } 38 return if a = 0 then b else a; 39 } </pre>	<pre> 22 proc gcdp(m:nat,n:nat): nat 23 requires m≠0 ∨ n≠0; 24 ensures result = gcd(m,n); 25 { 26 var a:nat = m; 27 var b:nat = n; 28 while a > 0 ∧ b > 0 do 29 invariant a ≠ 0 ∨ b ≠ 0; 30 invariant gcd(a,b) = gcd(old_a,old_b); 31 decreases a+b; 32 { 33 if a > b then 34 a = a%b; 35 else 36 b = b%a; 37 } 38 return if a = 0 then b else a; 39 } </pre>
--	--

Figure 7: Is Loop Invariant Preserved?

$$(\gcd(a, b) = \gcd(\text{old_a}, \text{old_b})) \Rightarrow ((a+b) \geq 0)))));$$

Again underlines in the editor mark the relevant parts of the program.

Is loop invariant preserved? For each invariant, at least one task is generated to show that the invariant is preserved by each iteration of the loop (Condition (2b) in Figure 4). In our program, actually, two tasks are generated, one for each path through the conditional command in the body of the loop. In case of the first invariant and the first path, we have the following definition:

```

theorem _gcdp_5_LoopOp3(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in
  (letpar old_a = a, old_b = b in
    (∀a:nat, b:nat. (((((a ≠ 0) ∨ (b ≠ 0)) ∧
      (gcd(a, b) = gcd(old_a, old_b))) ∧
      ((a > 0) ∧ (b > 0))) ⇒
      ((a > b) ⇒ (let a = a%b in ((a ≠ 0) ∨ (b ≠ 0)))))))));

```

A similar task is generated for the second branch. The editor highlights for each task the corresponding branch of the conditional statement (see Figure 7).

Is loop measure decreased? Likewise, for the loop measure a task is generated to show that the measure is decreased in each branch (Condition (3b) in Figure 4). In case of the first branch, we have the following definition:

```

theorem _gcdp_5_LoopOp7(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in
  (letpar old_a = a, old_b = b in

```



```

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

22 proc gcdp(m:nat,n:nat): nat
23   requires m ≠ 0 ∨ n ≠ 0;
24   ensures result = gcd(m,n);
25 {
26   var a:nat = m;
27   var b:nat = n;
28   while a > 0 ∧ b > 0 do
29     invariant a ≠ 0 ∨ b ≠ 0;
30     invariant gcd(a,b) = gcd(old_a,old_b);
31     decreases a+b;
32   {
33     if a > b then
34       a = a%b;
35     else
36       b = b%a;
37   }
38   return if a = 0 then b else a;
39 }

```

Figure 8: Does Operation Precondition Hold?

$$\begin{aligned}
& (\forall a:\text{nat}, b:\text{nat}. (((((a \neq 0) \vee (b \neq 0)) \wedge \\
& \quad (\text{gcd}(a, b) = \text{gcd}(\text{old}_a, \text{old}_b)))) \wedge \\
& \quad ((a > 0) \wedge (b > 0))) \Rightarrow \\
& (\text{letpar } _measure = a+b \text{ in } ((a > b) \Rightarrow \\
& \quad (\text{let } a = a\%b \text{ in } ((a+b) < _measure))))))));
\end{aligned}$$

Again underlines in the editor mark the relevant parts of the program.

Does operation precondition hold? For each application of a (predefined or user-defined) operation, the validity of its precondition has to be checked, giving rise to a number of corresponding verification conditions. For instance, the first generated condition has the following definition:

```

theorem _gcdp_5_PreOp0(m:nat, n:nat)
requires (m ≠ 0) ∨ (n ≠ 0);
⇔ let a = m in (let b = n in
  (((a ≠ 0) ∨ (b ≠ 0)) ⇒
    (letpar m = a, n = b in ((m ≠ 0) ∨ (n ≠ 0)))));

```

As Figure 8 illustrates, this condition refers to the application $\text{gcd}(a, b)$ of the operation gcd in the loop invariant; the corresponding precondition of the operation is also marked.

All conditions can be easily checked; for $N = 20$, however, due to the occurrence of the quantifiers in the conditions, some of the checks (e.g. the preservation of the invariants) start to take some time (1–2 minutes) which suggests the use of smaller models (for $N = 10$ the checks still need only 2–3 seconds) or the application of the parallel checking mechanism.

2.8 Visualizing Execution Traces

If RISCAL is started and visualization is enabled (see Section A.5), the graphical user interface depicts a row “Visualization” with two (mutually exclusive) check box options “Trace” and “Tree”.

If any of these options is selected, every run/check of a RISCAL operation (procedure, function, predicate, theorem) opens a window in which a visualization of this operation is displayed *provided that*

- the option “Nondeterminism” is *not* selected,
- the options “Multi-Threaded” and “Distributed” are *not* selected.

In other words, visualization is restricted to the deterministic execution of RISCAL operations in the sequential mode of the RISCAL software. The user may configure the size of the visualization area by the fields “Width” and “Height” (independent of this setting, the minimum size of a visualization area is 100 times 100 pixels).

The option “Tree” triggers the visualization of evaluation trees that will be discussed in Section 2.9. In the remainder of this section, we will describe the trace-based visualization of procedures that is triggered by the option “Trace”. This trace-based visualization displays the changes of variables by the commands in the body of a procedure; for functions (including predicates and theorems) whose result is determined by the evaluation of an expression the computation of the result is displayed as a single step (except that also calls of other operations are displayed). This mode of visualization is thus mainly useful for understanding the operational behavior of a procedure.

We demonstrate this by a visualization of the following RISCAL model:

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

type index =  $\mathbb{Z}[-N,N]$ ;
type elem =  $\mathbb{Z}[-M,M]$ ;
type array = Array[N, elem];

proc cswap(a:array, i:index, j:index): array
{
  var b:array = a;
  if b[i] > b[j] then
  {
    var x:elem := b[i];
    b[i] := b[j];
    b[j] := x;
  }
  return b;
}

proc bubbleSort(a:array): array
{
  var b:array = a;
```

```

for var i:index := 0; i < N-1; i := i+1 do
{
  for var j:index := 0; j < N-i-1; j := j+1 do
    b := cswap(b,j,j+1);
  }
return b;
}

```

This model implements a version of the “Bubble sort” algorithm as a procedure `bubbleSort` that uses an auxiliary procedure `cswap` for the conditional swap of two array elements.

If we select the operation `bubbleSort` and set with the button “Other Values” the parameters N and M to 4 and 3, respectively, the first check of the procedure is performed with the argument array $a = [-3, -3, -3, -3]$. Since here actually no swap is performed, we close the window that pops up immediately (by clicking on the top-right button in the window frame). This lets the execution proceed to the second check with $a = [-2, -3, -3, -3]$ whose execution is displayed by another window; it is this window that is shown on the top of Figure 9.

The window displays in the title the operation invocation `bubbleSort([-2, -3, -3, -3])` whose execution is visualized; the tag “Level 0” indicates that this is the top-level of the execution (every entry into the visualization of a nested procedure call increases this level by one). In the window, we see a directed graph (a linear sequence of nodes) that are connected by directed edges (arrows) and that are laid out in a two-dimensional manner from left to right and top to bottom. This graph has three kinds of nodes:

- *Numbered nodes* represent the sequence of states constructed by performing assignments to the variables of the procedure; such a node is always the target of a solid arrow (see below). By hovering with the mouse pointer over such a node, a small window pops up that displays the values of the various variables in that state (see the node numbered 17 in the figure).
- *Empty nodes* represent “intermediate” states that have the same variable values as the previous state; such a node is always the target of a dashed arrow (see below). An empty node is displayed separately, because it indicates an event that helps to understand how the state indicated by the next numbered node has been computed (see the explanation of the dashed arrows below).
- *Call nodes* display a small window with a graph symbol; such a node represents the call of another operation, the header of this window displays the call itself. A graph node is always the target of a dashed arrow (see the explanations below).

Nodes may be selected by a mouse click and moved to another location in the display.

Nodes are connected by two types of arrows:


- *Solid arrows* (which lead to a numbered node) represent changes from one state to another by the modification of a variable (thus generally some variable value in the target node is different from the value of the corresponding variable in the source node). The label associated to the arrow displays the name of the command that is responsible for the change

and in the second line (within parentheses) the new value of the variable. State changing commands are variable assignments but also various kinds of **choose** statements that nondeterministically set variables to values satisfying given conditions.

- *Dashed arrows* (which lead to an empty node) represent events within the change from one state to the next that contribute to the change but do not change the state themselves. We consider as such events on the one hand the testing of boolean conditions (respectively the matching of recursive structures against patterns) that direct the control flow of statements (conditionals and loops) and on the other hand the application of an operation (procedure or function) whose execution yields a value. The label associated to the arrow displays the test expression respectively the application expression in the first line and the result of the test respectively of the application (within parentheses) in the second line.

The labels associated to the arrows (actually to the source nodes of the arrows) may be selected by a mouse click and moved to another location in the display. In fact, some of the arrow labels displayed in the figure have been manually moved for greater clarity.

By double-clicking on a call node which represents the application of an operation the content of the window is modified to visualize the execution of that operation. The left diagram at the bottom of Figure 9 displays the execution corresponding to one application of `cswap(b, j, j+1)` where the test determines that the elements at positions j and $j + 1$ are not in the right order such that it is necessary to swap the two elements by three assignments; the right diagram displays an application where the test determines that the elements are already in the right order such that no swap is necessary. The tag “Level 1” in the titles of the windows indicates that the visualization refers to the application of an operation that was invoked on “Level 0”. If the “Level 1” operation would involve another operation application, it would contain another call node; by double-clicking on that node, we would move to “Level 2” and so on. A double click on any empty part of the window moves the display back to the previous level.

By closing the window (via a click on the top right button of the window frame) the visualization is terminated and the execution machine continues with the execution of the next invocation of the operation being checked. By clicking on the button  (Stop Execution) in the main window, this process can be prematurely terminated (such that it is not necessary to step through the visualizations of all possible calls of the operation).

The visualization of the execution trace of a procedure also displays the application of all operations (typically functions and predicates) applied when checking the formal annotations (pre- and postconditions, invariants, termination measures, assertions) of the procedure. If this is not desired, those annotations should be temporarily commented out.

2.9 Visualizing Evaluation Trees

While the previous section discussed the visualization of execution traces of procedures, this section is dedicated to the visualization of the evaluation of logic formulas. As an example, we visualize in the following the evaluation of the formula introduced by the following definition of predicate `forallPexistsQFormula()`:

```
val N = 4;
```

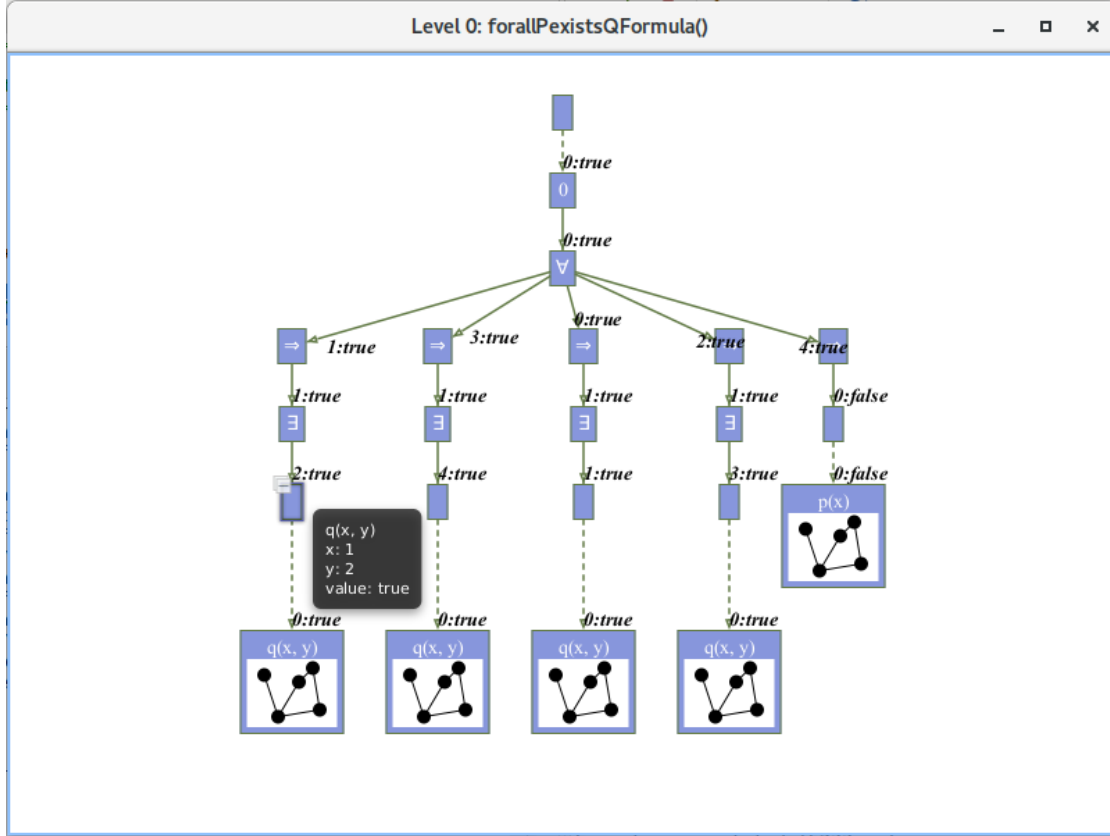


Figure 10: The Visualization of the Evaluation of $(\forall x. p(x) \Rightarrow \exists y. q(x, y))$

```

type T = N[N];
pred p(x:T)  $\Leftrightarrow$  x < N;
pred q(x:T,y:T)  $\Leftrightarrow$  x+1 = y;
pred forallPexistsQFormula()  $\Leftrightarrow$   $\forall x:T. p(x) \Rightarrow \exists y:T. q(x,y)$ ;

```

For this, we select in the RISCAL user interface, the operation `forallPexistsQFormula()`, the visualization option “Tree”, and press the “Start Execution” button labeled with the green arrow. After the execution of the operation has finished, the window depicted in Figure 10 pops up (by closing this window, via a click on the top right button of the window frame the visualization is terminated).

This header of the window displays the predicate `forallPexistsQFormula()` whose evaluation is visualized; the tag “Level 0” indicates that this is the top level of the evaluation (every entry into the nested visualization of a predicate application increases this level by one). The main panel of the window depicts a tree whose nodes are laid out layer by layer from top to bottom with directed edges (arrows) connecting the nodes from one layer to the next one.

Nodes A tree may have five kinds of nodes:

- *The root node* represents the evaluation of the predicate for all possible arguments. By hovering with the mouse pointer over this node, a small window pops up that displays the definition of the predicate and whether its execution for all possible arguments resulted in an error or not. If there are n possible arguments and no error has occurred, the root has n children each of which represents an evaluation with one of the arguments. However, if an error occurs, the root one has one child representing the evaluation with the argument that triggered the error.

In particular, if the operation is a “theorem” (keyword `theorem`) rather than a plain “predicate” (keyword `pred`), the evaluation triggers an error if there is any argument for which the formula defining the theorem has truth value “false”. In that case only the child representing this evaluation is displayed.

- *Numbered nodes* are children of the root node each of which represents the evaluation of the predicate for one particular argument. Nodes are numbered from 0 on, i.e., if there are n nodes, they have numbers $0, \dots, n - 1$ (the nodes are not necessarily laid out in the order of the numbers). By hovering with the mouse pointer over such a node, a small window pops up that displays the definition of the predicate, the value(s) of its argument(s), and the resulting truth value.
- *Labeled nodes* represent propositional formulas or quantified formulas where the label denotes the formula’s outermost logical symbol (logical connective or quantifier). By hovering with the mouse pointer over such a node, a small window pops up that displays the formula, the values of its free variables (actually the values of all variables visible at the occurrence of the formula), and the truth value of the formula.
- *Empty nodes* represent formulas whose outermost symbol is not a logical connective or quantifier, in particular (but not exclusively) atomic formulas. By hovering with the mouse pointer over such a node, the same information as for labeled nodes is displayed.
- *Call nodes* display a small window with a graph symbol; such a node represents the application of a user-defined predicate. The header of this window displays the application itself. By hovering with the mouse pointer over such a node, the same information as for a labeled node is displayed. By double-clicking on such a node, the visualization switches from the current formula to the formula defining the predicate (see the paragraph “Nested Visualization” below).

Call nodes are also generated for the applications of functions and procedures; however, the evaluation trees depicted for these entities are typically not very meaningful.

Nodes may be selected by a mouse click and moved to another location in the display.

Arrows Nodes are connected by two types of arrows:

- *Solid arrows* connect a formula to all those subformulas (respectively instances of subformulas) whose truth values are relevant for the truth value of the whole formula.

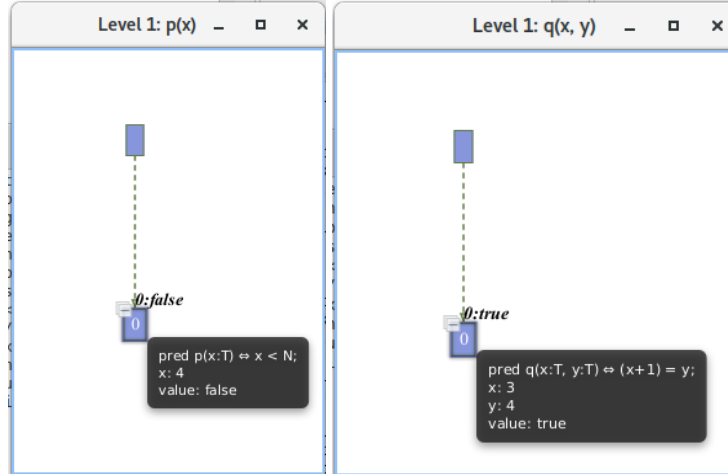


Figure 11: The Visualization of Predicates

- *Dashed arrows* are used at the top of the tree to connect the root node to the numbered nodes (representing the individual invocations of the predicate) and at the bottom of the tree to connect empty nodes (representing atomic formulas) to call nodes (representing the invocations of the predicates and functions within the atomic formula).

The targets of all arrows are annotated with labels of form $n:v$ where n is the number of the subexpression and v is its value. Numbers start with 0 (the arrows are not necessarily laid out in the order of the numbers): a formula with a unary logical connective has one arrow with number 0, a formula with a binary connective has at most two arrows with numbers 0 (the first subformula) and 1 (the second subformula); a quantified formula whose variable can be assigned n values has at most n arrows with numbers $0, \dots, n-1$ denoting the individual instances of the formula's body. Furthermore, if an atomic formula contains n procedure/function applications, the corresponding empty node has n arrows to the call nodes corresponding to these applications.

The labels associated to the arrows (actually to the target nodes of the arrows) may be selected by a mouse click and moved to another location in the display.

Nested Visualization By double-clicking on a call node which represents the application of an operation (in particular a predicate), the content of the window is modified to visualize the execution of that operation (the depicted evaluation tree is essentially only helpful for predicates since only these are defined by formulas). For instance, the left diagram in Figure 11 displays the evaluation of the predicate application $p(x)$ in branch 4 of Figure 10, while the right diagram displays the evaluation of $q(x, y)$ in branch 3. The tag “Level 1” in the titles of the window indicates that the visualization refers to an operation that was invoked on “Level 0”. If the “Level 1” operation would involve another operation application, it would contain another call node; by double-clicking on that node, we would move to “Level 2” and so on. A double click on any empty part of the window moves the display back to the previous level.

Tree Pruning Every evaluation tree is pruned such that it only displays the information necessary to understand how its truth value was derived:

- If the truth value of a conjunction ($F_1 \wedge F_2$) is “false”, only the first subformula is displayed whose truth value is “false”.
- If the truth value of a disjunction ($F_1 \vee F_2$) is “true”, only the first subformula is displayed whose truth value is “true”.
- If the truth value of an implication ($F_1 \Rightarrow F_2$) is “true”, only one subformula is displayed (either the “false” antecedent F_1 , or, if this antecedent is “true”, then the “true” consequent F_2).
- If the truth value of a universally quantified formula ($\forall x. F$) is “false”, only one instance $F[x \mapsto a]$ is displayed, where a is the first value encountered in the evaluation that makes F “false”.
- If the truth value of an existentially quantified formula ($\exists x. F$) is “true”, only one instance $F[x \mapsto a]$ is displayed, where a is the first value encountered in the evaluation that makes F “true”.

For example, in the tree depicted in Figure 10, all “true” implications ($p(x) \Rightarrow \exists y. q(x, y)$) are pruned: in branch 0, only the evaluation tree for the “false” atomic formula $p(x)$ is displayed; in all other branches, the evaluation tree for the “true” existential formula ($\exists y. q(x, y)$) is displayed. In the evaluation tree of every such existential formula only one branch is displayed corresponding to one “true” instance of atomic formula $q(x, y)$. By hovering the mouse pointer over the nodes, the corresponding variable values are displayed.

We further illustrate the role of tree pruning by the evaluation of the two “theorems” `forallPQR()` and `forallPQR2()` depicted below:

```
val N = 4;
type T = N[N];
pred p(x:T) ⇔ x < N;
pred p2(x:T) ⇔ x ≤ x;
pred q(x:T,y:T) ⇔ x+1 = y;
pred r(x:T,y:T) ⇔ x = y+1;
theorem forallPQR() ⇔ ∀x:T. p(x) ⇒ x = 0 ∨ (∃y:T. q(x,y)) ∧ (∃y:T. r(x,y));
theorem forallPQR2() ⇔ ∀x:T. p2(x) ⇒ x = 0 ∨ (∃y:T. q(x,y)) ∧ (∃y:T. r(x,y));
```

The first theorem is indeed valid as illustrated by the top diagram in Figure 12. Since the universally quantified formula is true, all branches of this formula are depicted. In each branch, the implication is true which leads to a continuation with a single branch. In this branch, the disjunction is true, which again leads to a continuation with a single branch. In this branch, the conjunction is true, which now leads to a split into two branches; each of these branches contains a true existential formula, for which it again suffices to depict a single branch.

On the contrary, the second theorem is invalid as illustrated by the bottom diagram in Figure 12. Since the universally quantified formula is false, only one branch of this formula is depicted.

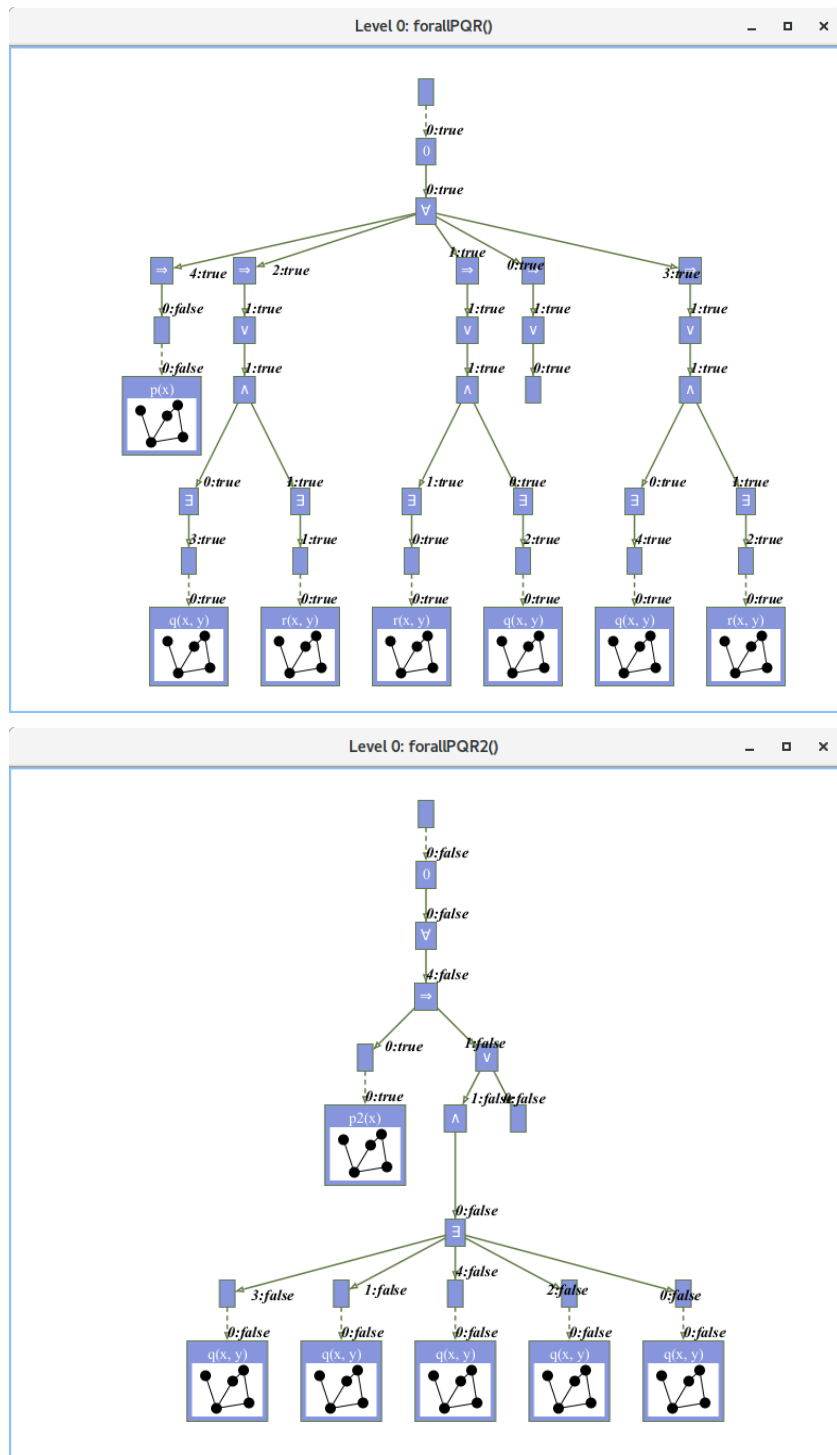


Figure 12: Pruned Evaluation Trees

The implication in this branch is false which leads to a split into two branches, the first of which is immediately true. In the second branch, we have a false disjunction, which also leads to a split into two branches, of which the first one is immediately false. In the second branch, we have a false conjunction, of which only the first branch with a false existential formula needs to be shown. To demonstrate the invalidity of this existential formula, all its branches have to be depicted.

Large Trees The implementation of the algorithms employed by RISCAL for layouting the evaluation trees has two drawbacks:

- The implementation becomes very slow for larger trees. This is deplorable all the more, as the layout is computed by that thread that also handles the graphical user interface; thus the RISCAL interface is blocked during this computation. In order to mitigate this problem, we attempt to limit the blocking time by visualizing only trees up to a certain maximum number of nodes (currently 250); for larger trees, RISCAL refuses any visualization attempt. However, even with this limit, the layout time may be still substantial; thus it is recommended to attempt the visualization first for very small model sizes before proceeding to larger ones.
- If the visualization area is too small, the layout algorithm may only compute a partial layout or no layout at all; this results in the placement of some/all nodes on the left upper corner of the visualization window. To overcome this problem, the user may choose the size of this visualization area by setting the values “Width” and “Height” in the RISCAL user interface; if the resulting visualization is unsatisfactory, larger dimensions may be chosen. These dimensions may also vastly exceed the dimensions of the physical screen; e.g., we may choose a layout area of 16000 times 4000 pixel, even if our physical screen size is just 1920 times 1080. In this case, the window will be maximized to the physical screen size with scroll bars allowing to navigate within the layout area.

To illustrate the second problem and its solution, we show the visualization of formula `forallPQR()` introduced in the previous section for $N = 12$:

```
val N = 12;
...
theorem forallPQR()  $\Leftrightarrow$ 
   $\forall x:T. p(x) \Rightarrow x = 0 \vee (\exists y:T. q(x,y)) \wedge (\exists y:T. r(x,y));$ 
```

The top diagram in Figure 13 displays the visualization for the default dimension 800 times 600 pixels; here the algorithm fails to layout many of the nodes, in particular most of the call nodes, which are placed all into the top left corner. Even the maximization of the window to the physical screen size 1920 times 1080 does not result in a fully satisfactory layout. However, if we set the layout area to 4000 times 1000 pixels we get the visualization depicted in the lower diagram of Figure 13. Clearly the tree exceeds horizontally the dimension of the window (in our illustration reduced from the original maximal screen size to about 800 times 600 pixels again), but we may use the horizontal scroll bar to investigate the currently not visible parts of the tree.

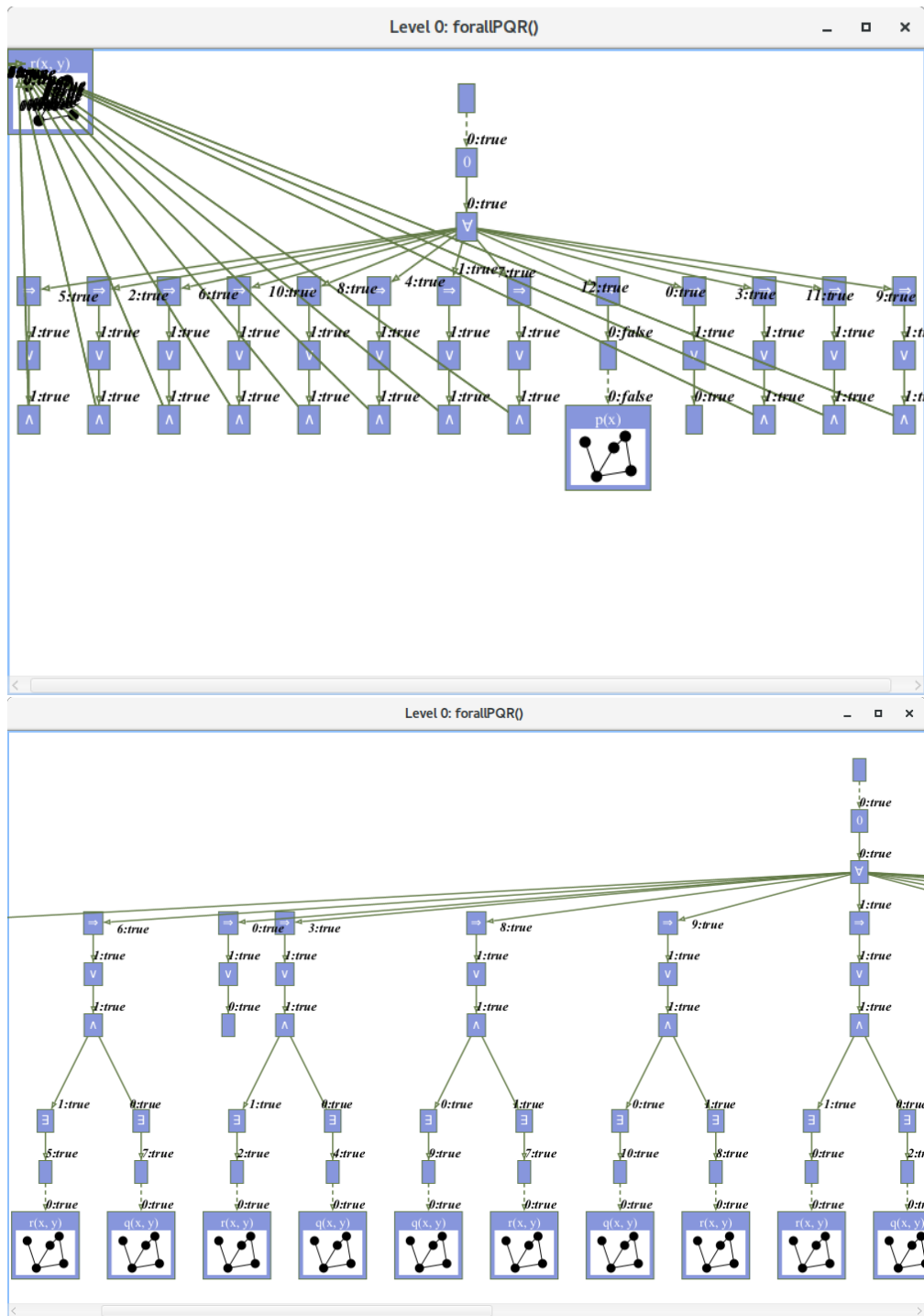


Figure 13: The Visualization of a Large Tree

3 More Examples

We continue by presenting some more examples of RISCAL specifications.

3.1 Linear and Binary Search

We start by modeling the linear search algorithm for looking in an array a of N natural numbers with maximum value M for an element x . The corresponding RISCAL specification is thus based on the following parameters and type declarations (see Appendix C.3 for the full specification):

```
val N:Nat;
val M:Nat;
type int =  $\mathbb{Z}[-N,N]$ ;
type elem =  $\mathbb{N}[M]$ ;
type array = Array[N,elem];
```

Here array denotes the type of all arrays of length N of values of type elem; type int denotes the domain of integers with absolute value less than or equal N (which includes the legal array indices $0, \dots, N-1$ but also the values -1 and N , which will be subsequently required).

The algorithm $\text{search}(a, x)$ for searching in array a for element x can then be modeled, specified, and annotated as follows:

```
proc search(a:array, x:elem): int
  ensures result = -1  $\Rightarrow \forall k:\text{int with } 0 \leq k \wedge k < N. a[k] \neq x$ ;
  ensures result  $\neq -1 \Rightarrow 0 \leq \text{result} \wedge \text{result} < N \wedge$ 
     $a[\text{result}] = x \wedge \forall k:\text{int with } 0 \leq k \wedge k < \text{result}. a[k] \neq x$ ;
{
  var i:int = 0;
  var r:int = -1;
  while i < N  $\wedge r = -1$  do
    invariant  $0 \leq i \wedge i \leq N$ ;
    invariant  $\forall j:\text{int}. 0 \leq j \wedge j < i \Rightarrow a[j] \neq x$ ;
    invariant  $r = -1 \vee (r = i \wedge i < N \wedge a[r] = x)$ ;
    decreases if  $r = -1$  then  $N-i$  else 0;
  {
    if  $a[i] = x$ 
      then  $r := i$ ;
      else  $i := i+1$ ;
  }
  return r;
}
```

The postcondition of search states that there are two cases:

- If the result is -1 , a does not hold x .
- Otherwise, the result is a legal index at which a holds x and it is the smallest such index.

The invariants state

- a range condition on the loop variable i ,
- the fact that all indices less than i do not hold x , and
- the fact that the auxiliary variable r is either -1 or identical to i which is then a legal index at which a holds x .

As long as r is -1 , the loop measure is $N - i$; once r is set to a different value, the measure drops to 0, which demands the immediate termination of the loop.

Selecting $N = 4$ and $M = 3$, the algorithm and the its annotation can be very quickly checked:

```
Using N=4.
Using M=3.
Type checking and translation completed.
Executing search(Array[Z],Z) with all 1024 inputs.
Execution completed for ALL inputs (65 ms, 1024 checked, 0 inadmissible).
```

Moreover, RISCAL generates the verification conditions illustrated in the left part of Figure 14: each of these conditions is indeed valid and can be checked in less than half a second.

To demonstrate the problem arises from underspecification, let us assume that we would have formulated the last invariant of the loop in a slightly weaker way:

```
invariant r = -1 ∨ (r = i ∧ a[r] = x);
```

With that weaker version of the invariant (which has dropped the clause $i < N$), most of the verification conditions are not valid, as indicated in the right part of Figure 14; these checks fail with an error of form

```
ERROR in execution of ...: evaluation of
  a[r]
at unknown position:
  array index 4 out of bounds
```

which indicates an error in some precondition; actually, among the failed conditions are also most of the preconditions. Indeed, clicking on one of the failed preconditions of type “Is index value legal?” lets the editor point to the exact source of the problem (see Figure 15): if $a[i] = x$, the assignment of $i + 1$ to i violates the index access in the evaluation of $a[r]$ in the invariant. Adding the clause $i < N$ to the invariant solves this problem.

If the array a is sorted, we may also apply the binary search algorithm which can be modeled by a recursive function as follows:

```
fun bsearch(a:array, x:elem, from: int, to: int): int
  requires 0 ≤ from ∧ from-1 ≤ to ∧ to < N;
  requires ∀k:int with from ≤ k ∧ k ≤ to-1. a[k] ≤ a[k+1];
  ensures result = -1 ⇒ ∀k:int with from ≤ k ∧ k ≤ to. a[k] ≠ x;
  ensures result ≠ -1 ⇒ from ≤ result ∧ result ≤ to ∧ a[result] = x;
```

▼ Verify correctness of result	▼ Verify correctness of result
☀ Is result correct?	☁ Is result correct?
☀ Is result correct?	☁ Is result correct?
▼ Verify iteration and recursion	▼ Verify iteration and recursion
☀ Does loop invariant initially hold?	☀ Does loop invariant initially hold?
☀ Does loop invariant initially hold?	☀ Does loop invariant initially hold?
☀ Does loop invariant initially hold?	☀ Does loop invariant initially hold?
☀ Is loop measure non-negative?	☁ Is loop measure non-negative?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop invariant preserved?	☁ Is loop invariant preserved?
☀ Is loop measure decreased?	☁ Is loop measure decreased?
☀ Is loop measure decreased?	☁ Is loop measure decreased?
▼ Verify implementation preconditions	▼ Verify implementation preconditions
☀ Is index value legal?	☀ Is index value legal?
☀ Is index value legal?	☀ Is index value legal?
☀ Is index value legal?	☁ Is index value legal?
☀ Is index value legal?	☁ Is index value legal?
☀ Is index value legal?	☁ Is index value legal?
☀ Is index value legal?	☁ Is index value legal?
☀ Is index value legal?	☁ Is index value legal?
☀ Is assigned value legal?	☁ Is assigned value legal?

Figure 14: The Correctness of Linear Search

```

12 proc search(a:array, x:elem): int
13   ensures result = -1  $\Rightarrow \forall k:\text{int with } 0 \leq k \wedge k < N. a[k] \neq x;$ 
14   ensures result  $\neq -1 \Rightarrow 0 \leq \text{result} \wedge \text{result} < N \wedge$ 
15     a[result] = x  $\wedge \forall k:\text{int with } 0 \leq k \wedge k < \text{result}. a[k] \neq x;$ 
16 {
17   var i:int = 0;
18   var r:int = -1;
19   while i < N  $\wedge r = -1$  do
20     invariant  $0 \leq i \wedge i \leq N;$ 
21     invariant  $\forall j:\text{int}. 0 \leq j \wedge j < i \Rightarrow a[j] \neq x;$ 
22     invariant  $r = -1 \vee (r = i \wedge a[r] = x);$ 
23     decreases if r = -1 then N-i else 0;
24   {
25     if a[i] = x
26     then r = i;
27     else i = i+1;
28   }
29   return r;
30 }

```

Figure 15: The Violation of a Condition “Is index value legal?”

```

decreases to-from+1;
= if from > to then
  -1
else
  let m = (from+to)/2 in
  if a[m] = x then m else
  if a[m] < x then bsearch(a, x, m+1, to)
    else bsearch(a, x, from, m-1);

```

```

fun bsearch(a:array, x:elem): int
  requires  $\forall k:\text{int with } 0 \leq k \wedge k < N-1. a[k] \leq a[k+1];$ 
  ensures result = -1  $\Rightarrow \forall k:\text{int with } 0 \leq k \wedge k < N. a[k] \neq x;$ 
  ensures result  $\neq -1 \Rightarrow 0 \leq \text{result} \wedge \text{result} < N \wedge a[\text{result}] = x;$ 
  = bsearch(a, x, 0, N-1);

```

The main algorithm $\text{bsearch}(a, x)$ is based on an auxiliary operation $\text{bsearch}(a, x, \text{from}, \text{to})$ which searches in array a for x within the index interval $[\text{from}, \text{to}]$ assuming that the array is sorted within that interval. The result of the function is -1 , if x does not occur in a within that interval, or some index (not necessarily the smallest one) within that interval at which a holds x . This operation is modeled as a recursive function where the interval size $\text{to} - \text{from} + 1$ shrinks in every recursive invocation, which ensures the termination of the algorithm.

Figure 16 displays the verification conditions generated both for the recursive auxiliary function $\text{bsearch}(a, x, \text{from}, \text{to})$ and the main function $\text{bsearch}(a, x)$. The validity of each condition can be checked in less than half a second.

<ul style="list-style-type: none"> Verify correctness of result <ul style="list-style-type: none"> Is result correct? Is result correct? Verify iteration and recursion <ul style="list-style-type: none"> Is function measure non-negative? Is function measure decreased? Is function measure decreased? Verify implementation preconditions <ul style="list-style-type: none"> Is index value legal? Is index value legal? Does operation precondition hold? Does operation precondition hold? Does operation precondition hold? Does operation precondition hold? Does operation precondition hold? Does operation precondition hold? 	<ul style="list-style-type: none"> Verify correctness of result <ul style="list-style-type: none"> Is result correct? Is result correct? Verify iteration and recursion Verify implementation preconditions <ul style="list-style-type: none"> Does operation precondition hold? Does operation precondition hold?
--	---

Figure 16: The Correctness of Binary Search

3.2 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.4 for the full specification):

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

We make use of the following type definitions

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

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

The insertion sort algorithm is then defined as follows:

```
proc sort(a:array): array
  ensures  $\forall i:\text{nat}. i < N-1 \Rightarrow \text{result}[i] \leq \text{result}[i+1]$ ;
  ensures  $\exists p:\text{Array}[N,\text{index}].$ 
     $(\forall i:\text{index}, j:\text{index}. i \neq j \Rightarrow p[i] \neq p[j]) \wedge$ 
     $(\forall i:\text{index}. a[i] = \text{result}[p[i]])$ ;
```

```

{
  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 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 will be discussed below).

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).

WARNING: not all nondeterministic branches 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).
WARNING: not all nondeterministic branches have been considered.
```

the core information in much shorter time.

To enable a proof-based verification of the algorithm, we have to annotate it with appropriate invariants. To simplify their formulation, we introduce the following predicates:

```
pred sorted(a:array, n:N[N]) ⇔
  ∀i:index. i < n-1 ⇒ a[i] ≤ a[i+1];

pred permuted(a:array, b:array) ⇔
  ∃p:Array[N,index].
    (∀i:index,j:index with i < j ∧ j < N. p[i] ≠ p[j]) ∧
    (∀i:index with i < N. a[i] = result[p[i]]);

pred equals(a:array, b:array, from:N[N], to:Z[-1,N-1]) ⇔
  ∀k:index with from ≤ k ∧ k ≤ to. a[k] = b[k];
```

Here `sorted(a,n)` states that array *a* is sorted in the first *n* positions, `permuted(a, b)` states that array *b* is a permutation of *a*, and `equals(a,b,from,to)` states that arrays *a* and *b* have identical elements in the index interval *[from,to]*.

With the help of these predicates, the postconditions of the algorithm can be reformulated and the invariants expressed as follows:

```

proc sort(a:array): array
  ensures sorted(result, N);
  ensures permuted(a, result);
{
  var b:array = a;
  for var i:N[N]:=1; i<N; i:=i+1 do
    invariant 1 ≤ i ∧ i ≤ N;
    invariant sorted(b, i);
    invariant permuted(a, b);
    invariant equals(b, old_b, i, N-1);
    decreases N-i;
  {
    var x:elem := b[i];
    var j:ℤ[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do
      invariant i = old_i;
      invariant x = old_b[i];
      invariant -1 ≤ j ∧ j ≤ i-1;
      invariant equals(b, old_b, i+1, N-1);
      invariant equals(b, old_b, 0, j+1);
      invariant ∀k:index with j+1 < k ∧ k ≤ i. b[k] = old_b[k-1];
      invariant ∀k:index with j+1 ≤ k ∧ k < i. b[k] > x;
      decreases j+1;
    {
      b[j+1] := b[j];
      j := j-1;
    }
    b[j+1] := x;
  }
  return b;
}

```

Apart from a range condition on the loop variable i , the invariants of the outer loop essentially state that the algorithm's postconditions hold up to position i and that from position i on, array b has not changed. The invariants of the inner loop are a bit more subtle: apart from stating a range condition on the loop variable j , that variable i is not changed by the inner loop (invariants should always explicitly state which variables remain unchanged), and that x is the original value of $b[i]$, they claim the following:

- array b has not been changed from position $i + 1$ on,
- array b has not been changed up to position $j + 1$,

- ▼ Verify correctness of result
 - ☀ Is result correct?
 - ☀ Is result correct?
- ▼ Verify iteration and recursion
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Is loop measure non-negative?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop measure decreased?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Does loop invariant initially hold?
 - ☀ Is loop measure non-negative?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop invariant preserved?
 - ☀ Is loop measure decreased?
- ▼ Verify implementation preconditions
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Does operation precondition hold?
 - ☀ Does operation precondition hold?
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Does operation precondition hold?
 - ☀ Does operation precondition hold?
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Is index value legal?
 - ☀ Is assigned value legal?
 - ☀ Is index value legal?
 - ☀ Is assigned value legal?

Figure 17: The Correctness of Insertion Sort

- in the interval $[j + 1, i]$, the elements of b are shifted by one position,
- in the interval $[j + 1, i[$, the elements of b are greater than x .

From these annotations, RISCAL generates the conditions depicted in Figure 17:

- The two tasks labeled as “Is result correct?” verify the two postconditions.
- The first set of tasks in the folder “Verify iteration and recursion” verifies the correctness of the outer loop under the assumption that the invariant of the inner loop is correct.
- The second set of tasks in the folder “Verify iteration and recursion” verifies the correctness of the inner loop and its invariant.
- The tasks in the folder “Verify implementation preconditions” verify, in addition to the usual operation preconditions, that arrays are only accessed with valid indices (“Is index value legal?”) and that variables are only assigned values in the range of the variable types (“Is assigned value legal?”); the later can be inferred mostly automatically from the types of the values except for the assignments $j := j - 1$ and $i := i + 1$.

As usual, by clicking on the conditions, the editor underlines those parts relevant to the condition. Using the parameter values $N = 4$ and $M = 2$ (considering arrays of length 4 with values in the range $[-2, +2]$), every condition can be with 4 threads checked in less than 5 seconds.

3.3 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.5):

```
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)  $\Leftrightarrow$   $\neg(l \in c \wedge \neg l \in c)$ ;
```

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

```
pred literal(l:Literal)  $\Leftrightarrow$  l $\neq$ 0;
pred clause(c:Clause)  $\Leftrightarrow$   $\forall l \in c. \text{literal}(l) \wedge \text{consistent}(l,c)$ ;
pred formula(f:Formula)  $\Leftrightarrow$   $\forall c \in f. \text{clause}(c)$ ;
pred valuation(v:Valuation)  $\Leftrightarrow$  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)  $\Leftrightarrow$  l $\in$ v;
pred satisfies(v:Valuation, c:Clause)  $\Leftrightarrow$   $\exists l \in c. \text{satisfies}(v, l)$ ;
pred satisfies(v:Valuation, f:Formula)  $\Leftrightarrow$   $\forall c \in f. \text{satisfies}(v,c)$ ;
```

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

```
pred satisfiable(f:Formula)  $\Leftrightarrow$ 
   $\exists v:\text{Valuation}. \text{valuation}(v) \wedge \text{satisfies}(v,f)$ ;
pred valid(f:Formula)  $\Leftrightarrow$ 
   $\forall v:\text{Valuation}. \text{valuation}(v) \Rightarrow \text{satisfies}(v,f)$ ;
```

We define the negation of a formula

```
fun not(f: Formula):Formula =
  { c | c:Clause with clause(c)  $\wedge$   $\forall d \in f. \exists l \in d. \neg l \in c$  };
theorem notFormula(f:Formula)
  requires formula(f);
 $\Leftrightarrow$  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);
 $\Leftrightarrow$  valid(f)  $\Leftrightarrow$   $\neg$ 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  $\exists c \in f. l \in c$ };
```

and the result of setting a literal *l* in formula *f* to “true”:

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

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  $\Leftrightarrow$  satisfiable(f);
  decreases |literals(f)|;
 $\Leftrightarrow$ 
  if f =  $\emptyset$ [Clause] then
     $\top$ 
  else if  $\emptyset$ [Literal]  $\in$  f then
     $\perp$ 
  else
    choose l  $\in$  literals(f) in
      DPLL(substitute(f,l))  $\vee$  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  $\in$  literals(f) };
val m = 2(n+1)-1;
fun size(f:Formula): N[m] = 2(|vars(f)|+1)-1;

```

which ultimately give a measure for the complexity of the work that is still to be performed for every formula stored on the 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  $\Leftrightarrow$  satisfiable(f);
{
  var satisfiable: Bool :=  $\perp$ ;
  var stack: Array[n+1,Formula] := Array[n+1,Formula]( $\emptyset$ [Clause]);
  var number: N[n+1] := 0;
  stack[number] := f;
  number := number+1;
  while  $\neg$ satisfiable  $\wedge$  number > 0 do
    invariant 0  $\leq$  number  $\wedge$  number  $\leq$  n+1;

```



```

invariant number > 0 ∧ stack[number-1] ≠ 0[Clause] ∧
      ¬0[Literal] ∈ stack[number-1] ⇒ number < n+1;
invariant satisfiable(f) ⇔ satisfiable ∨
      ∃i:N[n+1] with i<number. satisfiable(stack[i]);
decreases if satisfiable then 0 else
      ∑k:N[n] with k<number. size(stack[k]);
{
  number := number-1;
  var g:Formula := stack[number];
  if g = 0[Clause] then
    satisfiable := ⊤;
  else if ¬0[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 263) 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).
WARNING: not all nondeterministic branches 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.

3.4 DPLL Algorithm with Subtypes

As the previous example has shown, checking operations on large domains of “raw” values from which the meaningful values have to be filtered by auxiliary preconditions can become quite cumbersome. In many cases, the use of “subtypes” may make our lives considerably easier.

For this, we start with the following declarations that introduce the same constants as in the previous example (the full specification is given in Appendix C.6):

```

val n: N;
val cn: N;
val fn: N;

```

Now we define the domain of literals as follows:

```

type LiteralBase = ℤ[-n,n];
type Literal = LiteralBase with value ≠ 0;

```

Here the type `LiteralBase` denotes the type of all “raw literals”; the type `LiteralBase` then is defined as a subtype of `Literal` that only includes the meaningful (non-zero) values. The clause `with value ≠ 0` describes the side condition that every value of type `Literal` must fulfill; the special name `value` denotes the value to which the condition is applied.

Correspondingly, we can introduce the other types as subtypes of raw types based on the same auxiliary predicates that we have defined in the previous example; additionally we immediately restrict the sizes of the types such that exhaustive checking becomes feasible:

```

type ClauseBase = Set[Literal];
pred clause(c:ClauseBase) ⇔ ∀l∈c. ¬(l∈c ∧ ¬l∈c);

```

```

type Clause = ClauseBase with |value| ≤ cn ∧ clause(value);

type FormulaBase = Set[Clause];
pred formula(f:FormulaBase) ⇔ ∀c∈f. clause(c);
type Formula = FormulaBase with |value| ≤ fn ∧ formula(value);

type Valuation = ClauseBase with clause(value);

```

When we now process the specification, we get the following output which shows the processing of the subtype definitions:

```

Using n=3.
Using cn=2.
Using fn=20.
Evaluating the domain of Literal...
Evaluating the domain of Clause...
Evaluating the domain of Formula...
Evaluating the domain of Valuation...
Computing the value of m...
Type checking and translation completed.

```

Now, in the following definitions all occurrences of the side conditions can be removed, e.g. rather than writing

```

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));

```

(as we did in the previous example), we can now write

```

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

```

Furthermore, we can drop from predicate DPLL and procedure DPLL2 the precondition clause `requires formula(f)` which is now subsumed by the definition of subtype `Formula`.

When now checking the algorithm for all inputs, we get the following output:

```

Executing DPLL2(Formula) with all 524288 inputs.
PARALLEL execution with 4 threads (output disabled).
2081 inputs (1519 checked, 0 inadmissible, 0 ignored, 562 open)...
3507 inputs (2842 checked, 0 inadmissible, 0 ignored, 665 open)...
4153 inputs (3974 checked, 0 inadmissible, 0 ignored, 179 open)...
5247 inputs (5082 checked, 0 inadmissible, 0 ignored, 165 open)...
6334 inputs (6123 checked, 0 inadmissible, 0 ignored, 211 open)...
7344 inputs (7151 checked, 0 inadmissible, 0 ignored, 193 open)...
...

```

Compared to the output from the previous example, we see that the domain of the check has been automatically restricted to the values of interest.

Figure 18 displays the verification conditions generated for the function DPLL respectively the procedure DPLL2 (for DPLL2, only some of the tasks in the folder “Verify implementation preconditions” are shown, in total there are more than 30 such tasks). Checking these conditions is possible under the following constraints:

- For DPLL, the check of a condition like “Is result correct?” takes for $n = 3$ with 4 threads 4–5 minutes, which requires quite some patience. However, choosing $n = 2$ reduces the checking time even with a single thread to less than half a second.
- For DPLL2, even a check with $n = 2$ is not feasible for most of the verification conditions, because these involve a universal quantification of the *stack* variable which has a huge domain of values. Only for $n = 1$ all checks succeed within less than half a second (with limited evidential value, of course).

Checking verification conditions in such small models is apparently of very limited value to justify their general validity. However, even with $n = 1$ errors/inadequacies in the annotations of the loop of the DPLL2 procedure could be detected, which demonstrates that even very small models may help to falsify sloppy correctness arguments.

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 [8, 35] 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, 10] 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 [22, 15] embeds a functional programming language in which algorithms can be defined and verified and converted to programs in OCaml, Haskell, SML, and Scala. In [21], 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.

Automated reasoners may also provide support for counterexample generation which demonstrates the invalidity of formulas; however this is necessarily unreliable, because every sufficiently rich logic (such as first order predicate logic) is undecidable. For instance, the counterexample generator Nitpick [5] for Isabelle supports higher order logic but may (due to the presence

<ul style="list-style-type: none"> ▼ Verify correctness of result <ul style="list-style-type: none"> ☀ Is result correct? ▼ Verify iteration and recursion <ul style="list-style-type: none"> ☀ Is function measure non-negative? ☀ Is function measure decreased? ☀ Is function measure decreased? ▼ Verify implementation preconditions <ul style="list-style-type: none"> ☀ Is choice possible? ☀ Does operation precondition hold? ☀ Does operation precondition hold? ☀ Does operation precondition hold? ☀ Does operation precondition hold? ☀ Does operation precondition hold? ☀ Does operation precondition hold? 	<ul style="list-style-type: none"> ▼ Verify correctness of result <ul style="list-style-type: none"> ☀ Is result correct? ▼ Verify iteration and recursion <ul style="list-style-type: none"> ☀ Does loop invariant initially hold? ☀ Does loop invariant initially hold? ☀ Does loop invariant initially hold? ☀ Is loop measure non-negative? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop invariant preserved? ☀ Is loop measure decreased? ☀ Is loop measure decreased? ☀ Is loop measure decreased? ▼ Verify implementation preconditions <ul style="list-style-type: none"> ☀ Is index value legal? ☀ Is update value legal? ☀ Is assigned value legal? ☀ Is index value legal? ☀ Is index value legal? ☀ Does operation precondition hold? ☀ Is index value legal? ☀ Does operation precondition hold? ☀ Does operation precondition hold? ☀ Is index value legal?
---	---

Figure 18: The Correctness of DPLL respectively DPLL2

of unbounded quantifiers) fail to find counterexamples; in an “unsound mode” quantifiers are artificially bounded, but then invalid counterexamples may be reported. While counterexample generators such as Nitpick are usually based on SAT/SMT-solving techniques, RISCAL’s implementation is actually closer to that of the test case generator Smallcheck [29]. This system generates for the parameters of a Haskell function in an exhaustive way argument values up to a certain bound; the results of the corresponding function applications may be checked against properties expressed in first-order logic (encoded as executable higher-order Haskell functions). However, unlike RISCAL, the value generation bound is specified by global constants rather than by the types of parameters and quantified variables such that separate mechanisms have to be used to restrict searches for counterexamples.

Also the abstract data type specification languages of the OBJ family [14, 13] 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 [34, 33] 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 [16, 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 satisfying given constraints. While Alloy can be used to formulate algorithms/programs, this can become quite challenging [27], because the language differs very much from conventional languages. Event-B [1, 12] 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 [27].

RISCAL has been more directly influenced by the temporal logic of actions (TLA) [18, 36] which has evolved into a specification language TLA+ for describing concurrent systems. It also supports 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 [19, 23] 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 [27]: 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 [6, 37], 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 [20, 11] 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) [9, 17] 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 [27] 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 one for model checking (with VDM as an alternative for recursive algorithms, which are not supported by PlusCal).

5 Conclusions and Future Work

Since Version 1, the RISCAL software has allowed 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. Version 2 of the software now also includes a verification condition generator for the specification language. The conditions are parameterized over the unspecified domain bounds; for concrete values of these bounds, the resulting model is finite and conditions are decidable by evaluation in that model. If such concrete instances are invalid, also the general condition is invalid and a proof need not be attempted. Thus we are also 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.

Further work will concentrate to extend the software along various lines. First, we plan to provide explanation capabilities that help (e.g., by visualizing the execution that leads to the counterexample) to quickly understand why a particular formula is invalid. Next, we envision to augment the current evaluation-based checking mechanism for formulas (via translation to an SMT-decidable logic) by SMT-solver based checking, which has the potential to be much more efficient (whether this is really the case, remains to be seen). Furthermore, we plan a web-based frontend of the software to enable its use without requiring a local installation. Ultimately, we aim to connect the RISCAL software to a computer-aided interactive proof assistant such as the RISC ProofNavigator [30, 26] in order to perform general verifications.

The main use of RISCAL is envisioned in educational scenarios [32]; for this we will develop formal models and supporting lecture material in areas such as discrete mathematics [7], fundamental algorithms [24], logic, and computer algebra.

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] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Springer LNCS*, pages 131–146, 2010. https://doi.org/10.1007/978-3-642-14052-5_11.
- [6] 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>.

- [7] Alexander Brunhuemer. Validating the Formalization of Theories and Algorithms of Discrete Mathematics by the Computer-Supported Checking of Finite Models. Bachelor Thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2017. http://www.risc.jku.at/publications/download/risc_5485/Validating_RISCAL_Formalization_Discrete_Math.pdf.
- [8] 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>.
- [9] 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.
- [10] The Coq Proof Assistant, January 2017. <https://coq.inria.fr>.
- [11] 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>.
- [12] Event-B and the Rodin Platform, November 2015. <http://www.event-b.org>.
- [13] Kokichi Futatsugi et al. CafeOBJ. Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan, January 2017. <https://cafeobj.org>.
- [14] 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>.
- [15] Isabelle, November 2016. <https://isabelle.in.tum.de>.
- [16] Daniel Jackson. *Software Abstractions — Logic, Language, and Analysis*. MIT Press, Cambridge, MA, USA, revised edition, November 2011. <https://mitpress.mit.edu/books/software-abstractions>.
- [17] The Java Modeling Language (JML) Home Page, February 2013. <http://www.jmlspecs.org>.
- [18] 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>.
- [19] 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.

- [20] 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.
- [21] Tobias Nipkow and Gerwin Klein. *Concrete Semantics — With Isabelle/HOL*. Springer, Heidelberg, Germany, 2014. <https://doi.org/10.1007/978-3-319-10542-0>.
- [22] 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>.
- [23] Overture Tool — Formal Modelling in VDM, January 2017. <http://overturetool.org>.
- [24] Lucas Payr. Formalization and Validation of Fundamental Sequence Algorithms by Computer-assisted Checking of Finite Models. Bachelor Thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2018. To appear.
- [25] The RISC ProgramExplorer, June 2016. <https://www.risc.jku.at/research/formal/software/ProgramExplorer>.
- [26] The RISC ProofNavigator, September 2011. <https://www.risc.jku.at/research/formal/software/ProofNavigator>.
- [27] 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.
- [28] 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.
- [29] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *First ACM SIGPLAN Symposium on Haskell*, Haskell ’08, pages 37–48, New York, NY, USA, 2008. ACM. <https://doi.org/10.1145/1411286.1411292>.
- [30] 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>.
- [31] 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>.

- [32] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models. In Pedro Quaresma and Walther Neuper, editors, *Post-Proceedings ThEdu'17, Theorem proving components for Educational software*, Gothenburg, Sweden, 6 Aug 2017, volume 267 of *EPTCS*, pages 120–139, 2018. <https://doi.org/10.4204/EPTCS.267.8>.
- [33] About SETL, January 2015. <http://setl.org/setl>.
- [34] 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>.
- [35] The Theorema System, January 2017. <https://www.risc.jku.at/research/theorema/software>.
- [36] The TLA Home Page, November 2016. <https://research.microsoft.com/en-us/um/people/lamport/tla/tla.html>.
- [37] 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  
and the Eclipse 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 on how to get the virtual machine. After installation and login as "guest" you have the commands

RISCAL & (uses GTK3 but without visualization of execution traces)
RISCAL-visual & (uses GTK2 only but supports trace visualization)

[This distinction is necessary, because the virtual machine runs (due to larger compatibility with different types of host computers) a 32 bit version of GNU/Linux and thus also a 32 bit version of RISCAL. However, support for 32 bit machines stopped with Java 8; furthermore 32 bit machines are not properly supported by the packages of Eclipse 4.8 or newer any more (Eclipse 4.8 and 4.9 have bugs with GTK3 on 32 bit machines, Eclipse 4.10 has officially dropped 32 bit support). The virtual machine thus uses Java 8 and for command "RISCAL" a patched variant of the current RISCAL package. On the long term, the virtual machine will be replaced by a 64 bit version.]

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
  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

=====

First make sure that you have installed the third party software described below (Java Development Kit is required, JavaFX and WebKitGTK are optional).

Then copy file etc/RISCAL to a directory in your PATH and adapt in this file the variable JAVA to point to the Java executable "java" of your JDK. Adapt LIB to point to the directory "lib" of the RISCAL distribution and make sure that the subdirectory \$LIB/swt64 contains the SWT library intended for your operating system and processor. Also configure the graphical user interface options SWT_GTK3, TRACE, JAVAFX and JAVAFX11 as described in this file.

You should then be able to execute

RISCAL &

Third Party Software That You Have to Install

=====

RISCAL assumes that the following third party software is installed on your computer (if it is not already provided by your GNU/Linux distribution, you have to download and install it manually).

Java Development Kit (Oracle JDK 11 recommended)

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Go to the "Downloads" section to download the JDK.

Oracle JDK 11 is recommended. Oracle JDK 8,9,10 and OpenJDK 8,9,10,11 are also supported (potentially with some limitations of the RISCAL graphical user interface and the visualization options, see below).

For the (optional) use of the visualization features of RISCAL, Oracle JDK 11 respectively OpenJDK 11 also need an installation of JavaFX 11.

JavaFX (OpenJFX 11 recommended)

<https://openjfx.io/>

<https://gluonhq.com/products/javafx/>

Press the "Download" button to download the library.

JavaFX is a framework for graphical user interfaces. It is only needed if RISCAL is started with the command line option "-visual" to enable the visualization of the execution traces of procedures and of the evaluation of formulas.

JavaFX 11 is not part of the Oracle JDK 11/OpenJDK 11 distribution; it has to be downloaded and installed separately (the older versions Oracle JDK 8,9,10 have JavaFX included, OpenJDK 9 and 10 do not support JavaFX).

There is a version OpenJFX 8 working with OpenJDK 8; this can be installed e.g. on a Debian 9 "stretch" GNU/Linux distribution as package "openjfx" by executing (as superuser)

```
apt-get install openjfx
```

On the downside, OpenJFX 8 does not work with GTK3 (only with the outdated GTK2). For using this version, you therefore have to set the environment variable "SWT_GTK3" to 0:

```
SWT_GTK3=0
```

However, be warned that you then lose all the improvements of GTK3, which lets the look and feel of RISCAL suffer.

WebKitGTK 1.2.0

<https://webkitgtk.org/>

Select the latest version of the "Releases" section.

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
```

If you use OpenJDK 8 with OpenJFX and SWT_GTK3=0, you must install the older version WebKitGTK 1.0; e.g. on Debian 9 "stretch" execute

```
apt-get install libwebkitgtk-1.0-0
```

Third Party Software That Comes with RISCAL

=====

RISCAL also uses the the following open source software developed by third parties. This software is already included in the distribution, but if you want or need, you can download the source code from the denoted locations and compile it on your own. Many thanks to the respective developers for making this great software freely available!

The Eclipse Standard Widget Toolkit 4.10

<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_64/GTK 2)" (if you use Linux with a 64bit x86 processor).

Eclipse GEF/Zest 5.0

<https://www.eclipse.org/gef>

This is a framework for visualizing graphs. It is only needed if RISCAL is started with the command line option "-visual" to enable the visualization of the execution traces of procedures and the evaluation of formulas.

Go to the "Download" link and download the latest "5.0.x" release build.

ANTLR 4.7

<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
```

Tango Icon Library 0.8.90
<http://tango.freedesktop.org>

This library provides the button/menu icons used by RISCAL.

Go to the section "Base Icon Library", subsection "Download", to download the library.

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 2.0 (June 18, 2018)
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
-o: print help including the 'Analysis' options and exit
-visual: enable visualization
-nogui: do not use graphical user interface
-p: print the parsed specification
-t: print the typed specification with symbols
-e <F>: execute first operation named <F>
-en <F> <N>: execute operation <N> (>= 1) named <F>
-s <T>: run in server mode with <T> threads
```

Options `-h` and `-o` print help messages and exit. Option `-visual` enables the visualization of execution traces and formula evaluations (see also Section [A.5](#)). Option `-nogui` switches off the graphical user interface (which is mainly useful with one of the following operations). Option `-p` prints the parsed specification, option `-t` also includes symbol information in the output. Option `-e` executes the operation with the given name; if there are multiple operations with the same name, the first one is selected. Option `-en` executes the N -th operation with the given name.

With option `-s` 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 then 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.

If RISCAL `-o` is executed also the following options with name `-opt-key` are printed:

`<options>` includes the following ‘Analysis’ options:

```
-opt-nd <O>: if <O> = 1, select 'Nondeterminism'
-opt-dv <V>: set 'Default Value' to <V>
-opt-ov <N> <V>: set in 'Other Values' <N> to <V>
-opt-si <O>: if <O> = 1, select 'Silent'
-opt-in <I>: set 'Inputs' to <I>
-opt-pm <P>: set 'Per Mille' to <P>
-opt-br <B>: set 'Branches' to <B>
-opt-ta <O>: if <O> = 1, select 'Trace'
-opt-te <O>: if <O> = 1, select 'Tree'
-opt-wi <W>: set 'Width' to <W>
-opt-he <H>: set 'Height' to <H>
-opt-mt <O>: if <O> = 1, select 'Multi-Threaded'
-opt-tr <T>: set 'Threads' to <T>
-opt-di <O>: if <O> = 1, select 'Distributed'
-opt-se <S>: set 'Servers' to <S>
```

These options allow to set all the options and configuration values provided by the “Analysis” panel of the interactive user interface.

The RISCAL program terminates with a negative exit code if an error has occurred and with a non-negative code, otherwise.

A.3 The User Interface

Main Window The user interface depicted in Figure [19](#) is divided into two parts. The left part mainly embeds an editor panel with the current specification. The right part is mainly filled

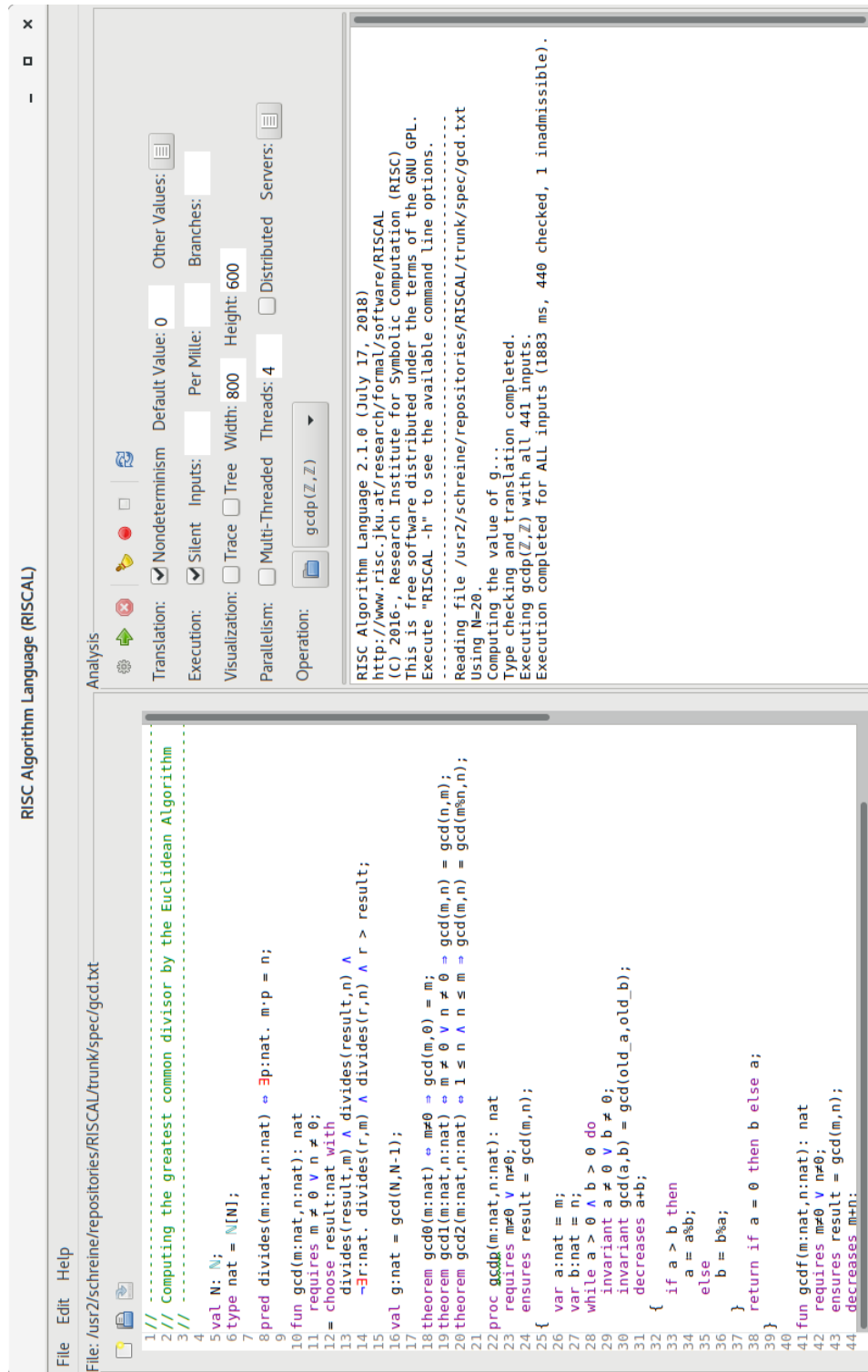




Figure 19: The RISCAL User Interface (Enlarged)

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.




Selecting the Operation To the right of the tag “Operation” there is the menu from which the operation can be selected that is subsequently executed by pressing the button  [Start Execution] (see below). Furthermore, if the button  [Show/Hide Tasks] is pressed, the window is extended to the right to display a panel (respectively hide it again) that contains additional tasks that may be performed to further validate the correctness of the currently selected operation (see Sections 2.6 and 2.7). Figure 20 displays this extended view.

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.

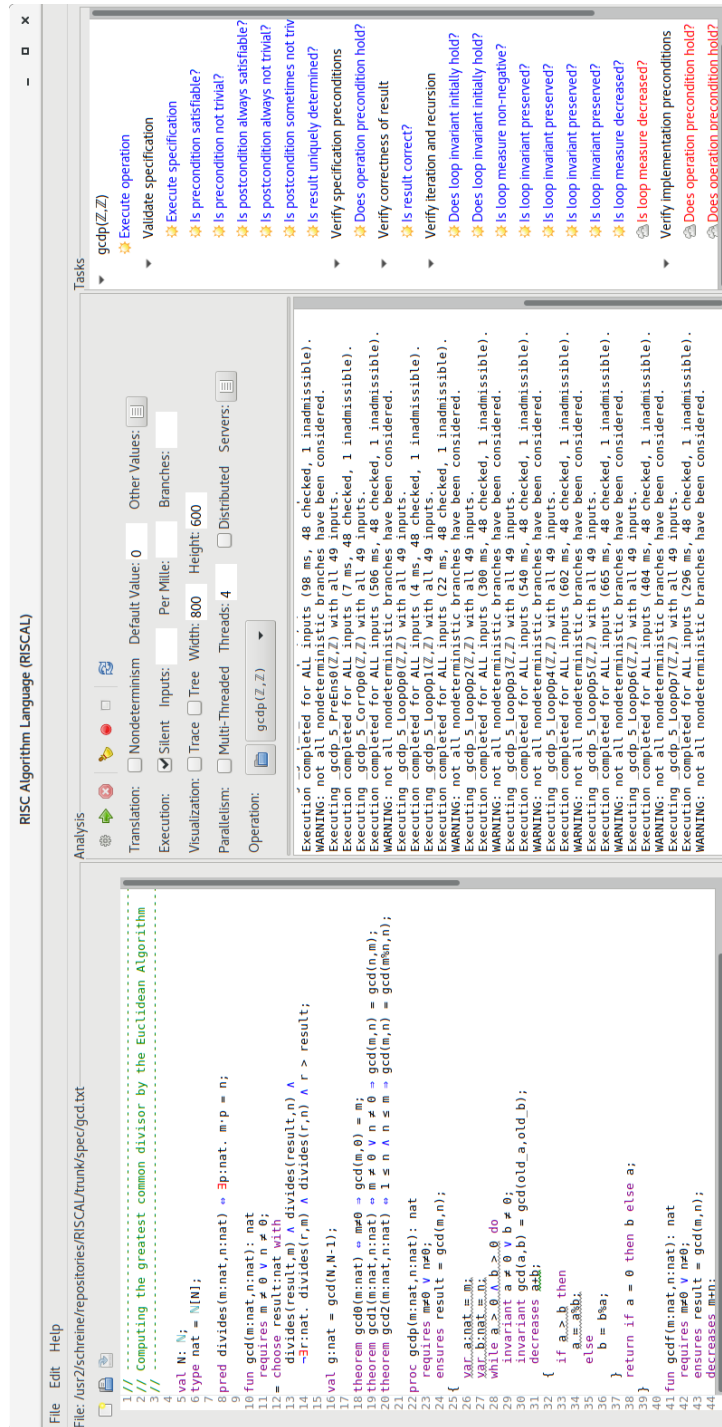






Figure 20: The GUI with the “Tasks” Panel

-  **[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 be 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.

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 21 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.

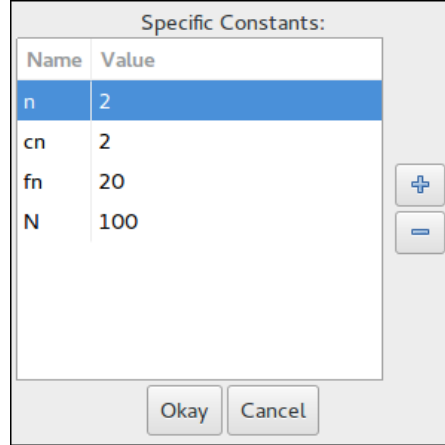



Figure 21: The “Other Values” Window

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 the Visualization The following options are only available, if RISCAL is started with the option `-visual` and visualization support is actually available (see Section A.5).

Trace If this option is selected and the options “Nondeterminism”, “Multi-Threaded” and “Distributed” are *not* selected, every subsequent execution of an operation creates a window in which the execution trace for the operation is visualized; when the window is closed, the visualization of the next operation execution is started. The process can be stopped (without the visualization of all executions) by pressing the button  (Stop Execution).

Tree If this option is selected and the options “Nondeterminism”, “Multi-Threaded” and “Distributed” are *not* selected, *after* the execution of an operation for *all* inputs has terminated, a window pops up that visualizes the evaluation tree for the formula(s) evaluated during this

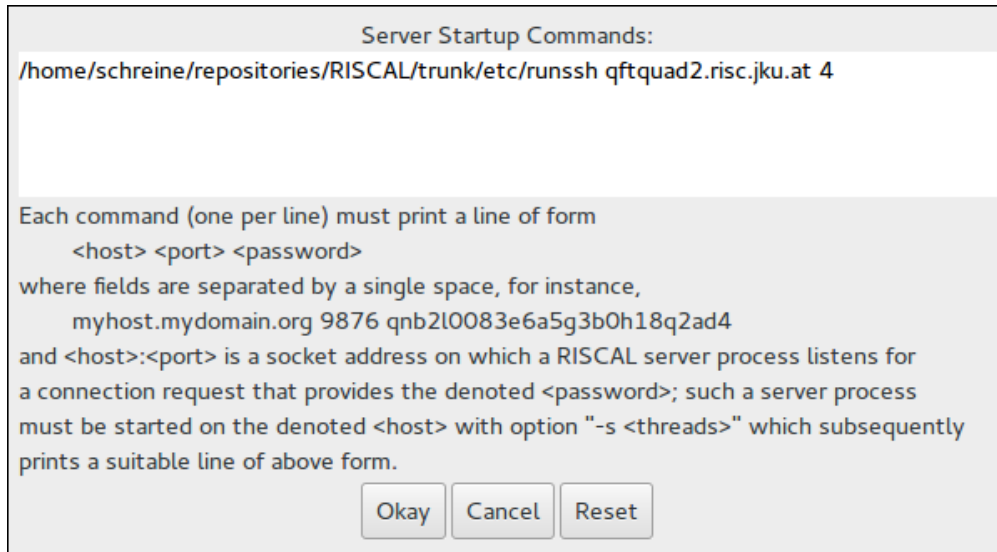


Figure 22: The “Servers” Window

execution. The layouting of this tree may take quite some time, thus small models should be visualized before attempting larger ones. Anyway, the software refuses to visualize trees whose node number exceeds a preconfigured maximum (currently 500).

Width/Height These text fields can be set to the desired size of the visualization area (which may be significantly larger than the actual screen size). In particular, if the algorithm for the layouting of evaluation trees fails to place tree nodes adequately (by putting them all into the left upper corner) larger dimensions should be chosen.

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 multi-threaded 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 22 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 “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 [22](#) on page [72](#) 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.

A.5 Visualization

Due to dependencies on external software (see below), the execution trace visualization of RISCAL is only enabled, if the command line option `-visual` is provided (this is hidden from the user: the local installation of the RISCAL script does or does not set this option).

The visualization has been implemented with the help of the Eclipse GEF/Zest framework for graph visualization¹. This software depends on the graphics framework JavaFX² which is part of the Oracle JDK 9 implementation of Java; the use of this version of Java is thus strongly recommended, if the visualization feature is to be used (otherwise also any version of JDK 8 is fine).

The Java open source implementation OpenJDK 9 does not yet support JavaFX; if for some reason Oracle JDK 9 cannot be used (e.g., Oracle JDK 9 is not available for 32 bit operating systems) but the visualization is nevertheless desired, the local installation must provide OpenJDK

¹<https://github.com/eclipse/gef/wiki/Zest>

²<https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>

8 and the corresponding OpenJFX 8 implementation of JavaFX. However, then the Eclipse SWT toolkit on which RISCAL is based must not use the current version GTK3 of the graphical toolkit GTK but the older version GTK2 (which gives RISCAL a somewhat outdated look and feel); this can be achieved by setting the environment variable `SWT_GTK3` to `0`.

The `README` file of the distribution (see Section [A.1](#)) contains more detailed information on the configuration of RISCAL for the use of the trace visualization extension. The RISCAL web site provides a 32 bit virtual machine with RISCAL preinstalled; in that machine the command `RISCAL-trace` invokes RISCAL with trace visualization enabled but only using GTK2; the command `RISCAL` uses GTK3 but has the visualization disabled.

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 23](#).

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* $\langle ident \rangle$ 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* $\langle decimal \rangle$ is a non-empty sequence of decimal digits, e.g. `123`.
- A *string literal* $\langle string \rangle$ 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}\""`.

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

Figure 23: ASCII Strings and their Equivalent Unicode Characters

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

$$\langle domain \rangle ::= \langle alternative1 \rangle \mid \dots \mid \langle alternativeN \rangle$$

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 \mathbb{Z} are literals that stand for themselves.
- $(\langle phrase1 \rangle \mid \dots \mid \langle phraseN \rangle)$ denotes one of the phrases $\langle phrase1 \rangle, \dots, \langle phraseN \rangle$.
- $(\langle phrase \rangle)?$ denotes 0 or 1 occurrence of $\langle phrase \rangle$.
- $(\langle phrase \rangle)^*$ denotes 0 or more occurrences of $\langle phrase \rangle$.
- $(\langle phrase \rangle)^+$ denotes 1 or more occurrences of $\langle phrase \rangle$.

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

B.2 Specifications and Declarations

A specification is a sequence of declarations:

$$\langle specification \rangle ::= (\langle declaration \rangle)^*$$

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

$$\langle declaration \rangle ::= \dots$$

of declarations.

B.2.1 Types

Grammar

```
type <ident> = <type> ( with <exp> )? ;
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`. If also a clause `with b` is given, then `b` must be a formula (an expression of type `Bool`) where the identifier `value` may appear as a free variable. In this case, `id` is a subtype of `T`, which contains every value `v` of `T` for which, if `value` is set to `v`, the evaluation of `b` yields “true”.
- A recursive type definition `rectype(n) id = c | f(T1, ..., Tn) | ...`; introduces a new type `id` with constant `id!c` and constructor `id!f`, a function with parameters of types `T1, ..., Tn` 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 itself appear as a parameter type of a constructor (thus the name “recursive type”). While thus arbitrarily deep nestings of constructor applications are possible, a variable/constant of the recursive type may only hold a value whose number of recursive constructor applications does not exceed the bound set by the constant expression $n \geq 0$. In particular, if $n = 0$, then only constant values (or non-recursive constructor applications) may be used; if $n = 1$, only applications of constructors to constants (or non-recursive constructor applications) are allowed. This constraint is checked at runtime; if it is violated, execution is aborted. See Section B.5.10 for a detailed description of how recursive constructors are defined and how the number of recursive constructor applications (determining the “height” of a recursive type value) is calculated.

A recursive type definition `rectype(n) id1 = ... and ... and idn = ...`; introduces n types `id1, ..., idn` that may mutually recursively depend on each other, i.e., one type may appear as a parameter type of a constructor of another type. The bound n applies to all types in common: not more than n nested recursive 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 = c1 | ... | cn`; is an abbreviation for the recursive type definition `rectype(0) id = c1 | ... | cn`;

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. The function `height(e)` described in Section B.5.10 allows to determine the actual “height” of the value of expression `e` of a recursive type. Examples of recursive type definitions are the following:

```

type T = ... ; value L:N ; value H:N ; value N:N ;
rectype(L) List = nil | cons(T,List) ;
rectype(H) Tree = empty | node(T,Tree,Tree) ;
rectype(H) NTree = empty | node(T,Array[N,NTree]) ;
enumtype Color = red | black | blue ;

```

Here type `List` denotes the type of lists whose length is at most L , type `Tree` denotes the type of binary trees whose height is at most H , while type `NTree` denotes the type of N -ary trees whose height is at most H . The enumeration type `Color` consists of the values `red`, `black`, and `blue`.

The definition of subtypes allows to check operations on smaller domains: let us assume that a type T with n values is constrained by a formula b to $m < n$ values of interest. If we then define for instance an operation with argument S of type $\text{Set}(T)$ and a precondition $\forall x \in S. b(x)$ that restricts the arguments to the only interesting sets (those with only interesting values), then the checker generates 2^n possible argument values from which the precondition filters the $2^m \ll 2^n$ interesting ones to which the operation is ultimately applied. If, however, we define a subtype T' of T with condition $b(\text{value})$ and give argument S type $\text{Set}(T')$, then the checker generates in the first place only 2^m values to which the operation is applied.

B.2.2 Values

Grammar

```

val <ident> : ( N | Nat ) ;
val <ident> ( : <type> )? = <exp> ;
pred <ident> (  $\Leftrightarrow$  |  $\leq$  ) <exp> ;
theorem <ident> (  $\Leftrightarrow$  |  $\leq$  ) <exp> ;
axiom <ident> (  $\Leftrightarrow$  |  $\leq$  ) <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 \Leftrightarrow can be alternatively written as \leq) 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 \Leftrightarrow can be alternatively written as \leq) 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.

- **axiom** $t \Leftrightarrow b$; is interpreted in the same way as a corresponding **theorem** declaration (however, see below for the pragmatic distinction between theorems and axioms).

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.

Axioms shall describe constraints on the unspecified (externally defined) constant values such that for those values that satisfy these constraints the specification is well-defined and the theorems hold; consequently only such values may be externally chosen for the constants. While the distinction of axioms and theorems is technically not utilized by the RISCAL checker, it allows external provers respectively satisfiability solvers to verify a RISCAL specification for infinitely many constant values: they may assume that all these values satisfy the axioms and from these assumptions prove the theorems.

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)? axiom <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> )* ; | modular ;
```

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

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

- **theorem** $t(p_1:T_1, \dots, p_n:T_n) \Leftrightarrow b$; (where the symbol \Leftrightarrow can be alternatively written as \Leftrightarrow) 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.
- **axiom** $t(p_1:T_1, \dots, p_n:T_n) \Leftrightarrow b$; is interpreted in the same way as the corresponding theorem declaration (but see Section B.2.2 for the pragmatic distinction).
- **proc** $p(p_1:T_1, \dots, p_n:T_n):T \{ c_1; \dots; c_n; \text{return } e; \}$ introduces a procedure p with n parameters p_1, \dots, p_n of types T_1, \dots, T_n , respectively; the value of the procedure is computed by executing the commands c_1, \dots, 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 $p(p_1:T_1, \dots, p_n:T_n):() \{ c_1; \dots; c_n; \}$ introduces a procedure p with result type $()$ (i.e., type **Unit**); this procedure executes commands c_1, \dots, 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, \dots, 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). The verification condition generator has some stronger requirements on termination measures (see the paragraph “Pragmatics” below).

The definition of a function f may be annotated by a clause `modular`, which affects the generation of verification conditions that involve some application $f(a)$ of the function. Without using `modular`, if f is *not* a procedure and *not* (directly or indirectly) recursively called by the function for which the verification condition is generated, the verification condition generator treats f as a “mathematical” function whose application $f(a)$ is plainly inserted in the verification condition. The effect on the verification condition is thus as if the definition of f would have been inserted literally; any change on the definition of f may thus affect the validity of the condition. However, if f is a procedure or recursively called, then an application $f(a)$ is essentially replaced by the expression `let $x=a$ in choose result: T with Q` where x is the parameter of f , T the result type of f , and Q its postcondition (`true`, if f has no postcondition). The verification thus considers all possible results of f allowed by its specification *independent* of its actual definition (which may thus change without effect on the validity of the verification condition, provided that the result still satisfies the postcondition). However, if f is annotated as `modular`, this kind of “modular reasoning” is also applied if f is not a procedure and not recursively called.

Pragmatics See Section B.2.2 for the pragmatic difference between theorems and axioms.

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 only appear at the end of a procedure definition; this simplifies the later development of a verification calculus.

In (mutually) recursive functions or procedures, the verification condition generator only considers the termination measure e_1, \dots, e_n provided by the *first* `decreases e_1, \dots, e_n` clause in each of the functions of the recursion cycle. Furthermore, all functions have to use the *same* number n of terms in their termination measures and *every* invocation of a function or procedure in the recursion cycle must decrease the measure. This requirement is actually stronger than ensured by the type checker respectively the model checker, because when checking the execution of recursive functions it is only required that every function decreases its own termination measures. If the stronger requirement is violated, the generated verification conditions are not true (i.e., invalid), and the termination of the functions respectively procedures cannot be verified.

The clause `modular` enables modular reasoning also for functions that look “mathematical” but are to be considered as program functions. The default for “mathematically-looking” functions is non-modular reasoning in order to avoid the necessity to annotate every such function with a

postcondition (which would often just repeat the definition of the function).

B.3 Commands

In this section, we are going to describe the domain

$$\langle \text{command} \rangle ::= \dots \mid \langle \text{exp} \rangle \mid \{ (\langle \text{command} \rangle)^* \} \mid ;$$

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

$$\begin{aligned} \text{val } \langle \text{id} \rangle (: \langle \text{type} \rangle)? (:= \mid \text{:=} \mid =) \langle \text{exp} \rangle ; \\ \text{var } \langle \text{id} \rangle : \langle \text{type} \rangle ((:= \mid \text{:=} \mid =) \langle \text{exp} \rangle)? ; \\ \langle \text{id} \rangle (\langle \text{sel} \rangle)^* (:= \mid \text{:=} \mid =) \langle \text{exp} \rangle ; \\ \langle \text{sel} \rangle ::= [\langle \text{exp} \rangle] \mid . \langle \text{decimal} \rangle \mid . \langle \text{id} \rangle \end{aligned}$$

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:

$$a := a \text{ with } [i] := (a[i] \text{ with } .id = e)$$

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 $\{ p_1 \rightarrow c_1; \dots; p_n \rightarrow c_n; \}$** attempts to “match” the value of e (which must be of some recursive type T) to the patterns p_1, \dots, p_n . Each pattern can be either the name id of a constant of type T or an application $id(p_1, \dots, p_n)$ 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 $p_i = id(p_1, \dots, p_n)$, the parameters p_1, \dots, 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  $\langle exp \rangle$  do (  $\langle loopspec \rangle$  ) *  $\langle command \rangle$ 
do (  $\langle loopspec \rangle$  ) *  $\langle command \rangle$  while  $\langle exp \rangle$  ;
for  $\langle command \rangle$  ;  $\langle exp \rangle$  ;  $\langle command \rangle$  do (  $\langle loopspec \rangle$  ) *  $\langle command \rangle$ 

for  $\langle qvar \rangle$  do (  $\langle loopspec \rangle$  ) *  $\langle command \rangle$ 
choose  $\langle qvar \rangle$  do (  $\langle loopspec \rangle$  ) *  $\langle command \rangle$ 

 $\langle loopspec \rangle ::= \text{invariant } \langle exp \rangle ; | \text{decreases } \langle exp \rangle ( , \langle exp \rangle )^* ;$ 

```

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). Likewise, in a loop **choose** q **do** c , the invariant may refer to the identifier `chooseSet` which denotes the set of all values chosen in the previous iterations of the loop.

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, \dots, 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). While a loop may be annotated with multiple measures that are also checked during execution of the loop, the verification condition generator only considers the first measure (see also the paragraph “Pragmatics” below).

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.

The verification condition generator only considers the termination measure e_1, \dots, e_n provided by the *first decreases* e_1, \dots, e_n clause in each loop for generating verification conditions; subsequent measures are silently ignored.

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, \dots, e_n` ; prints the values of e_1, \dots, e_n . A command like `print "...{i} ..."`, e_1, \dots, e_n ; prints these 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 $\text{Set}[T]$. If f has multiple parameters of types T_1, \dots, T_n , S must have type $\text{Set}[T_1 \times \dots \times T_n]$.

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
  | (  $\mathbb{Z}$  | Int ) [ <exp> , <exp> ]
  | (  $\mathbb{N}$  | 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 \perp (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, $\text{Int}[min, max]$) 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, $\text{Nat}[max]$) 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.

- $\text{Set}[T]$ 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.

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

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

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

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

- $\text{Array}[n, T]$ 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 $0, \dots, n-1$.

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

- $\text{Map}[K, E]$ 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.

$\text{Array}[n, T]$ is essentially a synonym for $\text{Map}[\mathbb{N}[n-1], T]$. 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

$$\langle exp \rangle ::= \dots \mid (\langle exp \rangle)$$

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 $(\langle exp \rangle)$ with parentheses may be used to indicate the intended parsing structure.

B.5.1 Constants and Applications

Grammar

$$\begin{aligned} &\langle ident \rangle \\ &\langle ident \rangle ((\langle exp \rangle (, \langle exp \rangle)^*)?) \end{aligned}$$

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 $id(e_1, \dots, e_n)$ denotes the value of the application of the “parameterized entity” denoted by id to the values of expressions e_1, \dots, 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

\top | true
 \perp | false
 $(\neg | \sim) \langle exp \rangle$
 $\langle exp \rangle (\wedge | /\wedge) \langle exp \rangle$
 $\langle exp \rangle (\vee | /\vee) \langle exp \rangle$
 $\langle exp \rangle (\Rightarrow | \Rightarrow) \langle exp \rangle$
 $\langle exp \rangle (\Leftrightarrow | \Leftrightarrow) \langle exp \rangle$
 $(\forall | \text{forall}) \langle qvar \rangle . \langle exp \rangle$
 $(\exists | \text{exists}) \langle qvar \rangle . \langle exp \rangle$

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 \top (respectively `true`) denotes the truth value “true”; likewise the literal \perp (respectively `false`) denotes “false”.
- The logical connectives that combine formulas to bigger formulas are represented as follows: the unary operator \neg (respectively \sim) denotes logical negation, while the binary operators \wedge (respectively $/\wedge$), \vee (respectively $/\vee$), \Rightarrow (respectively \Rightarrow), \Leftrightarrow (respectively \Leftrightarrow) denote logical conjunction, disjunction, implication, and equivalence, respectively. The operators are listed in the order of decreasing binding power, i.e. $a \vee \neg b \wedge c$ is parsed as $a \vee ((\neg b) \wedge c)$.
- The logical quantifiers that bind a variable in a formula are represented as follows: the quantifier \forall respectively `forall` denotes universal quantification, the quantifier \exists 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

$\langle exp \rangle = \langle exp \rangle$
 $\langle exp \rangle (\neq | \sim=) \langle exp \rangle$
 $\langle exp \rangle < \langle exp \rangle$
 $\langle exp \rangle (\leq | \leq) \langle exp \rangle$
 $\langle exp \rangle > \langle exp \rangle$
 $\langle exp \rangle (\geq | \geq) \langle exp \rangle$

Description Two values of the same arbitrary (also compound) type may be compared by application of the atomic predicates = denoting “equals” and (respectively \sim =) 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”, \leq (respectively \leq) denoting “less than or equal”, > denoting “greater than”, or \geq (respectively \geq) 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>
(  $\sum$  |  $\Sigma$  | sum ) <qvar> . <exp>
(  $\prod$  |  $\Pi$  | product ) <qvar> . <exp>
min <qvar> . <exp>
max <qvar> . <exp>
# <qvar>

```

Description The following kinds of expressions denote integers:

- A sequence $d_1 \dots d_n$ 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., $-a+b \cdot c$ is parsed as $(-a)+(b \cdot c)$.

- The expression $a + \dots + b$ denotes the sum $\sum_{i=a}^b i$; its value is 0, if $a > b$. Likewise, $a \cdot \dots \cdot b$ (alternatively, $a * \dots * b$) 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

```

(  $\emptyset$  | { } ) [  $\langle type \rangle$  ]
(  $\cap$  | Intersect )  $\langle exp \rangle$ 
(  $\cup$  | Union )  $\langle exp \rangle$ 
{  $\langle exp \rangle$  ( ,  $\langle exp \rangle$  ) * }
 $\langle exp \rangle \dots \langle exp \rangle$ 
|  $\langle exp \rangle$  |
 $\langle exp \rangle$  (  $\cap$  | intersect )  $\langle exp \rangle$ 
 $\langle exp \rangle$  (  $\cup$  | union )  $\langle exp \rangle$ 
 $\langle exp \rangle \setminus \langle exp \rangle$ 
{  $\langle exp \rangle$  |  $\langle qvar \rangle$  }
 $\langle exp \rangle$  ( (  $\times$  | times )  $\langle exp \rangle$  ) +
Set (  $\langle exp \rangle$  )
Set (  $\langle exp \rangle$  ,  $\langle exp \rangle$  )
Set (  $\langle exp \rangle$  ,  $\langle exp \rangle$  ,  $\langle exp \rangle$  )
 $\langle exp \rangle$  (  $\in$  | isin )  $\langle exp \rangle$ 
 $\langle exp \rangle$  (  $\subseteq$  | subseteq )  $\langle exp \rangle$ 

```

Description We have the following operations on sets:

- The literal $\emptyset[T]$ (alternatively $\{ \}[T]$) denotes the empty set of type $\text{Set}[T]$.
- $\{e_1, \dots, e_n\}$ denotes the set that consists of the values e_1, \dots, e_n (which must have all the same type).
- $a \dots b$ 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 \cap (alternatively **intersect**) denotes the intersection of two sets of the same type, the operator \cup (alternatively **union**) denotes their union, the operator \setminus 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 $S_1 \cup S_2 \cap S_3$ is parsed as $S_1 \cup (S_2 \cap S_3)$.

- $\cap S$ (alternatively **Intersect** S) denotes the intersection of all sets contained in set S while $\cup S$ (alternatively **Union** S) denotes their union.
- The set builder $\{ e \mid q \}$ denotes the set of all values of e that result from the values of the quantified variables in q .
- The Cartesian product $S_1 \times \dots \times S_n$ (alternatively, S_1 times \dots 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 $(S_1 \times \dots \times S_n)$.
- The power set $\text{Set}(S)$ denotes the set of all subsets of set S . $\text{Set}(S, n)$ denotes the set of all subsets of S whose cardinality (number of elements) is n . $\text{Set}(S, a, b)$ denotes the set of all subsets of S whose cardinality n satisfies $a \leq n \leq b$.
- The atomic formula $e \in S$ (alternatively $e \text{ is in } S$) 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 $S_1 \text{ subseteq } S_2$) 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 $\text{Set}(S, n)$ respectively $\text{Set}(S, a, b)$ are more memory-efficient than the computation of the same sets by the expressions

$$\{s \mid s \in \text{Set}(S) \text{ with } |s| = n\}$$

$$\{s \mid s \in \text{Set}(S) \text{ with } a \leq |s| \wedge |s| \leq b\}$$

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

B.5.6 Tuples

Grammar

$$(\langle \mid \langle \langle \rangle \langle exp \rangle (' , ' \langle exp \rangle)^* (\rangle \mid \rangle \rangle)$$

$$\langle exp \rangle . \langle decimal \rangle$$

$$\langle exp \rangle \text{ with } . \langle decimal \rangle = \langle exp \rangle$$

Description We have the following operations on tuples:

- $\langle e_1, \dots, e_n \rangle$ (alternatively, $\langle \langle e_1, \dots, e_n \rangle \rangle$) denotes a tuple of type $\text{Tuple}[T_1, \dots, T_n]$ whose components are the values e_1, \dots, e_n of types T_1, \dots, 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 $t.1, \dots, t.n$.
- $t \text{ with } .d = e$ 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

$(\langle | << \rangle \langle \text{ident} \rangle : \langle \text{exp} \rangle (', \langle \text{ident} \rangle : \langle \text{exp} \rangle)^* () | >>)$
 $\langle \text{exp} \rangle . \langle \text{ident} \rangle$
 $\langle \text{exp} \rangle \text{ with } . \langle \text{ident} \rangle = \langle \text{exp} \rangle$

Description We have the following operations on records:

- $\langle id_1 : e_1, \dots, id_n : e_n \rangle$ (alternatively, $\langle \langle id_1 : e_1, \dots, id_n : e_n \rangle \rangle$) denotes a record of type $\text{Record}[id_1 : T_1, \dots, id_n : T_n]$ whose components are values e_1, \dots, e_n of types T_1, \dots, T_n , respectively.
- $r.id$ denotes the value that record r holds in the component denoted by identifier id .
- $r \text{ with } .id = e$ 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

$\text{Array} [\langle \text{exp} \rangle , \langle \text{type} \rangle] (\text{exp})$
 $\langle \text{exp} \rangle [\langle \text{exp} \rangle]$
 $\langle \text{exp} \rangle \text{ with } [\langle \text{exp} \rangle] = \langle \text{exp} \rangle$

Description We have the following operations on arrays:

- $\text{Array}[n, T](e)$ denotes an array of type $\text{Array}[n, T]$ (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 $a[0], \dots, a[n-1]$. The value at any other index is undefined, i.e., the computation of such a value aborts.
- $a \text{ with } [i] = e$ 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

$\text{Map} [\langle \text{type} \rangle , \langle \text{type} \rangle] (\text{exp})$
 $\langle \text{exp} \rangle [\langle \text{exp} \rangle]$
 $\langle \text{exp} \rangle \text{ with } [\langle \text{exp} \rangle] = \langle \text{exp} \rangle$

Description We have the following operations on maps:

- $\text{Map}[K, E](e)$ denotes a map of type $\text{Map}[K, E]$ (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

$$\begin{aligned} \langle \text{ident} \rangle & ! \langle \text{ident} \rangle \\ \langle \text{ident} \rangle & ! \langle \text{ident} \rangle (\langle \text{exp} \rangle (, \langle \text{exp} \rangle)^*) \\ \text{height} & (\langle \text{exp} \rangle) \end{aligned}$$

Description A recursive value is a value whose type T has been introduced by a definition $\text{rectype}(n) T = \dots$ 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.
- $T!id(e_1, \dots, e_n)$ 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, \dots, e_n .
- $\text{height}(e)$ denotes the “height” of the “expression tree” of the value e of some recursive type T . This height is the maximum number of applications of “recursive constructors” in any path from the root of the tree to some of its leaves; a constructor of T is recursive if T (respectively some other type defined in the same rectype declaration as T) occurs as the type of some parameter of the constructor, respectively (if the parameter type is constructed from a builtin type constructor) as a part of this type. In detail, the height of e is defined as follows:
 - a constant $T!id$ has height 0; likewise, a non-recursive constructor application $T!id(e_1, \dots, e_n)$ has height 0;
 - a recursive constructor application $T!id(e_1, \dots, e_n)$ has height $1 + m$ where m is the maximum height of all e_i ;

- an atomic value (Boolean, natural number, or integer) has height 0; a container value (array, map, set, tuple, record) has as its height is the maximum height of its elements.

If e is to be the value of a variable or constant of type T , this height must range from 0 to the bound n given in the `rectype` declaration of T .

Pragmatics Take the declarations

```
type T = ... ; value L:N ; value H:N ; value N:N ;
rectype(L) List = nil | cons(T,List) ;
rectype(H) Tree = empty | node(T,Tree,Tree) ;
rectype(H) NTree = empty | node(T,Array[N,NTree]) ;
```

Then the height of some constant l of recursive type `List` denotes the length of list l (from 0 to L inclusively) by counting the number of applications of recursive constructor `cons` in the construction of l . Likewise, the height of some constant t of recursive type `Tree` denotes the usual notion of the height of binary tree t (from 0 to H inclusively) by counting the maximum number of applications of recursive constructor `node` in any path of t . Analogously, the height of some constant u of recursive type `NTree` denotes the usual notion of the height of N -ary tree u .

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 { $p_1 \rightarrow e_1$; ... ; $p_n \rightarrow e_n$; }` attempts to “match” the value of e (which must be of some recursive type T) to the patterns p_1, \dots, p_n . Each pattern can be either the name id of a constant of type T or an application $id(p_1, \dots, p_n)$ 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 $p_i = id(p_1, \dots, p_n)$, the parameters p_1, \dots, 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>
letpar <ident> = <exp> ( , <ident> = <exp> )* in <exp>
```

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

In contrast, the binder expression `letpar $id_1=e_1, \dots, id_n=e_n$ in e` binds *simultaneously* each constant id_i to the value of e_i (no binding can refer to the previously introduced ones) and then returns the value of e when evaluated in this environment.

Pragmatics The `letpar` binder allows to avoid a syntactic replacement $E[E_1/x_1, \dots, E_n/x_n]$ of free occurrences of variables x_1, \dots, x_n in expression E by the expressions E_1, \dots, E_n ; rather we may use the expression `letpar $x_1 = E_1, \dots, x_n = E_n$ in E` instead. This feature is heavily used in the automatically generated verification conditions.

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>
print ( <string> , )? <exp> ( , <exp> )* in <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 print expression `print e_1, \dots, e_n in e` prints the values of e_1, \dots, e_n and returns the value of e as a result. An expression like `print "... {i} ...", e_1, \dots, e_n in e` prints these 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 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 $\text{Set}[T]$. If f has multiple parameters of types T_1, \dots, T_n , S must have type $\text{Set}[T_1 \times \dots \times T_n]$.

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

$$\begin{aligned} \langle qvar \rangle &::= \langle qvcore \rangle (, \langle qvcore \rangle)^* ? \\ \langle qvcore \rangle &::= \langle ident \rangle : \langle type \rangle (\text{with } \langle exp \rangle) \\ &\quad | \langle ident \rangle (\in | \text{in } \langle exp \rangle (\text{with } \langle exp \rangle) \end{aligned}$$

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 $x \in S$ 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 $\text{Set}[T]$, 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 $x_1:T_1$ with $b_1, \dots, x_n:T_n$ with b_n introduces n quantified variables x_1, \dots, x_n of types T_1, \dots, T_n . If the optional formulas b_1, \dots, b_n are given (where each b_i may refer to variables x_1, \dots, x_i), the formulas yield for the value of these variables “true”. Analogously, each clause $x_i:T_i$ can be replaced by a clause $x_i \in S_i$ 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

$$\{ x/y \mid x \in S, y:T \text{ with } y \neq \mathbf{0} \wedge x > y \}$$

in the formula

$$\forall x \in S, y: T \text{ with } y \neq 0 \wedge x > y. x/y \geq 1$$

or in the following choice statement:

$$\text{choose } x \in S, y: T \text{ with } y \neq 0 \wedge x > y;$$

If there are multiple quantified variables, multiple predicates may restrict the space of the enumerated variable values and thus speed up the evaluation of phrases. For instance, in

$$\forall x \in S \text{ with } x \% 2 = 0, y: T \text{ with } y \neq 0 \wedge x > y. x/y \geq 1$$

only for every even value of x all values of y are enumerated, while in

$$\forall x \in S, y: T \text{ with } x \% 2 = 0 \wedge y \neq 0 \wedge x > y. x/y \geq 1$$

for all possible values of x this enumeration takes place.

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 ( 'with' exp )? 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
| 'axiom' ident ( '⇔' | '<=>' ) exp EOS #AxiomDefinition
| 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
| multiple 'axiom' ident '(' ( param ( ',' param)* )? ')'
  ( funspec )* ( '⇔' | '<=>' ) exp EOS #AxiomParamDefinition
;

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

// 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
| ( 'T' | 'true' )                             #TrueExp
| ( 'F' | 'false' )                             #FalseExp
| decimal                                       #IntLiteralExp
| ( '0' | '{' '}' ) '[' 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
| 'height' '(' exp ')'                        #HeightExp

// 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' | 'Σ' | 'sum' ) qvar '.' exp     #SumExp
| ( 'Π' | '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
| 'letpar' binder ( ',' binder )* 'in' exp #LetParExp
| 'choose' qvar 'in' exp              #ChooseInExp
| 'choose' qvar 'in' exp 'else' exp   #ChooseInElseExp

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

// print expression
| 'print' ( STRING ',' )? exp ( ',' exp)* 'in' exp #PrintInExp
| '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 )* #QuantifiedVariable
;

```

```

qvcore :
  ident ':' type ( 'with' exp )?      #IdentifierTypeQuantifiedVar
| ident ( '∈' | 'in' ) exp ( 'with' 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 a ≠ 0 ∨ b ≠ 0;
    invariant gcd(a,b) = gcd(old_a,old_b);
    decreases a+b;
  end while;
  result := a;
}

```

```

{
  if a > b then
    a := a%b;
  else
    b := b%a;
}
return if a = 0 then b else a;
}

fun gcdf(m:nat,n:nat): nat
  requires m≠0 ∨ n≠0;
  ensures result = gcd(m,n);
  decreases m+n;
= if m = 0 then n
  else if n = 0 then m
  else if m > n then gcdf(m%n, n)
  else gcdf(m, n%m);

proc gcdr(m:nat,n:nat): nat
  requires m≠0 ∨ n≠0;
  ensures result = gcd(m,n);
  decreases m+n;
{
  var result:nat = 0;
  if m = 0 then
    result := n;
  else if n = 0 then
    result := m;
  else if m > n then
    result := gcdr(m%n, n);
  else
    result := gcdr(m, n%m);
  return result;
}

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

```

C.2 Bubble Sort

```

// -----
// Sorting arrays by the Bubble Sort Algorithm
// -----

val N:N; val M:N;

type index = ℤ[-N,N];
type elem = ℤ[-M,M];
type array = Array[N, elem];

```

```

proc cswap(a:array, i:index, j:index): array
{
  var b:array = a;
  if b[i] > b[j] then
  {
    var x:elem := b[i];
    b[i] := b[j];
    b[j] := x;
  }
  return b;
}

proc bubbleSort(a:array): array
{
  var b:array = a;
  for var i:index := 0; i < N-1; i := i+1 do
  {
    for var j:index := 0; j < N-i-1; j := j+1 do
      b := cswap(b,j,j+1);
    }
  }
  return b;
}

```

C.3 Linear and Binary Search

```

// -----
// Linear and binary search in arrays
// -----

val N:N;
val M:N;

type int =  $\mathbb{Z}[-N,N]$ ;
type elem =  $\mathbb{N}[M]$ ;
type array = Array[N,elem];

proc search(a:array, x:elem): int
  ensures result = -1  $\Rightarrow \forall k:\text{int with } 0 \leq k \wedge k < N. a[k] \neq x;$ 
  ensures result  $\neq -1 \Rightarrow 0 \leq \text{result} \wedge \text{result} < N \wedge$ 
    a[result] = x  $\wedge \forall k:\text{int with } 0 \leq k \wedge k < \text{result}. a[k] \neq x;$ 
{
  var i:int = 0;
  var r:int = -1;
  while i < N  $\wedge$  r = -1 do
    invariant  $0 \leq i \wedge i \leq N;$ 
    invariant  $\forall j:\text{int}. 0 \leq j \wedge j < i \Rightarrow a[j] \neq x;$ 
    invariant  $r = -1 \vee (r = i \wedge a[r] = x);$ 
    decreases if r = -1 then N-i else 0;
  {
    if a[i] = x
      then r := i;
      else i := i+1;
  }
}

```

```

    }
    return r;
}

proc bsearchp(a:array, x:elem, from: int, to: int): int
  requires 0 ≤ from ∧ from-1 ≤ to ∧ to < N;
  requires ∀k:int with from ≤ k ∧ k ≤ to-1. a[k] ≤ a[k+1];
  ensures result = -1 ⇒ ∀k:int with from ≤ k ∧ k ≤ to. a[k] ≠ x;
  ensures result ≠ -1 ⇒ from ≤ result ∧ result ≤ to ∧ a[result] = x;
  decreases to-from+1;
{
  var result:int;
  if from > to then
    result := -1;
  else
    {
      val m:int = (from+to)/2;
      if a[m] = x then
        result := m;
      else if a[m] < x then
        result := bsearchp(a, x, m+1, to);
      else
        result := bsearchp(a, x, from, m-1);
    }
  return result;
}

fun bsearch(a:array, x:elem, from: int, to: int): int
  requires 0 ≤ from ∧ from-1 ≤ to ∧ to < N;
  requires ∀k:int with from ≤ k ∧ k ≤ to-1. a[k] ≤ a[k+1];
  ensures result = -1 ⇒ ∀k:int with from ≤ k ∧ k ≤ to. a[k] ≠ x;
  ensures result ≠ -1 ⇒ from ≤ result ∧ result ≤ to ∧ a[result] = x;
  decreases to-from+1;
= if from > to then
  -1
else
  let m = (from+to)/2 in
  if a[m] = x then m else
  if a[m] < x then bsearch(a, x, m+1, to)
  else bsearch(a, x, from, m-1);

fun bsearch(a:array, x:elem): int
  requires ∀k:int with 0 ≤ k ∧ k < N-1. a[k] ≤ a[k+1];
  ensures result = -1 ⇒ ∀k:int with 0 ≤ k ∧ k < N. a[k] ≠ x;
  ensures result ≠ -1 ⇒ 0 ≤ result ∧ result < N ∧ a[result] = x;
= bsearch(a, x, 0, N-1);

```

C.4 Insertion Sort

```

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

```

```

val N:N;
val M:N;

type elem = N[M];
type array = Array[N,elem];
type index = N[N-1];

pred sorted(a:array, n:N[N]) ⇔
  ∀i:index. i < n-1 ⇒ a[i] ≤ a[i+1];

pred permuted(a:array, b:array) ⇔
  ∃p:Array[N,index].
    (∀i:index,j:index with i < j ∧ j < N. p[i] ≠ p[j]) ∧
    (∀i:index with i < N. a[i] = b[p[i]]);

pred equals(a:array, b:array, from:N[N], to:Z[-1,N-1]) ⇔
  ∀k:index with from ≤ k ∧ k ≤ to. a[k] = b[k];

proc sort(a:array): array
  ensures sorted(result, N);
  ensures permuted(a, result);
{
  var b:array = a;
  for var i:N[N]:=1; i<N; i:=i+1 do
    invariant 1 ≤ i ∧ i ≤ N;
    invariant sorted(b, i);
    invariant permuted(a, b);
    invariant equals(b, old_b, i, N-1);
    decreases N-i;
  {
    var x:elem := b[i];
    var j:Z[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do
      invariant i = old_i;
      invariant x = old_b[i];
      invariant -1 ≤ j ∧ j ≤ i-1;
      invariant equals(b, old_b, i+1, N-1);
      invariant equals(b, old_b, 0, j+1);
      invariant ∀k:index with j+1 < k ∧ k ≤ i. b[k] = old_b[k-1];
      invariant ∀k:index with j+1 ≤ k ∧ k < i. b[k] > x;
      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.5 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 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 number > 0 ∧ stack[number-1] ≠ ∅[Clause] ∧
      ¬∅[Literal] ∈ stack[number-1] ⇒ number < n+1;
    invariant satisfiable(f) ⇔ satisfiable ∨
      ∃i:N[n+1] with i<number. satisfiable(stack[i]);
    decreases if satisfiable then 0 else
      ∑k:N[n] with k<number. size(stack[k]);
  {
    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,-1);
        number := number+1;
        stack[number] := substitute(g,1);
        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;
  }
}

```

C.6 DPLL Algorithm with Subtypes

```

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

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

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

// the raw types and the variously constrained subtypes

type LiteralBase = Z[-n,n];
type Literal = LiteralBase with value ≠ 0;

type ClauseBase = Set[Literal];
pred clause(c:ClauseBase) ⇔ ∀l∈c. ¬(l∈c ∧ ¬l∈c);
type Clause = ClauseBase with |value| ≤ cn ∧ clause(value);

type FormulaBase = Set[Clause];
pred formula(f:FormulaBase) ⇔ ∀c∈f. clause(c);
type Formula = FormulaBase with |value| ≤ fn ∧ formula(value);

type Valuation = ClauseBase with clause(value);

// 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. satisfies(v,f);
pred valid(f:Formula) ⇔ ∀v:Valuation. satisfies(v,f);

// the negation of a formula
fun not(f: Formula):Formula =
  { c | c:Clause with ∀d∈f. ∃l∈d. ¬l∈c };
theorem notFormula(f:Formula) ⇔ formula(not(f));
theorem notValid(f:Formula) ⇔ 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)
  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 iterative DPLL algorithm (without optimizations)
proc DPLL2(f:Formula): Bool
  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 number > 0 ∧ stack[number-1] ≠ ∅[Clause] ∧
      ¬∅[Literal] ∈ stack[number-1] ⇒ number < n+1;
    invariant satisfiable(f) ⇔ satisfiable ∨
      ∃i:N[n+1] with i<number. satisfiable(stack[i]);
    decreases if satisfiable then 0 else
      ∑k:N[n] with k<number. size(stack[k]);
  {
    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;
}
```