

The RISC ProgramExplorer Tutorial and Manual*

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<http://www.risc.jku.at>

October 3, 2011

Abstract

This document describes the use of the RISC ProgramExplorer, an interactive program reasoning environment which has been developed at the Research Institute for Symbolic Computation (RISC) and which integrates the previously developed RISC ProofNavigator as an interactive proving assistant. The environment allows to formally specify, analyze, and verify programs written in a subset of Java. For this purpose, it translates annotated programs into a semantic model which describes programs as state relations and is open for human investigation; from this model the software generates verification conditions which can be semi-automatically proved. Within the environment the user may elaborate mathematical theories as the basis of program specifications; an advanced graphical user interface links theories, programs, semantic models, and verification tasks and allows to easily navigate between the different views. The software runs on computers with x86-compatible processors under the GNU/Linux operating system; it is freely available under the terms of the GNU GPL.

*The hypertext version of this document can be found at <http://www.risc.jku.at/research/formal/software/ProgramExplorer/manual>.

Contents

1. Introduction	3
2. User Interface	5
3. Programs, Theories, and Specifications	13
3.1. Computing Factorial Numbers	13
3.2. Searching for Records	22
3.3. Failed Tasks and Interactive Proofs	28
4. Semantics and Verification	31
4.1. Computing Factorial Numbers	32
4.2. Recursive Computation of Factorials	46
4.3. Searching in Arrays	51
4.4. Program States and Control Flow Interruptions	56
4.5. Objects and Method Side Effects	60
A. Programs as State Relations	66
B. Programming Language	70
C. Specification Language	73
C.1. Logic Language	73
C.1.1. Declarations	73
C.1.2. Types	74
C.1.3. Mapping Program Types to Logical Types	74
C.1.4. Program Variables	77
C.1.5. Program States	78
C.1.6. State Functions	78
C.2. Theory Definitions	80
C.3. Class Specifications	81
C.4. Class Invariants	82
C.5. Method Specifications	82
C.6. Loop Specifications	84
C.7. Statement Specifications	85
D. New RISC ProofNavigator	87

E. Software Invocation	88
F. Software Installation	90
F.1. README	90
F.2. INSTALL	93
G. Task Directories	97
H. Grammars	98
H.1. Programming Language	98
H.2. Specification Language	103

1. Introduction

This document describes the use of the RISC ProgramExplorer, an interactive program reasoning environment which has been developed at the Research Institute for Symbolic Computation (RISC) and which integrates the previously developed RISC ProofNavigator [8, 5] as an interactive proving assistant. The environment allows to formally specify, analyze, and verify programs written in a subset of Java based on a calculus developed in [6, 7, 9]. In more detail,

- the RISC ProgramExplorer allows the user to elaborate mathematical theories and to formally specify a program on the basis of these theories;
- it translates the specified program into a semantic model which describes programs commands as state relations and which is open for human investigation;
- it generates from the semantic model the verification conditions which can be semi-automatically proved with the help of the RISC ProofNavigator.

The software provides an advanced graphical user interface that links theories, programs, semantic models, and verification tasks and allows the user to conveniently navigate between the different views.

The system is freely available under the GNU Public License at the URL

<http://www.risc.jku.at/research/formal/software/ProgramExplorer>

The software has been reasonably well tested but is certainly not free of bugs; the author is glad to receive error reports at

Wolfgang.Schreiner@risc.jku.at

The remainder of the document is split in two parts:

- **Chapters 2–4** essentially represent a tutorial for the RISC ProgramExplorer based on examples contained in the software distribution; for learning to use the system, we recommend to study this material in sequence.
- **Appendices A–H** essentially represent a reference manual with an explanation of the software’s programming and specification language; this material can be studied on demand.

This document does not explain in detail how to interactively prove the generated verification conditions; this is described in the manual of the RISC ProofNavigator [5].

The RISC ProgramExplorer uses the following third party software; detailed references can be found in the README file of the distribution listed on page 90:

- CVC Lite
- RIACA OpenMath Library
- General Purpose Hash Function Algorithms Library
- ANTLR
- Eclipse Standard Widget Toolkit
- Mozilla Firefox
- GIMP Toolkit GTK+
- Sun JDK
- Tango Icon Library

Many thanks to the respective authors for their great work.

2. User Interface

In the following we explain the main points of interaction with the user interface of the RISC ProgramExplorer. We assume that the system is appropriately installed (see Appendix F), that the current working directory is the subdirectory `examples` of the installation directory with write permission enabled (respectively that the current working directory is a writable copy of that directory), and that the task directory has been restored from file `PETASKS.tar.gz` (see the README file in the directory). After typing on the command line

```
ProgramExplorer &
```

first a splash screen with a copyright message appears. After a few seconds, a window pops up that displays the actual startup screen shown in Figure 2.1.

This window initially displays the “Analysis” view of the RISC ProgramExplorer, which is one of the three main views:

Analysis This view is mainly used to display/edit the source code of a program and theory file, to select symbols for the investigation of their semantics and to start the execution of verification tasks.

Semantics This view is used to display the semantics of a program method; it is shown, when in the menu associated to a method symbol (right-click the symbol) the entry “Show Semantics” is selected.

Verification This view is used to show a proof resulting from a task; it is displayed, when in the menu associated to a task (right-click the task), one selects the entry “Execute Task” (if the proof is not yet completed) respectively “Show Proof” (if the proof is completed).

One may switch to another view by clicking on the corresponding tab on the top-right of the window (provided the view is currently applicable).

In the following, we only describe the “Analysis” view; the other views are explained in Chapter 4. As already stated, the main purpose of this view is to edit a program or theory file; when saving the file, it is automatically type-checked and semantically processed. As a consequence, in the tab “Symbols”, the resulting semantic symbols (e.g. program methods) are displayed; in the tab “All Tasks”, the generated (e.g. verification) tasks are shown. Source code, symbols, and tasks are interlinked; by double-clicking a symbol/task, the corresponding line in the source code is highlighted; by double-clicking on the defining occurrence of a symbol name in the source code, the corresponding symbol is high lighted; by double-clicking any other occurrence is definition is printed.

The view has three menus at the top:

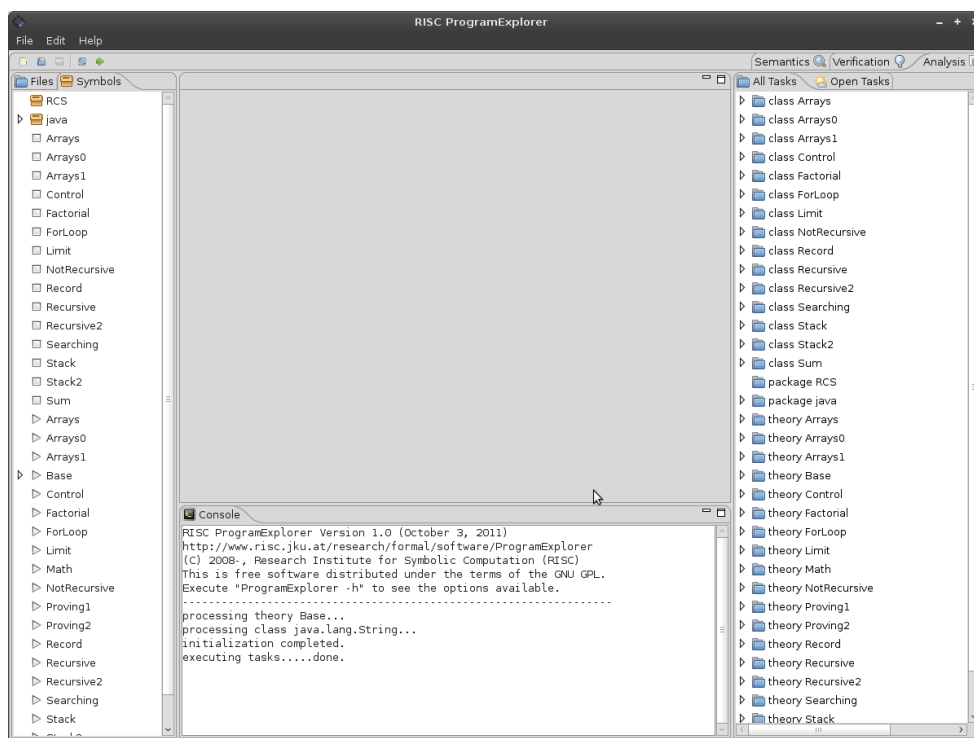


Figure 2.1.: Startup Window

File The menu entry “New File” creates a new file; files with extension `.java` are considered as program files, files with extension `.theory` are considered as specification files. The menu entry “Open File” opens such a file. The menu entry “Close File” closes the currently selected open file. The menu entry “Close All Files” closes all open files. The menu entry “Save File” saves the currently selected open file to disk.

The menu entry “Workspace...” displays the window shown in Figure 2.2. This window displays those directories that together represent the root of the package hierarchy for the RISC ProgramExplorer. The default is the list of those directories set in the environment variable `PE_CLASSPATH` (see Section E) respectively, if the variable is not set, the current working directory. The buttons “Add Directory” and “Remove Directory” modify the list, the button “Restore Directories” restores the original setting. The button “Okay” activates the current selection, the button “Cancel” discards it.

The entry “Properties...” displays the window shown in Figure 2.3. This window allows to configure various properties related to the installation of the software: the path to the executable of the Cooperating Validity Checker Lite (CVCL) version 2.0, the path to the Java compiler, the path to the Java application launcher, the path of the working directory (used e.g. for creating new files), the path for the main class of the program (the class containing method `main`). The values of these variables can be configured by various environment variables (see Section E). The button “Okay” confirms all modifications, the button “Cancel” discards them.

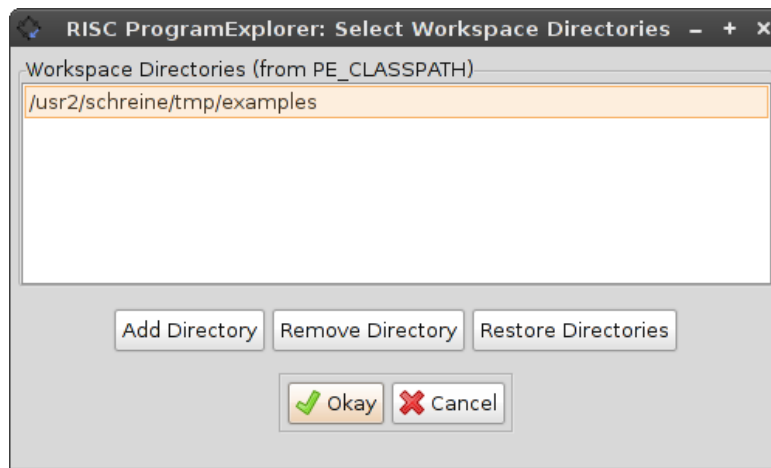


Figure 2.2.: Workspace Configuration

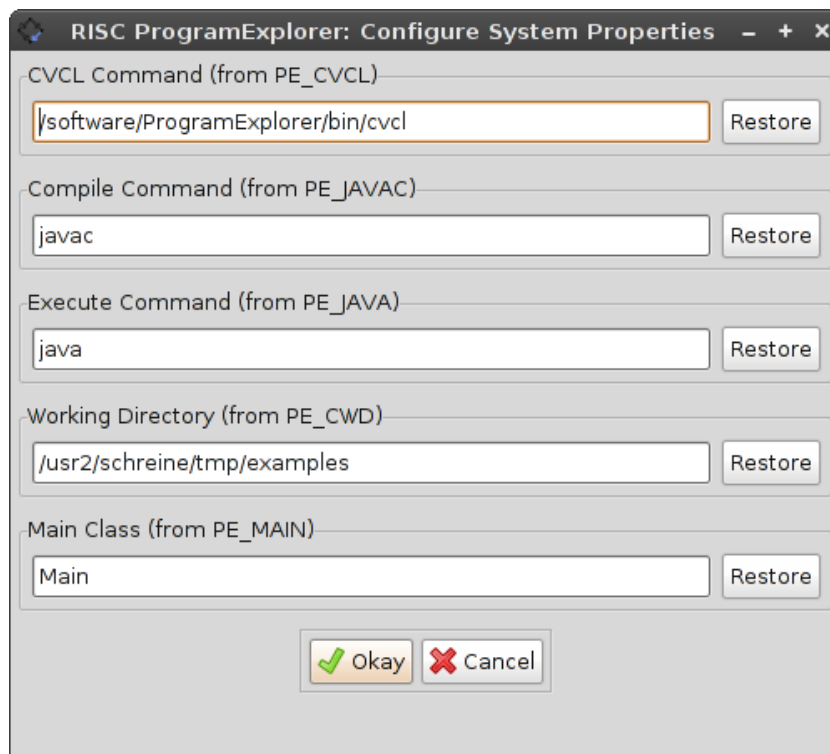


Figure 2.3.: Properties Configuration

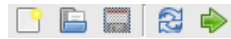



Figure 2.4.: Analyze Buttons

The menu entry “Quit” terminates the program.


Edit The menu entry “Undo” undoes the last change in the file currently being edited, the menu entry “Bigger/Smaller Font” allows to change the size of the font of the editor and of the console.


Help The entry “Online Manual” displays the hypertext version of this document; the entry “About RISC ProgramExplorer” displays a copyright message.


Below the menu, a row of buttons is displayed as shown in Figure 2.4.

New File  Like the menu option “New File”, this button creates a new file and opens it in the editing area.

Open File  Like the menu option “Open File”, this button opens an already existing file.

Save File  Like the menu option “Save File”, this button saves an open file that was modified in the editing area.

Refresh View  This button removes from the view all information (symbols and tasks) that was created by processing a class or theory.

Run Program  This button calls the Java compiler to compile the “Main” class indicated in the “Properties” configuration and calls the Java application launcher to execute it; the output is displayed in the console window (currently no input is possible).

The main area of the view is split into four areas (whose borders may be dragged by the mouse pointer). The central area (which is initially empty) is the “editing” area where program and specification files may be displayed and edited. The other three areas are:

Console This area displays textual output of the RISC ProgramExplorer, initially a copyright message. When program/specification files are processed, this area displays the success status respectively error messages, if something went wrong.

Files/Symbols In this area, the tabs “Files” and “Symbols” display the directory respectively symbol structure of the workspace as shown in Figure 2.5. By moving the mouse pointer over a directory/file, a yellow “tip” window pops up that displays the path of the corresponding directory/file respectively information on the corresponding symbol.

Double-clicking on a file opens the corresponding file in the central editing area. Right-clicking on a directory opens a pop-up menu with an option “Refresh” to refresh the display of the directory content and an option “Delete” to delete the directory (after a confirmation). Right-clicking on a file opens a pop-up menu with an option “Open” to open the file in the editing area and an option “Delete” to delete the file (after a confirmation).

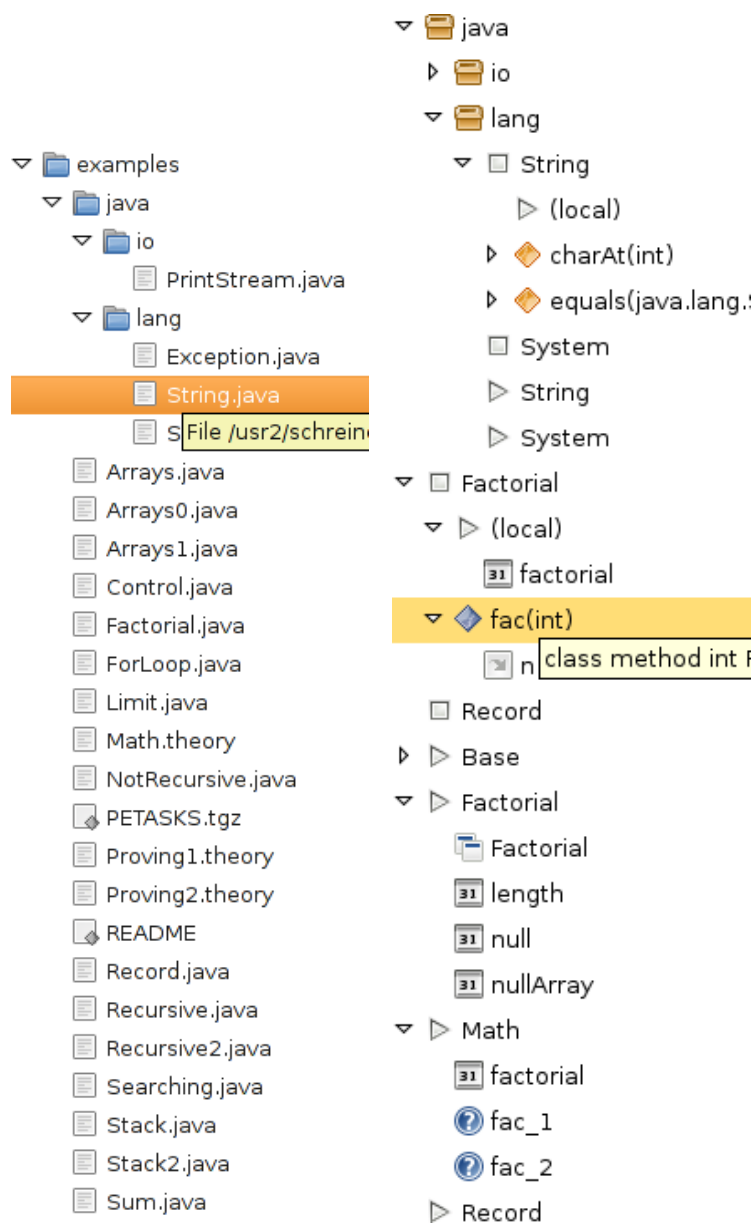





Figure 2.5.: Workspace Files/Symbols


Double-clicking on a symbol (e.g. a class symbol or a theory symbol) also opens the corresponding source file (a `.java` file or a `.theory` file) in the editor but also immediately processes it; the success of the operation is displayed in the “Console” area. There are the following kinds of symbols:


Package  A symbol denoted by a `package` declaration.


Class  A symbol introduced by a `class` declaration.


Class Variable  A symbol introduced by the declaration of a `static` variable in a class.

Object Variable  A symbol introduced by the declaration of a non-`static` variable in a class.


Class Method  A symbol introduced by the declaration of a `static` method in a class.


Object Method  A symbol introduced by the declaration of a non-`static` method in a class.


Constructor  A symbol introduced by the declaration of a constructor in a class.

Method Parameter  A symbol introduced by the declaration of a parameter in a method header in a class.


Theory  A symbol introduced by a `theory` declaration.


Type  A symbol introduced by a `TYPE` declaration in a theory.


Value  A symbol introduced by a value declaration in a theory.


Formula/Axiom  A symbol introduced by a `FORMULA/AXIOM` declaration in a theory.

All Tasks/Open Tasks In this area, the tabs “All Tasks” and “Open Tasks” display the tree of all tasks organized in task folders respectively the list of all open tasks as shown in Figure 2.6. The status of the task is indicated by in icon and the color of the description:

New Task  This task (described in red color) is *new* i.e. it has not yet been attempted to solve it.

Open/Almost Completed Task  If described in red color, this task is *open* i.e. it has been already attempted but not yet solved. If described in violet color, this task is *almost completed*, i.e. the task was solved by a proof in a previous invocation of the RISC ProgramExplorer. The corresponding proof may be replayed in the current invocation to become fully closed.

Closed Task  This task (described in blue color) is *closed*, i.e. it has been successfully solved.

Failed Task  This task (described in red) is *failed*, i.e. the task is impossible to solve (which indicates a program/specification error).

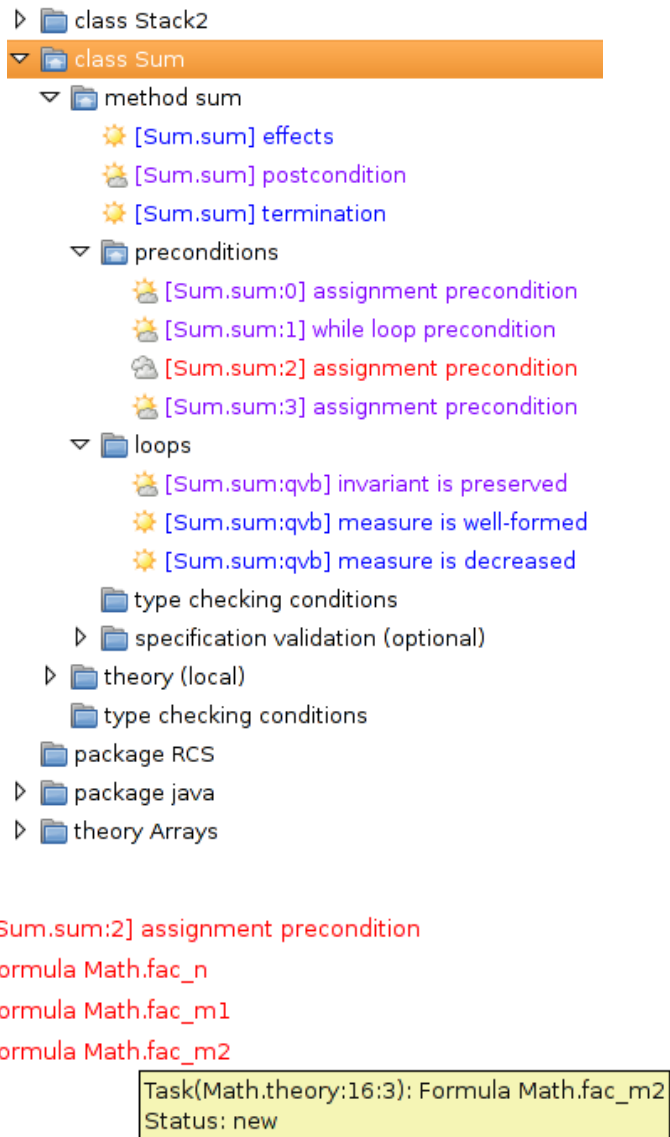


Figure 2.6.: All/Open Tasks

By default all task folders are open and display their contents (an exception are those folders that are marked as “optional”). By right-clicking one of the tabs “All Tasks” or “Open Tasks” a menu pops up whose entry “Hide Completed Tasks” closes all folders that do not contain open tasks; the entry “Show All Tasks” opens all folders again. The entry “Execute All Tasks” attempts all open tasks that have automatic solution strategies associated.

By moving the mouse pointer over a task, a yellow “tip” window pops up that displays information on the task such as the kind of task and its status. By double-clicking on the task, the position in the source code of the program or theory is displayed that triggered the creation of the task.

The solution of presumably simple tasks (such as type checking) is immediately attempted by an automatic strategy when the task is created; if this fails, the user may attempt an interactive proof to solve the task. Other presumably complex tasks (such as the proof of the correctness of loop invariants) are not automatically attempted; their solution must be explicitly triggered by the user.

By right-clicking on the task a pop-up menu shows various options depending on the kind of task: “Execute Task” attempts to solve the task e.g. by an automatic proof or, if that fails, by a computer-assisted interactive proof; “Print Task” prints information on the task (the content of the “tip” window) in the “Console” area. “Print State Proving Problem” prints a translation of the task into a proving problem in an extended logic that involves reasoning about program states. “Print Classical Proving Problem” prints a translation of the problem into a classical predicate logic proving problem. “Print Status Evidence (Proof)” shows an associated proof; “Reset Task” resets the task into the “new” state (and deletes any associated proof).

By right-clicking on a task folder, a pop-up menu shows up whose option “Execute Task” attempts to solve all tasks that have an automatic solution strategy associated (interactive proofs have to be individually triggered as shown above).

3. Programs, Theories, and Specifications

In this chapter, we are going to illustrate the basics of the RISC ProgramExplorer by some small examples (which are included in the software distribution) that describe how to write formally specified programs. In more detail, we are going to show how

- to write programs in a subset of Java called “MiniJava” (see Appendix B) and have them parsed and type-checked;
- to use a logic language (see Appendix C.1) in order to write logical theories (see Appendix C.2) and have them parsed and type-checked;
- to annotate programs with program specifications (see Appendix C) and have them parsed and type-checked;
- to prove the generated *type-checking conditions*, either by automatic proofs (using the integrated Cooperating Validity Checker Lite CVCL [3, 2]), or, if this should not succeed, by a computer-assisted interactive proof (using the integrated RISC ProofNavigator [8, 5]).

In Chapter 4, we will discuss how to investigate the semantics of these programs and how to reason about them (i.e., how to prove their correctness with respect to their specification).

3.1. Computing Factorial Numbers

This example is about the specification of the following program

```
public class Factorial
{
    public static int fac(int n)
    {
        int i=1;
        int p=1;
        while (i <= n)
        {
            p = p*i;
            i = i+1;
        }
        return p;
    }
}
```

```

    }
  }

```

The program is written in Java-syntax; it introduces a method `fac` which is supposed to return the factorial of its argument `n`.

The specification of the program is to be based on the mathematical function $factorial: \mathbb{N} \rightarrow \mathbb{N}$ which is uniquely characterized by the axioms

$$factorial(0) = 1$$

$$\forall n \in \mathbb{N}: factorial(n+1) = (n+1) \cdot factorial(n)$$

First we describe how to define the corresponding mathematical theory, next we describe how to specify the program with the help of this theory.



Theory We define a theory *Math* which introduces a function *factorial* on the natural numbers and constrains its behavior by two axioms as discussed above and also contains a logical consequence of these axioms:

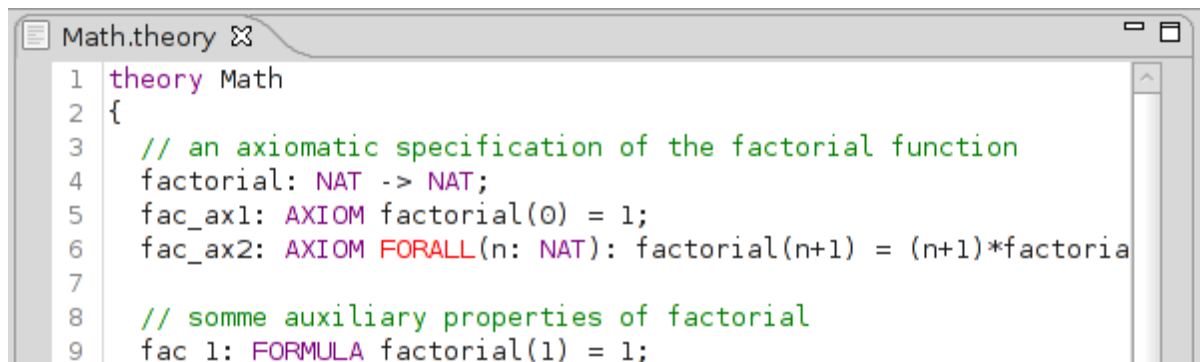
```

theory Math
{
  // an axiomatic specification of
  // the factorial function
  factorial: NAT -> NAT;
  fac_ax1: AXIOM factorial(0) = 1;
  fac_ax2: AXIOM FORALL(n: NAT):
    factorial(n+1) = (n+1)*factorial(n);

  // some auxiliary properties of factorial
  fac_1: FORMULA factorial(1) = 1;
  ...
}

```

We can use the RISC ProgramExplorer to write this theory in a file *Math.theory* in the unnamed top-level package as follows: we select the button *New File* , enter the file name *Math.theory*, and press *Okay*. In the central region a new editing area titled *Math.theory* opens; we enter above theory declaration and press the button *Save File* . The RISC ProgramExplorer window has then the state shown in Figure 3.1. The theory is displayed as



```

Math.theory
1 theory Math
2 {
3   // an axiomatic specification of the factorial function
4   factorial: NAT -> NAT;
5   fac_ax1: AXIOM factorial(0) = 1;
6   fac_ax2: AXIOM FORALL(n: NAT): factorial(n+1) = (n+1)*factoria
7
8   // somme auxiliary properties of factorial
9   fac_1: FORMULA factorial(1) = 1;

```

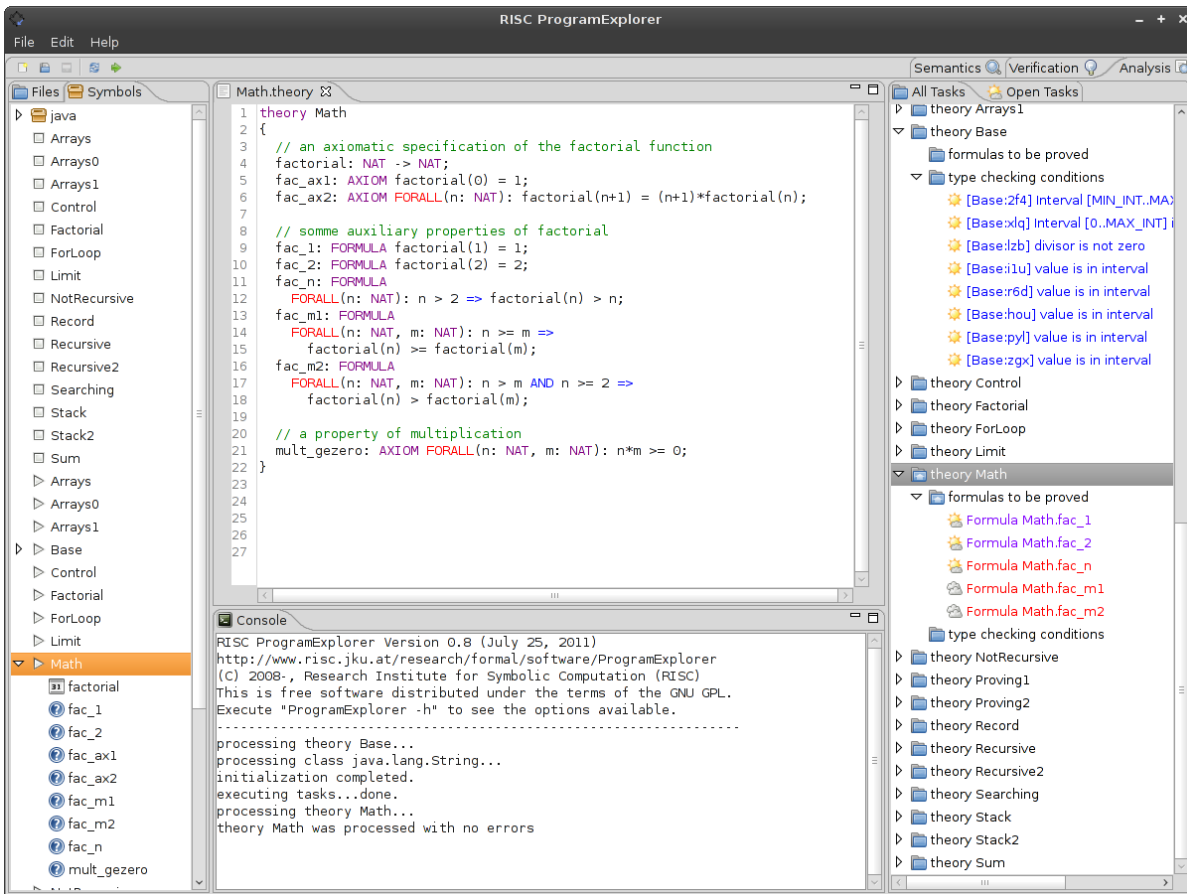


Figure 3.1.: A Logic Theory

with colors indicating keywords of the specification language. Identifiers are *active*, e.g. by double-clicking on *factorial* the identifier is highlighted, the tab *Symbol* on the left side of the window highlights the corresponding symbol and the *Console* area displays

```

value factorial: (NAT) -> NAT
factorial: (NAT) -> NAT

```

(likewise the identifiers *Math*, *fac_1*, and *fac_2* can be double-clicked). Moving the mouse pointer over the symbol on the left tab displays a corresponding yellow “tip” window, clicking with the right mouse-button allows to choose between *Print Symbol* and *Print Declaration*. Choosing the symbol *Math* and selecting *Print Symbol* displays in the console area output similar to

```
theory Math (file ../../examples/Math.theory)
```

Selecting *Print Declaration* displays

```

theory Math
{
  factorial: (NAT) -> NAT;
}

```



```

    fac_ax1: AXIOM factorial(0) = 1;
    fac_ax2: AXIOM FORALL(n: NAT): factorial(n+1) = ...;
    ...
}

```

If we introduce in the declaration an error, e.g. by mistyping the function name as *factorials* in axiom

```
fac_ax1: AXIOM factorials(0) = 1;
```

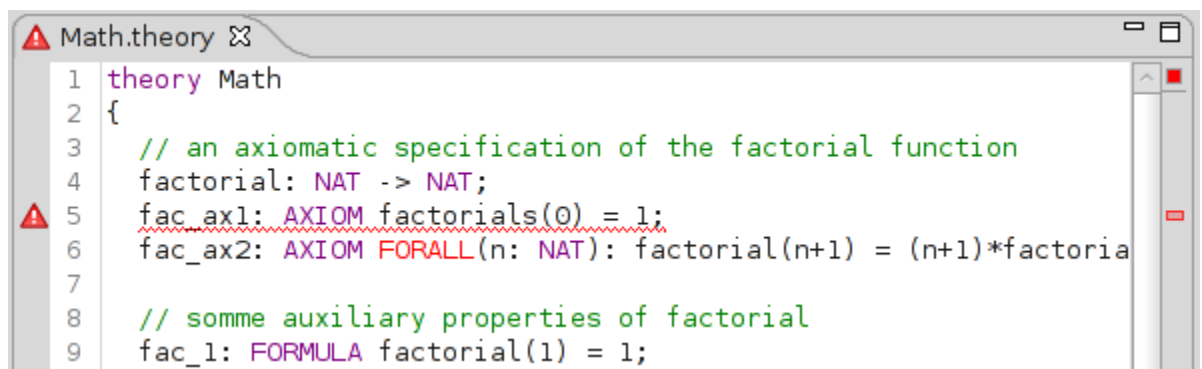
the *Console* area shows the output



```

ERROR (Math.theory:5:16):
  there is no value named factorials
theory Math was processed with 1 error

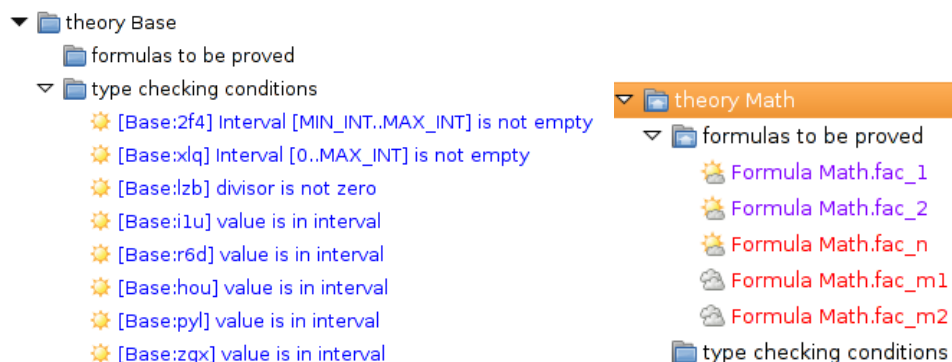
```

In the editing area, the theory is then displayed as







The position of the error in the file is indicated by an icon  on the left bar, by a corresponding red marker on the right bar and by underlining the syntactic phrase in red; moving the mouse pointer over the icon on the left or over the marker on the right displays the corresponding error message. The same icon at the top of the editing tab and in the tab *Symbols* indicates that the theory has an error. Moving the mouse pointer over the red square on the top-right corner of the editing area displays the number of errors in the theory. After fixing the error and pressing the button *Save File* , the correct state is restored.

The tab *All Tasks* on the right now contains the following entries:



The folder *theory Base* contains a subfolder *type checking conditions* which holds those tasks that arose from type-checking a built-in “base” theory. The task labels shortly describe the tasks; the automatically generated tags of the form $[Base:ccc]$ allow unique referencing; the blue font indicates that all the tasks could be performed with the help of an automatic strategy. Moving the mouse pointer over the tasks (respectively right-clicking the tasks and selecting the option *Print Task*) shows the task descriptions, e.g.

```
Task: [Base:2f4] Interval [MIN_INT..MAX_INT] is not empty
Status: done (solved by decision procedure)
Type: verify type checking condition
Goal formula: MIN_INT <= MAX_INT
```

The special icon  indicates that it contains (subfolders with) open tasks to be performed. Indeed it contains a subfolder *formulas to be proved* with some open tasks labeled in red referring to formulas that have not yet been proved. The icon  indicates that the corresponding task is “new”, while the icon  indicates that the task has a previously constructed proof attempt (i.e. an incomplete proof that may be replaced and completed in the current invocation). On the other hand, those tasks with icon  and violet labels are already closed; they refer to formulas that have been proved in a previous invocation such that the proofs may be simply replayed in the current invocation.

Right-clicking any of these tasks and selecting the option *Print Classical Proving Problem* shows the detailed proofs to be performed for performing the tasks: in the case of “Formula *fac_1*”, this proof is

```
Declarations:
STRING: TYPE;
MIN_INT: INT = -2147483648;
MAX_INT: INT = 2147483647;
...
Goal: factorial(1) = 1
```

Right-clicking the task and selecting the option *Execute Task* replays the (simple) previously constructed proof; selecting the option *Show Proof* displays the view shown in Figure 3.2. This is essentially a view on the [RISC ProofNavigator](#) [8, 5], the computer-assisted interactive proving assistant integrated into the RISC ProgramExplorer. The task generated by the RISC ProgramExplorer was translated into a proving problem of the RISC ProofNavigator and performed with human aid (in this case, by invoking the `auto` command for heuristic instantiation of quantified formulas). By clicking on the individual nodes of the proof tree shown to the left, the corresponding proof situations are displayed; pressing the button *View Declarations* displays the declarations related to the proof. By selecting the tab “Analysis” we return to the original view. Right-clicking the task and selecting “Reset Task” resets the task to its original “new” state (deleting the previously constructed proof) such that a fresh interactive proof attempt can be performed.

Program We can now create (in analogy to file *Math.theory*) a program file *Factorial.java* which holds the program class *Factorial* described above. Saving the file type-checks it and

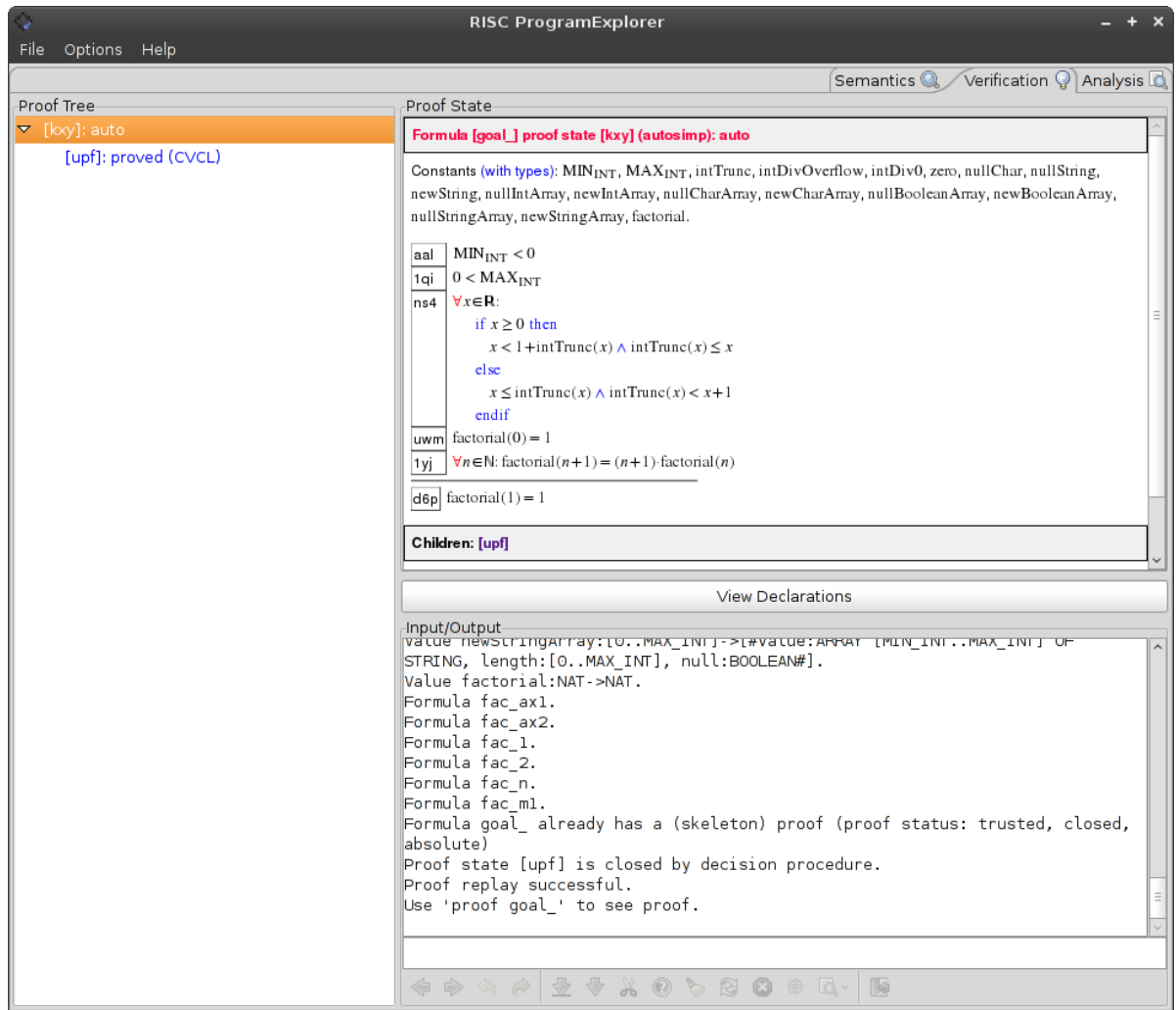


Figure 3.2.: An Interactive Proof

creates in tab *Symbols* a class symbol *Factorial* with method symbol *fac* and parameter symbol *n*. As for theories, also the identifiers in the source file are active, double-clicking e.g. on *fac* in the editing area prints the method declaration in the *Console area* and highlights the symbol *fac* in the *Symbols* tab. Correspondingly, double-clicking the symbol *fac* moves the editor to the position of the declaration of the method and highlights its header.




To formally specify the behavior of the method *fac* with the help of the theory *Math*, we annotate the class *Factorial* by special program comments `/*@ ... @*/` as follows:

```
/*@
theory uses Math {
  // the mathematical factorial function
  factorial: NAT -> NAT = Math.factorial;
}
```

```

@*/
public class Factorial
{
    public static int fac(int n) /*@
        requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
        ensures VALUE@NEXT = factorial(VAR n);
    @*/
    {
        int i=1;
        int p=1;
        while (i <= n) /*@
            invariant VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT
                AND 1 <= VAR i AND VAR i <= VAR n+1
                AND VAR p = factorial(VAR i -1);
            decreases VAR n - VAR i + 1;
        @*/
        {
            p = p*i;
            i = i+1;
        }
        return p;
    }
}

```

The RISC ProgramExplorer window has then the state shown in Figure 3.3. The actual program code is displayed as shown in Figure 3.4. Annotations can become “folded” away from the program source code; clicking on the icon  folds the annotation, clicking on the icon  unfolds it again. Moving the mouse pointer over the icon  displays the content of the folded annotation in a yellow “tip” window.

The annotation `theory ...` before the class declaration introduces the “local” theory for the class i.e. those entities that may be further on referenced by short names; the uses `Math` clause indicates that the local theory refers to entities of the previously defined theory `Math`. The local theory is simple: it just defines a function *factorial* by the corresponding function in theory `Math` which can be referenced by the long name `Math.factorial`. Alternatively, we might have referred in the following directly to `Math.factorial` (however, even then an empty declaration `theory uses Math { }` is required because we refer to theory `Math`) or we might have just axiomatized the function *factorial* directly in the local theory (without referring to theory `Math` at all).

The annotation `requires ...ensures ...` after the header of method *fac* introduces a method specification by a precondition (`requires ...`) that describes the assumptions on the prestate of the method call (in particular constraints of the method arguments) and a postcondition (`ensures ...`) that describes the obligation on the poststate of the method call (in particular obligations on the method result). In our case, the precondition

```
requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
```

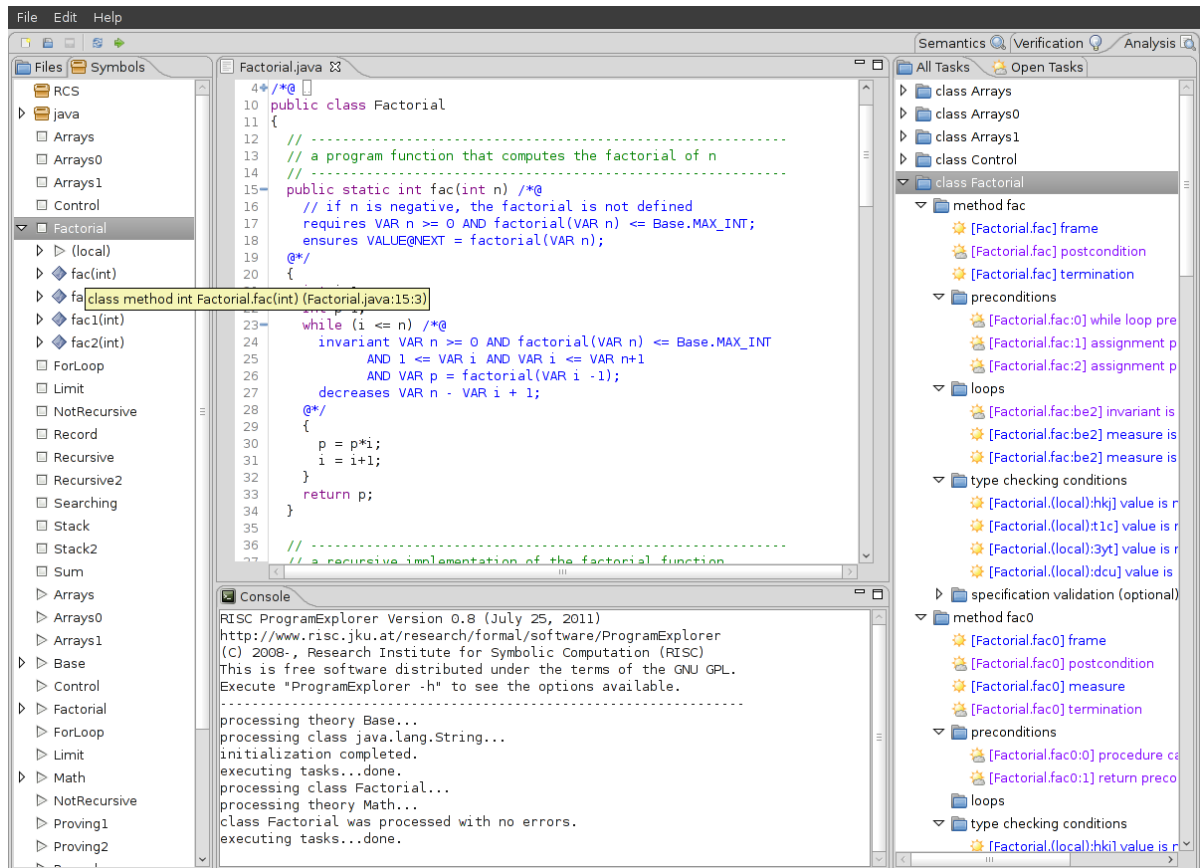


Figure 3.3.: A Program Class

states that the value of the program variable n (the method parameter) indicated by the term $\text{VAR } n$ must not be negative when the method is called and that the value of $\text{factorial}(n)$ must not exceed the limit of the program type int indicated by the mathematical constant MAX_INT in the base theory. The postcondition

$$\text{ensures VALUE@NEXT} = \text{factorial}(\text{VAR } n);$$

states the method result indicated by the term VALUE@NEXT must be identical to the value of the logical function factorial when applied to the value of n .

Finally, the body of the `while` loop is annotated by a loop invariant and a termination term: the loop invariant essentially states the relationship of the prestate of the loop to the poststate of every iteration of the loop body; the termination term denotes a non-negative integer number that is decreased by every iteration of the loop. In our case, the invariant

$$\begin{aligned} &\text{invariant } \text{VAR } n \geq 0 \text{ AND } \text{factorial}(\text{VAR } n) \leq \text{Base.MAX_INT} \\ &\quad \text{AND } 1 \leq \text{VAR } i \text{ AND } \text{VAR } i \leq \text{VAR } n+1 \\ &\quad \text{AND } \text{VAR } p = \text{factorial}(\text{VAR } i - 1); \\ &\text{decreases } \text{VAR } n - \text{VAR } i + 1; \end{aligned}$$

limits the range of the iteration counter i and states that the value of the program variable p is

```

4+ /*@
10 public class Factorial
11 {
12     // -----
13     // a program function that computes the factorial of n
14     // -----
15     public static int fac(int n) /*@
16         // if n is negative, the factorial is not defined
17         requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
18         ensures VALUE@NEXT = factorial(VAR n);
19     @*/
20     {
21         int i=1;
22         int p=1;
23         while (i <= n) /*@
24             invariant VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT
25                 AND 1 <= VAR i AND VAR i <= VAR n+1
26                 AND VAR p = factorial(VAR i -1);
27             decreases VAR n - VAR i + 1;
28         @*/
29         {
30             p = p*i;
31             i = i+1;
32         }
33         return p;
34     }

```

Figure 3.4.: The Program Class in Detail

identical to the factorial of the value of i . The termination term

```
decreases VAR n - VAR i + 1;
```

states that the value of i is decremented by every loop iteration but does not become bigger than $n+1$.

Type-checking the annotations and semantically processing the class gives rise to a number of tasks in folder *class Factorial* organized in a couple of subfolders (as usual double-clicking on the tasks highlights the corresponding source code positions; moving the mouse over the tasks shows their description). The folder *type checking conditions* contains those tasks arising from type checking the annotations; all of them can be closed by an automatic strategy. The other conditions refer to the correctness of the method with respect to its specification; they will be discussed in Chapter 4.

3.2. Searching for Records

This example deals with the specification of a program that searches in an array of records for a record with a specific key. The example is based on the following program class:

```
class Record
```

```
{
    String key;
    int value;

    Record(String k, int v)
    {
        key = k;
        value = v;
    }

    boolean equals(String k)
    {
        boolean e = key.equals(k);
        return e;
    }

    public static int search(Record[] a, String key)
    {
        int n = a.length;
        for (int i=0; i<n; i++)
        {
            Record r = new Record(a[i].key, a[i].value);
            boolean e = r.equals(key);
            if (e) return i;
        }
        return -1;
    }

    public static void main()
    {
        int N = 10;
        Record[] a = new Record[N];
        for (int i=0; i<N; i++)
            a[i] = new Record("abc", i);
        a[5] = new Record("xyz", 5);
        int i = search(a, "xyz");
        System.out.println(i);
    }
}
```

This program introduces an object type *Record* with a string field *key* and an integer field *value*. The type has an constructor to build a record from an given string and integer and a method *equals* that allows to check whether the record has the denoted key. This function calls the method *equals* on object type *String*; while this class is part of the Java standard library, it has to be explicitly defined in a file accessible to the RISC ProgramExplorer. We therefore introduce a dummy class

```
package java.lang;
public class String
{
    public boolean equals(String s)    return false;
}
```

solely for declaring the method *equals* (without caring for the actual representation of strings or the actual implementation of the method). Unlike real Java, our programming language only allows to call program methods with return values to initialize/assign to variables, not as parts of program expressions¹. The two statements

```
boolean e = key.equals(k);
return e;
```

in the body of *equals* can therefore *not* be merged into one.

The core of the program is the method *search* which takes an array *a* of records and a key and returns the index of the first record in *a* that contains that key (or -1 , if there is no such record). The core of the method body is represented by the two statements

```
Record r = new Record(a[i].key, a[i].value);
boolean e = r.equals(key);
```

The first statement builds a record *r* from the key and the value of record *a*[*i*]. The second statement calls the method *equals* on *r* to compare its key with *key*. This apparently clumsy way of using the function *equals* is necessary because the specification formalism considers object variables (variables of object types) to hold *object values* rather than *object references* (which considerably simplifies reasoning because then the modification of an object via one variable cannot affect an object referenced by another variable).

However, since the programming language Java (like most programming languages) lets object variables hold references, the semantics of our programming language would deviate from classical program semantics. Therefore the type checker ensures that two different program variables cannot refer to the same object; consequently it does not make any difference whether an object variable holds an object value or an object reference. Consequently, if *equals* would modify its record, above solution would not update array *a* (independent of whether object variables hold object values or object references) while the solution

```
Record r = a[i];
boolean e = r.equals(key);
```

would update *a* in a language with reference semantics for objects but not in a language with value semantics. For similar reasons, the even shorter solution

```
boolean e = a[i].equals(key);
```

is also (even syntactically) prohibited. See Appendix B for a more thorough description of the constraints of our programming language compared to Java.

¹The reason is that methods may cause side effects and we do not want the computation of program expressions to cause side effects

The method *main* of the program creates an array, fills it with values, updates it, and calls the method *search* in the usual way. It also calls the method *System.out.println* of the Java Standard API. This method has to be declared with the help of the dummy classes

```
package java.lang;
import java.io.*;
public class System
{
    public static PrintStream out;
}

package java.io;
public class PrintStream
{
    public void println(boolean b)
    public void println(int i)
    public void println(char c)
    public void println(String s)
    public void println()
}
}
```

Type-checking the class *Record* creates a new theory *Record* in the same package as the class. Right-clicking this theory from the tab *Symbols* and selecting *Print Declaration* displays

```
theory Record uses java.lang.String, Base
{
    Record: TYPE =
        [#key: java.lang.String.String, value: Base.int,
         null: BOOLEAN, new: INT#];
    null: Record;
    newObject: INT -> Record =
        LAMBDA(x: INT): null WITH .null:=FALSE WITH .new:=x;
    Array: TYPE =
        [#value: ARRAY Base.int OF Record, length: Base.nat,
         null: BOOLEAN#];
    nullArray: Array;
    newArray: Base.nat -> Array;
    newArrayAxiom:
        AXIOM FORALL(x: Base.nat):
            LET y = newArray(x) IN
            NOT y.null AND y.length = x AND
            (FORALL(i: Base.nat): i < x => y.value[i] = null);
}
```

which introduces the following entities that mathematically model the program operations that involve program type *Record*.

- a logical record type *Record* which contains (among other fields) one field for each object variable in class *Record*, the specification language considers program variables

of object type `Record` to hold values of the logical type *Record* (more details later);

- a constant *null* representing a logical counterpart of the `null` pointer of type `Record`;
- a function *newObject* which returns a non-null object of type *Record* as created by a call of a constructor of class `Record`;
- a type *Array* representing the logical counterpart of the program type `Record[]`;
- a type *nullArray* representing the logical counterpart of the `null` pointer of the type `Record[]`;
- a function *newArray* which returns an *Array*;
- an axiom *newArrayAxiom* which describes the result *newArray* as a non-null array of a certain length whose entries are *null*.

There are some subtleties about the modeling a program object of type `Record` as a mathematical *Record* that should be known for their proper use:

1. If the *null* field of a valid *Record* value is not `FALSE`, the value represents the `null` pointer of type `Record`. The `Record` pointer `null` is thus not only represented by the single mathematical constant *null* but by every *Record* value whose *null* field is `TRUE` (the constant *null* is just an unknown *Record* value for which the opposite cannot be proved)². A precondition that asserts that a `Record` object is not `null` must therefore state for the corresponding *Record* value *r* that `NOT r.null` holds (the condition `r != null` alone is too weak).
2. The field *new* is initialized to an unknown value at the creation of a record; two records created in different constructor calls can thus not be proved equal.

The program is now specified with the help of the following local theory

```
/*@
theory uses Base, Record, java.lang.String, java.io.PrintStream
{
  String: TYPE = java.lang.String.String;
  notFound: PREDICATE(Record.Array, Base.int, String) =
    PRED(a:Record.Array, n: Base.int, key: String):
      FORALL(i:INT):
        0 <= i AND i < n => a.value[i].key /= key;
}
@*/
class Record { ... }
```

which introduces a predicate *notFound* to describe that in an array *a* of records, all positions less than *n* hold records whose keys are different from *key*.

The program method *search* can now be specified as

²By this modeling, the proper handling of update operations is simplified: writing a field in a non-null object automatically yields another non-null object.

```

public static int search(Record[] a, String key) /*@
  requires
    NOT (VAR a).null AND NOT (VAR key).null AND
    (FORALL(i:INT): 0 <= i AND i < VAR a.length =>
      NOT (VAR a).value[i].null AND
      NOT (VAR a).value[i].key.null);
  ensures
    (LET result=VALUE@NEXT, n = (VAR a).length IN
      IF result = -1 THEN
        notFound(VAR a, n, VAR key)
      ELSE
        0 <= result AND result < n AND
        notFound(VAR a, result, VAR key) AND
        (VAR a).value[result].key = VAR key
      ENDIF);
  @*/
{ ... }

```

The method's precondition states that *search* must not be called with the array `null` as argument and that its result is either `-1` (indicating that the given key has not been found in the array) or that its result is the smallest index of the array such that the corresponding record has the denoted key. The specification makes use of local logical variables *result* and *n* representing the return value of the method and the length of the method parameter *a*.

The core loop of the method's body can be annotated as

```

for (int i=0; i<n; i++)
  /*@
    invariant
      NOT (VAR a).null AND NOT (VAR key).null AND
      (FORALL(i:INT): 0 <= i AND i < VAR a.length =>
        NOT (VAR a).value[i].null AND
        NOT (VAR a).value[i].key.null) AND
      VAR n = (VAR a).length AND
      0 <= VAR i AND VAR i <= VAR n AND
      notFound(VAR a, VAR i, VAR key) AND
      (RETURNS@NEXT =>
        VAR i < VAR n AND
        (VAR a).value[VAR i].key = VAR key AND
        VALUE@NEXT = VAR i);
    decreases VAR n - VAR i;
  @*/
{ ... }

```

to give a suitable invariant and termination term.

The state after type-checking the annotated program is shown in Figure 3.5. All type checking conditions can be automatically solved; the verification of the other conditions proceeds in a similar way as discussed in Chapter 4.

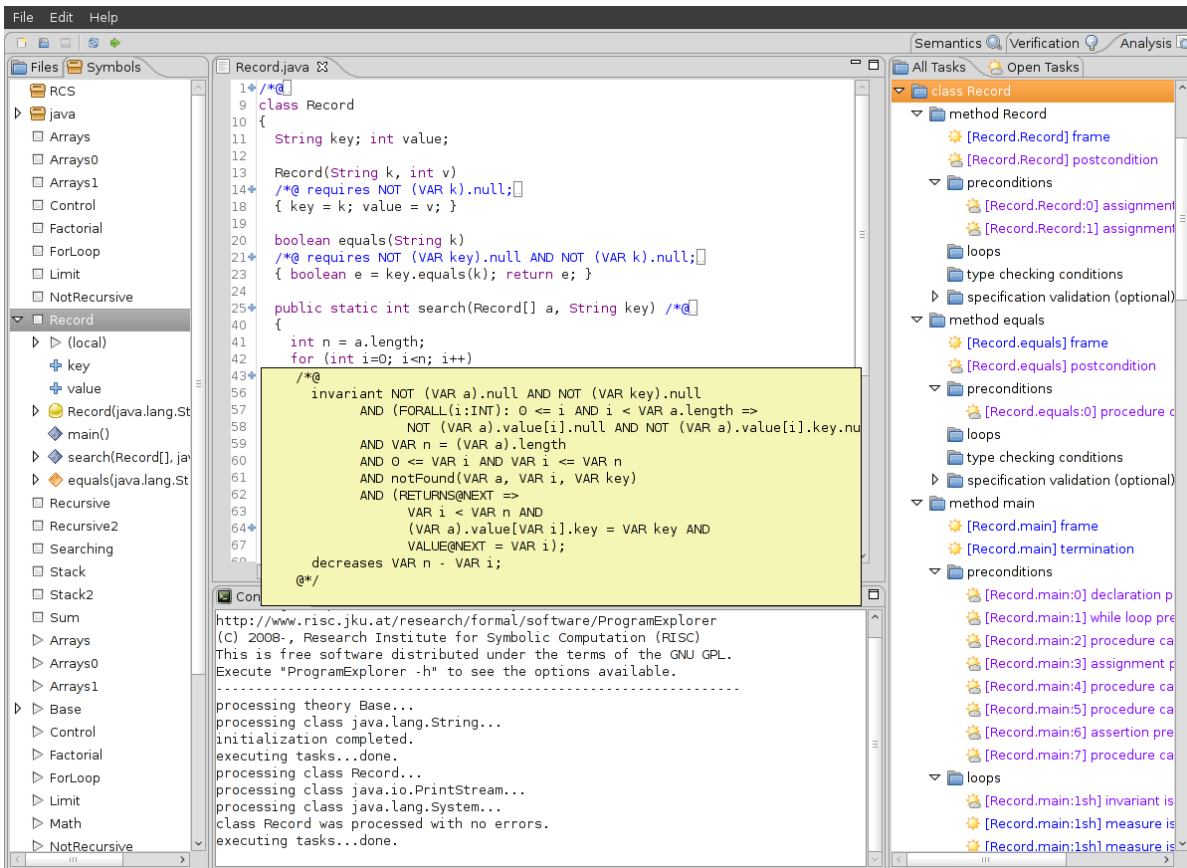


Figure 3.5.: Searching for Records

3.3. Failed Tasks and Interactive Proofs


The previous sections dealt mainly with tasks (type-checking conditions) that could be automatically solved by the integrated decision procedure. In this section, we discuss in somewhat more detail what happens if the automatic decision procedure does not succeed, in particular because it is impossible to achieve the goal of the task (i.e. the task is a priori doomed to fail).

One kind of failure (the goal of a proof is contradictory) can be demonstrated by the theory

```
theory Proving1
{
  // type-checking task can be proved unsatisfiable
  a: INT = 1;
  b: INT = 0;
  T: TYPE = [a..b];
}
```

which attempts to erroneously define an empty type T (by an interval of the integers whose lower bound is bigger than the upper bound). Type-checking this theory generates the task

```
Task(Proving1.theory:6:13):
  [ia] Interval [a..b] is not empty
Status: new
Type: verify type checking condition
Goal formula: a <= b
```

The task is labeled in red (which indicates task status “open”) with the icon  (which indicates that an automatic proof has been attempted but failed). If we select *Execute Task*, an interactive proof with the following root situation is created:



Formula [goal_] proof state [kxy] (autosimp)

Constants (with types): MIN_{INT}, MAX_{INT}, intTrunc, intDivOverflow, intDiv0, zero, nullChar, nullString, newString, nullIntArray, newIntArray, nullCharArray, newCharArray, nullBooleanArray, newBooleanArray, nullStringArray, newStringArray, *a*, *b*.

aal	MIN _{INT} < 0
1qi	0 < MAX _{INT}
ns4	$\forall x \in \mathbb{R}$: if $x \geq 0$ then $x < 1 + \text{intTrunc}(x) \wedge \text{intTrunc}(x) \leq x$ else $x \leq \text{intTrunc}(x) \wedge \text{intTrunc}(x) < x + 1$ endif

This situation depicts as knowledge three formulas derived from the base theory and no goal formula. The reason for the disappearance of the goal can be inferred by pressing the button “View Declarations” which depicts (among others)

$$\left. \begin{array}{l} a \in \mathbb{Z} = 1 \\ b \in \mathbb{Z} = 0 \end{array} \right\}$$

From this, the goal $a \leq b$ was automatically reduced to `FALSE` and consequently discarded from the proof situation. The proof could be thus only completed if the knowledge were inconsistent (such that `FALSE` could be derived) which is here not the case. We therefore press the button *Quit Proof*  and return (after a confirmation) to the “Analysis” view with task status still unchanged as indicated by the icon .

Another kind of failure (the goal of a proof is unprovable) can be demonstrated by the theory

```
theory Proving2
{
  // type-checking task cannot be solved
  a: INT;
  b: INT;
  T: TYPE = [a..b];
}
```

which defines a type as a subrange of the integers with unspecified bounds *a* and *b*. Type checking this theory yields the task

```
Task(Proving2.theory:6:13):
```

```

[ia] Interval [a..b] is not empty
Status: new
Type: verify type checking condition
Goal formula: a <= b

```

Also for this task the automatic proof fails and it is consequently indicated by the red label as “open”. If we select *Execute Task*, an interactive proof with the following root situation is created:

Formula [goal_] proof state [kxy] (autosimp)

Constants (with types): MIN_{INT}, MAX_{INT}, intTrunc, intDivOverflow, intDiv0, zero, nullChar, nullString, newString, nullIntArray, newIntArray, nullCharArray, newCharArray, nullBooleanArray, newBooleanArray, nullStringArray, newStringArray, *a*, *b*.

aal	MIN _{INT} < 0
1qi	0 < MAX _{INT}
ns4	$\forall x \in \mathbb{R}$: if $x \geq 0$ then $x < 1 + \text{intTrunc}(x) \wedge \text{intTrunc}(x) \leq x$ else $x \leq \text{intTrunc}(x) \wedge \text{intTrunc}(x) < x + 1$ endif

[ia] $a \leq b$

The knowledge is consistent but does not say anything about the variables *a* and *b* referred in the goal. By pressing the button *View Declarations*, the declarations related to the proof are shown below:

	$a \in \mathbb{Z}$
	$b \in \mathbb{Z}$

Since *a* and *b* have no defined values, there is no chance of completing the proof successfully. We therefore press again the button *Quit Proof*  and return to the “Analysis” view.

It may also (infrequently) happen that the type-checker generates tasks that cannot be solved by the integrated automatic decision procedure but can be solved by an interactive proof. If this should be the case, the proof remains *persistent* across multiple invocations of the RISC ProgramExplorer; it can be later displayed (menu option *Print Status Evidence*) and also replayed again (menu option *Execute Task*). Menu option *Reset Task* erases the proof and returns the task to state “new”.

If there is no ongoing interactive proof, the user may also manually switch to the “Verification” view and enter declarations and commands of the RISC ProofNavigator. The use of the software is then essentially the same as in the standalone version of the RISC ProofNavigator.

Thus the RISC ProofNavigator is already fully integrated into the task solution framework of the RISC ProgramExplorer; while this is not of major importance for the purpose of verifying type checking conditions, it becomes essential for verifying program correctness as discussed in the following chapter.

4. Semantics and Verification

The core idea of the RISC ProgramExplorer is to translate every program into its “semantic essence”, i.e. a declarative form that exhibits all that is to be known about the program from the mathematical point of view. This semantics is exhibited to the user for investigation and analysis (programming and specification errors may so be detected early); it also represents the basis for the subsequent verification of the program. Given a method

```
method (...) /*@ ensures f_s @*/  
{ c }
```

with specification f_s and body c , c is translated into a formula f_c which describes the effect of c ; the proof that the method satisfies its specification is then essentially the proof of

$$f_c \Rightarrow f_s$$

i.e., that the effect of c implies the specification.

In somewhat more detail, we translate every program command (e.g. the body of a program method) into a logic formula that describes the relationship between the value of every program variable x in the prestate of the execution (this value is in the formula denoted as `old x`) to the value of the variable in the poststate (this value is denoted as `var x`). For instance, the program command

```
x = x+y
```

is translated to the formula (the *state relation*)

```
var x = old x + old y
```

This formula, however, does not explicitly say anything about the variables whose values are not changed (y, z, \dots) such that the post-state value of such variables is undefined. To overcome this problem, we additionally determine the *frame* of the variable which is the set of program variables that may be changed by the command. For above command, the program frame would be just x , from which we can deduce

```
var y = old y  $\wedge$  var z = old z  $\wedge$  ...
```

i.e., `var v = old v` , for every program variable v different from x .

A sequence of commands

```
y = y+1;  
x = x+y
```

is handled by first translating the individual commands to state relations

```
var y = old y + 1 ∧ var x = old x
var x = old x + old y ∧ var y = old y
```

with common frame x, y . These formulas are then combined to a single state relation

```
∃x0, y0 :
  y0 = old y + 1 ∧ x0 = old x ∧
  var x = x0 + y0 ∧ var y = y0
```

where x_0, y_0 represent the values of the program variables after the execution of the first command; these values are not visible as pre/post-state values of the combined command and therefore “hidden” by existential quantification. The combined formula can then be logically simplified by inserting the definitions of these values. This results in the state relation

```
var x = old x + old y + 1 ∧ var y = old y + 1
```

with frame x, y .

Likewise, a local variable declaration

```
{var x; c}
```

is translated to a state relation of form

```
∃x0, x1 : fc[x0/old x, x1/var x]
```

where f_c denotes the state relation of c ; in this formula, all references to `old x` respectively `var x` are substituted by references to existentially quantified variables x_0 respectively y_0 . A conditional command

```
if (e) then c1 else c2
```

is translated to a state relation

```
if fe then f1 else f2
```

(which is syntactic sugar for $(f_e \Rightarrow f_1) \wedge (\neg f_e \Rightarrow f_2)$) where f_e, f_1, f_2 denote the translations of e, c_1, c_2 . Finally, a while loop

```
while (e) /*@ invariant f; @*/ c
```

is translated to the state relation

```
f ∧ ¬f'e
```

i.e. a loop invariant just denotes (an upper bound approximation of) the loop’s state relation; the formula f'_e describes that the value of e in the poststate is `false`.

The translation of program commands to state relations is more thoroughly discussed in Appendix A; in the following we present its application to some sample programs.

4.1. Computing Factorial Numbers

In this section, we investigate the semantics and the verification of the program for computing factorial numbers already presented in Section 3.1:


```

public static int fac(int n) /*@
    requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
    ensures VALUE@NEXT = factorial(VAR n);
/*@/
{
    int i=1;
    int p=1;
    while (i <= n) /*@
        invariant VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT
            AND 1 <= VAR i AND VAR i <= VAR n+1
            AND VAR p = factorial(VAR i -1);
        decreases VAR n - VAR i + 1;
    @*/
    {
        p = p*i;
        i = i+1;
    }
    return p;
}

```

Method Semantics

Right-clicking the item *fac* in the “Symbols” tree and selecting the menu entry “Show Semantics” lets the RISC ProgramExplorer switch to the “Semantics” view and display the information shown in Figure 4.1.

The view is split into several parts:

Method Definition In the left pane, the definition of the method is shown with a column of active buttons (squares turning blue when the mouse is moved over them) to select the whole method body or an individual command of the body; additionally for each command a radio button is shown (see item “Condition Box” for their usage).

Statement Knowledge In the right pane, all information derived from the analysis of the selected command is shown (see below for a detailed explanation of this information).

Condition Box Below the method definition, a text box is shown. The user may select via the radio buttons to the left of the method body a single command, enter a state condition into the textbox, and fix this condition (via the two radio buttons below the box) to be either a “Prestate” or a “Poststate” condition. By pressing the button “Submit”, an additional view will be displayed that exhibits the effect of “pushing” the condition through the rest of the method body (we will later give an example).

In the pane “Bodsy/Statement Knowledge”, the following information is displayed for the selected method/command (which is also shown at the bottom of the pane):

Pre-State Knowledge A condition which is known to hold whenever the command starts execution.

The screenshot shows the RISC Program Explorer interface. The left pane displays the source code for the `fac` method with annotations and a tree view. The right pane shows the formal semantics for the method.

Factorial.fac

```
requires old n ≥ 0 ∧ factorial(old n) ≤ Base.MAXINT
ensures value@next = factorial(var n)

public static int fac(int n) /*@
  requires OLD n ≥ 0 AND factorial(OLD n)
  ensures VALUE@NEXT = factorial(VAR n);
  @*/
{
  int i = 1;
  int p = 1;
  while (i ≤ n) /*@
    invariant VAR n ≥ 0 AND factorial(VAR p) = factorial(VAR i)
    decreases OLD n-OLD i+1;
  @*/
  {
    p = p*i;
    i = i+1;
  }
  return p;
}
```

Select a statement and define a condition for its pre/poststate:

Submit Reset Prestate Poststate

Body Knowledge

[Show Original Formulas]

Pre-State Knowledge

$$\text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}}$$

Effects

executes: false, continues: false, breaks: false, returns: true
variables: -; exceptions: -

Transition Relation

$$(\exists i \in \text{Base.int}. i = \text{old } n + 1 \wedge 1 \leq i \wedge \text{value@next} = \text{factorial}(i-1)) \wedge \text{old } n \geq 0$$

$$\wedge$$

$$\text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge \text{returns@next}$$

Termination Condition

$$\text{executes@now} \Rightarrow \text{old } n \geq 0$$
Figure 4.1.: Semantics of Method `fac`

Effects The effects that may be produced by the execution of the command.

Variables The set of variables that may be modified by the command.

Exceptions The set of exceptions that may be thrown by the command.

Executes True, only if the command may terminate with the execution of a “normal” statement.

Continues True, only if the command may terminate by executing `continue`.

Breaks True, only if the command may terminate with the execution of `break`.

Returns True, only if the command may terminate with the execution of `return`.

Transition Relation The transition relation of the command, i.e., a formula describing the relationship between the pre-state and the post-state of the command execution.

Termination Condition The termination condition of the command, i.e., a formula that describes a condition on the pre-state of the command execution which guarantees the termination of the execution.

By default, all formulas are shown after extensive logical simplification. By selecting the link “Show Original Formulas” the original formulas as constructed by the formal calculus are displayed (these are potentially huge but may be of interest if it is unclear how the simplified formula was derived).

For example, for the body of the method `fac`, the following information is derived:

Pre-State Knowledge

`old n ≥ 0 ∧ factorial(old n) ≤ Base.MAXINT`

This is a copy of the method precondition.

Effects

`executes: false, continues: false, breaks: false, returns: true`
`variables: -; exceptions:-`

The method ends with a `return` statement and does not modify any global variable (all modified variables are local to the method).

Transition Relation

`(∃ i ∈ Base.int: i = old n + 1 ∧ 1 ≤ i ∧ value@next = factorial(i - 1)) ∧ old n ≥ 0`
`∧`
`factorial(old n) ≤ Base.MAXINT ∧ returns@next`

The core of this relation is the existential formula from which the formula `value@next = factorial(old n)` can be derived, i.e., the result of the function is the factorial of parameter `n` (the term `value@next` denotes the returned value in the post-state of the method execution). The remaining information drawn from the relation is that the original value of `n` must not

be negative ($\text{old } n \geq 0$), that the factorial of this value must not exceed the limit of type `int` ($\text{factorial}(\text{old } n) \leq \text{Base.MAX_INT}$), and that the command always terminates with the execution of a `return` statement (`returns@next`).

Termination Condition

`executes@now` \Rightarrow `old` $n \geq 0$

The formula states that the method body terminates, if the original value of n is not negative (or that a predecessor command prevents the execution of the command, i.e., the pre-state of the command execution is not an “executing” one such that `executes@now` does not hold). This information is deduced from the fact that the value of the loop’s termination term must be initially non-negative.

By selecting the body of the while loop, the following information is displayed:

Pre-State Knowledge

`old` $i \leq \text{old } n \wedge \text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX_INT} \wedge 1 \leq \text{old } i$
 \wedge
`old` $i \leq \text{old } n + 1 \wedge \text{old } p = \text{factorial}(\text{old } i - 1)$

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: p, i ; **exceptions:** -

Transition Relation

`var` $i = \text{old } i + 1 \wedge \text{var } p = \text{old } p \cdot \text{old } i$

We see from the pre-state knowledge that the body is only executed for $1 \leq \text{old } i \leq \text{old } n$ and that `old` $p = \text{factorial}(\text{old } i - 1)$; from the effects and the transition relation we can determine that the only effect of the body is to increment i and multiply p with (the old value of) i .

Selecting the whole loop, gives the following information:

Pre-State Knowledge

$$\text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge \text{old } i = 1 \wedge \text{old } p = 1$$
Precondition

$$\begin{aligned} & \text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{old } i \wedge \text{old } i \leq \text{old } n + 1 \\ & \wedge \\ & \text{old } p = \text{factorial}(\text{old } i - 1) \end{aligned}$$
Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: p, i ; **exceptions:** -

Transition Relation

$$\begin{aligned} & \text{var } i = \text{old } n + 1 \wedge \text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{var } i \\ & \wedge \\ & \text{var } p = \text{factorial}(\text{var } i - 1) \end{aligned}$$
Termination Condition

$$\text{executes@now} \Rightarrow \text{old } n - \text{old } i \geq -1$$

We see from the pre-state knowledge that the loop is executed with $\text{old } n \geq 0$ and $\text{old } i = 1$; from this and the termination condition $\text{old } n - \text{old } i \geq -1$, the termination of the loop can be established. The transition relation also tells us that $\text{var } i = \text{old } n + 1$, i.e., that the loop terminates with the value of i being $n + 1$ (since from the effects we know that n is not changed); from this and $\text{var } p = \text{factorial}(\text{var } i - 1)$, we know that the loop terminates with p being $\text{factorial}(n)$.

We may also investigate the effect of imposing additional constraints on certain states. For example, we may select with the radio button the body of the while loop and enter into the textbox as a “Prestate” condition

$$\text{VAR } i=2$$

After pressing the “Submit” button, we get the view depicted in Figure 4.2.

The green marker indicates for which statement the condition has been fixed that is indicated at the bottom of the left pane. By moving the mouse over the markers we may investigate the effects of this constraint for selected statement. For instance, when selecting

$$p = p * i;$$

we get the information

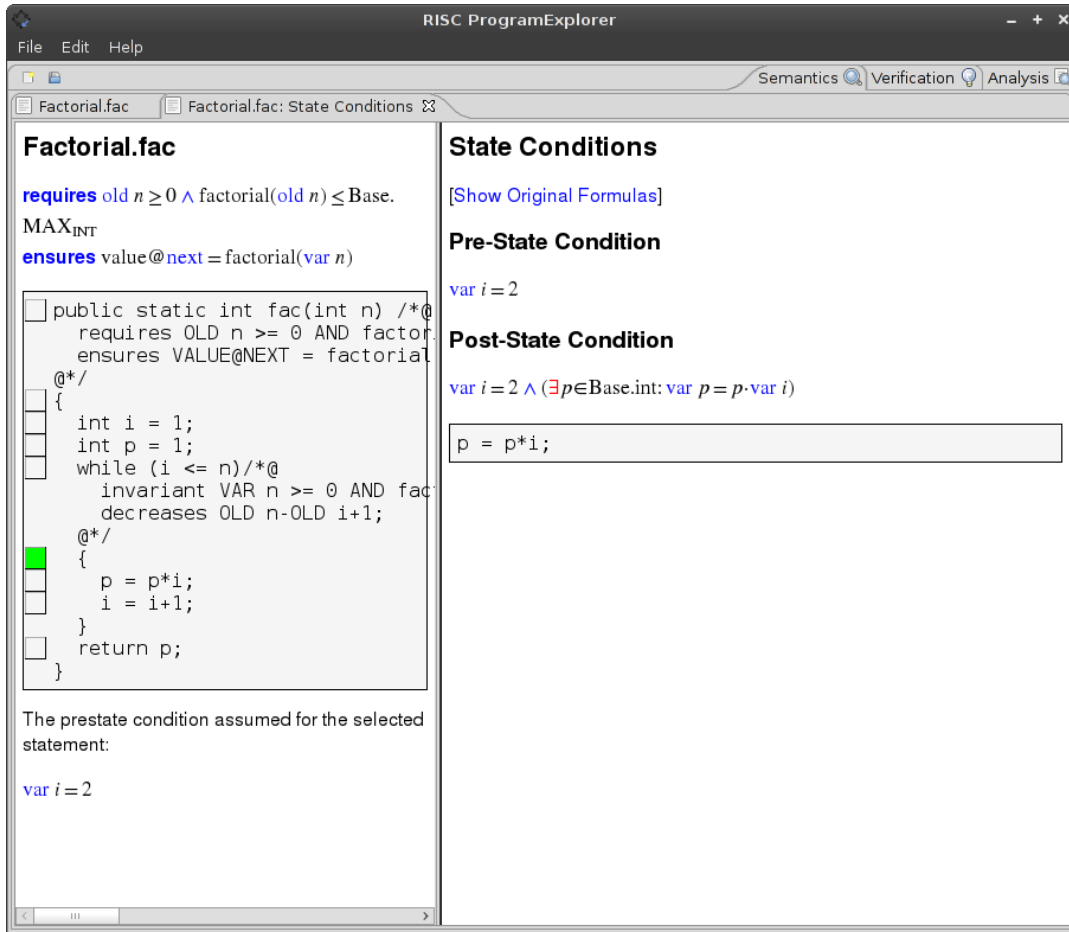


Figure 4.2.: Constraining a State

Pre-State Condition

```
var i = 2
```

Post-State Condition

```
var i = 2 ∧ (∃ p ∈ Base.int: var p = p · var i)
```

When selecting

```
i = i + 1;
```

we get

Pre-State Condition

```
var i = 2 ∧ (∃ p ∈ Base.int: var p = p · var i)
```

Post-State Condition

```
(∃ p ∈ Base.int: var p = p · 2) ∧ var i = 3
```

i.e., we know that after the execution of the loop body p is even and i equals 3.

Method Verification

After the investigation of the method semantics (which has not given any indication of a problem so far), we may turn to the actual verification of the method. We return to the “Analysis” view and investigate the task tree of the method which is displayed in Figure 4.3 (some of the tasks labeled in violet may also be labeled red, if the corresponding proof has not yet been performed). Open tasks are indicated by red task labels.

The solution of a task proceeds by

1. translating the task to a “state proving problem” in the language of the RISC ProgramExplorer (such a problem may include built-in state predicates that are not part of classical logic), then
2. translating the state proving problem into a “classical proving problem” in the language of the RISC ProgramExplorer (the built-in state predicates have been translated to defined predicates), and finally
3. translating the classical proving problem into a “ProofNavigator problem”, i.e. a problem in the language of the RISC ProofNavigator (which is slightly different from that of the RISC ProgramExplorer).

By right-clicking the symbol of an open task, a menu appears that allows to print the corresponding task translations in a linear format (this is mainly useful to investigate how the final proof problem was generated from the original task).

For solving the proof problem associated to an open task, there exist three options:

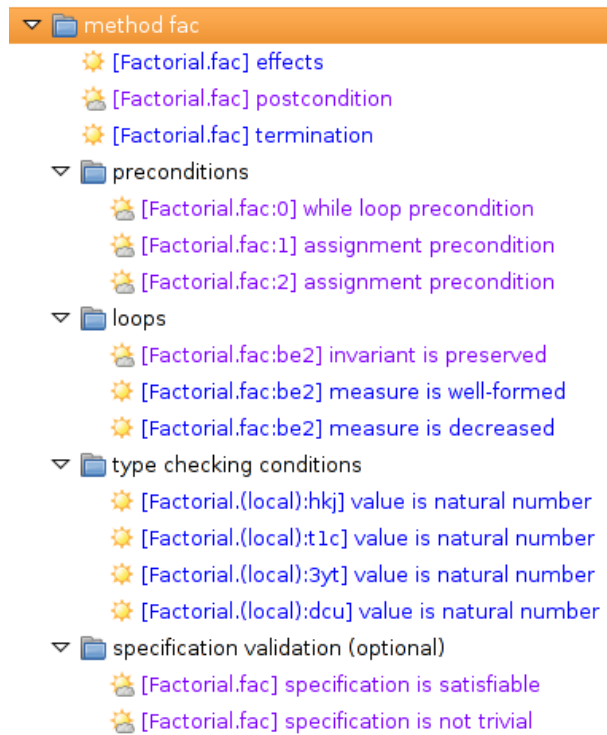



Figure 4.3.: Verification Tasks

1. For certain tasks, the built-in validity checker is automatically applied to the problem; if it is successful, the task label turns blue immediately.
2. Otherwise (the validity checker does not succeed or is not even attempted due to minor chances of success), a proof is required. Such a proof is started by right-clicking a task symbol and selecting “Execute Task”. Then the automatic `scatter` strategy is applied; if it is successful, the task label turns blue.
3. Otherwise (the `scatter` strategy does not succeed), an interactive proof is started. The RISC Program Explorer switches to the “Verification” view which is essentially the interface of the RISC ProofNavigator. This view is left by pressing the button *Quit Proof*  (after confirmation); if the proof has been completed, the task label turns blue, otherwise it stays red.

In the last two cases, a proof is generated and stored on disk. If the RISC Program Explorer is terminated and restarted, the labels of the corresponding tasks are displayed in violet color. On performing “Execute Task”, the stored proof is replayed and the label turns blue again.

In detail, the following tasks need to be performed:

Effects This is the task to show that the method only modifies those global variables that are indicated in the `modifies` clause of the method header. This task can be automatically performed by the implemented calculus¹.

Postcondition This is the task to show the *partial correctness* of the method, i.e. that the method ensures the postcondition indicated in the `ensures` clause of the method header. No attempt is made to apply the validity checker; there is always a proof required.

Termination This is the task to extend the partial correctness of the method to its *total correctness*, i.e. to show that the method also terminates. It is first attempted to solve the task by the validity checker before a proof is requested.

Measure For a recursive method, this is the task to show that the method measure is well-formed, i.e. that the method precondition implies that the method measure is not negative (the fact that the measure is decreased by a recursive call is proved as part of the caller method’s termination condition).

Preconditions Each statement (such as an assignment) may be only executed, if the statement’s precondition is satisfied (e.g. that the evaluation of an expression that occurs in the statement does not yield an arithmetic overflow). For every statement with a non-trivial precondition, a task is generated to show that the precondition is ensured when the statement is executed. It is first attempted to solve the task by the validity checker before a proof is requested.

Loops For each loop (`for` or `while` loop), several tasks are generated:

¹However, iff a `modifies` clause lists some object variable, the task of verifying that the other variables of the object are not modified, is added to the verification of the method’s postcondition

Invariant is preserved This is the task to show that the invariant is preserved by the execution of the loop body (the fact that the invariant initially holds is added to the loop's precondition). No attempt is made to apply the validity checker; there is always a proof required.

Measure is well-formed This is the task to show that the value of the termination term does not become negative² by any execution of the loop body (the fact that the value of the termination term is initially not negative is shown as part of the enclosing method's termination condition). The validity checker is applied before a proof is requested.

Measure is decreased This is the task to show the value of the termination term is decreased by every execution of the loop body³. The validity checker is applied before a proof is requested.

Type-checking Conditions Most aspects of a formula's type correctness can be verified by a static calculus, some aspects (such that in a particular occurrence of a term x/y the variable y is not negative) give rise to verification tasks. Most of these tasks can be performed by the validity checker; in some cases, however, a proof is requested.

Specification Validation (Optional) The software also generates two conditions whose proofs are not mandatory but which may/should be performed before proceeding with the rest of the verification.

- It should be shown that a “specification is satisfiable”, i.e., that for every pre-state value that satisfies the method's precondition there exists a post-state value that satisfies the postcondition. If a specification is not satisfiable, every attempt to verify partial correctness is a priori doomed.
- It should be shown that a “specification is not trivial”, i.e., that for every pre-state value that satisfies the method's precondition there exists a post-state value that *does not* satisfy the postcondition. If a specification is trivial, it indicates that every implementation is legal, which is a strong hint that the specification is flawed.

No attempt is made to apply the validity checker; there is always a proof requested.

The tasks are partially interdependent, e.g., the proof of partial correctness (performed in the “postcondition” task) depends of the correctness of the “preconditions” and of the “invariants”. However, the tasks can be performed in any order, e.g. it is recommended to first solve the “postcondition” task (under the assumption that the invariants hold) and only afterward solve the “invariants” tasks (which are typically more difficult). In this way, no time is wasted on proving the correctness of an invariant that is not strong enough to show the partial correctness of a method.

In the following, we investigate the tasks for the method `fac` in some more detail.

² A termination term is either a single integer or a vector of integers; in the latter case, all components must not be negative.

³ If the termination term is a vector of integers, some component i must be decreased while all prior components $0, \dots, i - 1$ must stay the same.

Postcondition For a method with precondition p and state relation r , to show that the postcondition q is satisfied amounts to proving

$$p \wedge r \Rightarrow q$$

(logically equivalent to $r \Rightarrow (p \Rightarrow q)$). Here p and q are part of the method header (`requires` p `ensures` q) and r is shown in the *Semantics* view of the method (“State Relation”).

In our case, we have to prove

$$\begin{aligned} & n_{\text{old}} \geq 0 \wedge \text{factorial}_0(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \\ \wedge & \\ & (\exists i \in \text{int}: i = n_{\text{old}} + 1 \wedge 1 \leq i \wedge \text{value}_{\text{now}} = \text{factorial}_0(i-1)) \\ \wedge & \\ & n_{\text{old}} \geq 0 \wedge \text{factorial}_0(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \wedge \text{returns}_{\text{now}} \\ \Rightarrow & \\ & \text{value}_{\text{now}} = \text{factorial}_0(n_{\text{old}}) \end{aligned}$$

where the first line corresponds to p , the next two lines correspond to r , and the last line corresponds to q . The proof succeeds by the automatic `scatter` strategy.

Termination The proof that the “method body terminates” is of form

$$p \wedge \neg d \Rightarrow t$$

where p represents the precondition of the method, d represents the `diverges` condition in the method’s specification (an optional condition under which the method is allowed to run forever) and t is the loop body’s termination condition (as depicted in the “Semantics” view of the method).

In our case, the termination condition is essentially `old n ≥ 0` which is part of the pre-condition; the corresponding task is solved by the validity checker.

Preconditions For a command with pre-state knowledge k and pre-condition c , to show that the pre-condition is met amounts to proving

$$k \Rightarrow c$$

where both k and c are displayed in the *Semantics* view of the method (“Pre-State Knowledge” and “Precondition”).

In our example, there are three precondition tasks, one for the loop (to show that the loop invariant initially holds), the other two for the assignments and in the loop body (to show that no arithmetic overflows occur). The first proof of

$$\begin{aligned}
& n_{\text{old}} \geq 0 \wedge \text{factorial}_0(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \wedge i_{\text{old}} = 1 \wedge p_{\text{old}} = 1 \\
& \wedge \\
& \text{executes_next_} \\
\Rightarrow & \\
& n_{\text{old}} \geq 0 \wedge \text{factorial}_0(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \wedge 1 \leq i_{\text{old}} \wedge i_{\text{old}} \leq n_{\text{old}} + 1 \\
& \wedge \\
& p_{\text{old}} = \text{factorial}_0(i_{\text{old}} - 1)
\end{aligned}$$

is (after triggering the proof) solved automatically. The second proof of

$$\begin{aligned}
& i_{\text{old}} \leq n_{\text{old}} \wedge n_{\text{old}} \geq 0 \wedge \text{factorial}(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \wedge 1 \leq i_{\text{old}} \\
& \wedge \\
& i_{\text{old}} \leq n_{\text{old}} + 1 \wedge p_{\text{old}} = \text{factorial}(i_{\text{old}} - 1) \wedge \text{executes_next_} \\
\Rightarrow & \\
& \text{MIN}_{\text{INT}} \leq p_{\text{old}} \cdot i_{\text{old}} \wedge p_{\text{old}} \cdot i_{\text{old}} \leq \text{MAX}_{\text{INT}}
\end{aligned}$$

(which corresponds to the assignment $p = p * i$) requires some arithmetic reasoning with the help of additional lemmas about the factorial function specified in theory *Math*; we omit it here. The third proof of

$$\begin{aligned}
& i_{\text{old}} \leq n_{\text{old}} \wedge n_{\text{old}} \geq 0 \wedge \text{factorial}_0(n_{\text{old}}) \leq \text{MAX}_{\text{INT}} \wedge 1 \leq i_{\text{old}} \\
& \wedge \\
& i_{\text{old}} \leq n_{\text{old}} + 1 \wedge p_{\text{old}} = \text{factorial}_0(i_{\text{old}} - 1) \cdot i_{\text{old}} \wedge \text{executes_next_} \\
\Rightarrow & \\
& \text{MIN}_{\text{INT}} \leq i_{\text{old}} + 1 \wedge i_{\text{old}} + 1 \leq \text{MAX}_{\text{INT}}
\end{aligned}$$

(which corresponds to the assignment $i = i + 1$) is simpler; it only depends on the theorem $\forall n \in \mathbb{N} : n > 2 \Rightarrow \text{factorial}(n) > n$; the corresponding proof tree is

```

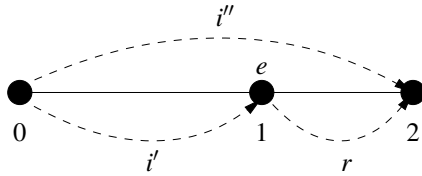
[xy]: scatter
  [upf]: proved (CVCL)
  [vpf]: case i_old > 2
    [rtb]: auto dcw
      [v5n]: proved (CVCL)
    [stb]: expand MAX_INT
      [mlv]: proved (CVCL)

```

Loops For a while loop with invariant i and a body with state relation r , the proof of the correctness of the invariant amounts to proving a formula of form

$$i' \wedge e \wedge r \Rightarrow i''$$

Here i' represents a variant of i that expresses the relationship between the initial state 0 of the loop and the state 1 before the current loop iteration, e expresses the fact that the loop condition holds at state 1, r expresses the relation between the states 1 and 2 before and after the current loop iteration, and i'' represents a variant of i that expresses the relationship between states 0 and 2:



In our example, this amounts to proving (after some simplification) the formula

$$\begin{aligned}
 & \exists i \in \text{int}: \\
 & 1 \leq i \wedge i \leq n_{\text{old}} + 1 \wedge i \leq n_{\text{old}} \wedge i_{\text{new}} = i + 1 \wedge p_{\text{new}} = \text{factorial}(i - 1) \cdot i \\
 \Rightarrow & \\
 & n_{\text{old}} < 0 \vee \text{MAX}_{\text{INT}} < \text{factorial}(n_{\text{old}}) \\
 \vee & \\
 & 1 \leq i_{\text{new}} \wedge i_{\text{new}} \leq 1 + n_{\text{old}} \wedge p_{\text{new}} = \text{factorial}(-1 + i_{\text{new}})
 \end{aligned}$$

This corresponding proof tree is

```

[xy]: scatter
  [upf]: proved (CVCL)
    [vpf]: auto 1y
      [rtb]: proved (CVCL)
  
```

i.e., the proof requires the heuristic instantiation of formula “1y” which represents the factorial axiom $\forall n \in \mathbb{N} : \text{factorial}(n + 1) = (n + 1) \cdot \text{factorial}(n)$.

The proof that the loop “measure is well-formed” is essentially to show

$$i' \wedge e \wedge r \Rightarrow t' \geq 0$$

where i' , e , and r are as indicated in the paragraph “Invariants” above and t' represents the value of the termination term after the loop iteration.

In our case, t' is denoted by $\text{old } n - \text{var } i + 1$ where e represents $\text{old } i \leq \text{old } n$ and r implies $\text{var } i = \text{old } i + 1$; the corresponding task can be solved by the validity checker.

The proof that the loop “measure is decreased” is essentially to show

$$i' \wedge e \wedge r \Rightarrow t > t'$$

where i' , e , r , and t' are indicated as above and t represents the value of the termination term before the iteration of the loop.

In our case, t is denoted by $\text{old } n - \text{old } i + 1$ with the other definitions given above; the corresponding task can be solved by the validity checker.

Type Checking Conditions The type checking conditions all result from the fact that in the method specification respectively loop invariant terms like $\text{factorial}(\text{var } n)$ respectively $\text{factorial}(\text{var } i - 1)$ occur; for each such occurrence it has to be shown that the argument of $\text{factorial} : \mathbb{N} \rightarrow \mathbb{N}$ (which is by static type checking known to be an integer number) is a natural number, i.e., not negative. All the corresponding tasks can be automatically solved by the validity checker.

Specification Validation The validation of a method specification with precondition p and postcondition q amounts to proving

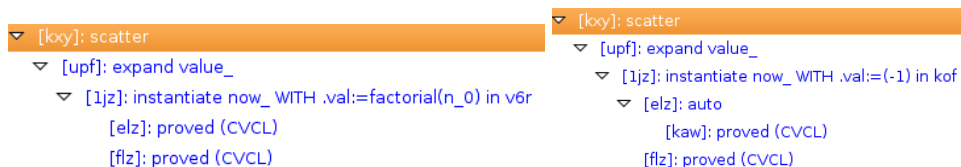
$$\begin{aligned} \forall x : p(x) &\Rightarrow \exists y : q(x, y) \\ \forall x : p(x) &\Rightarrow \exists y : \neg q(x, y) \end{aligned}$$

where x represents the pre-state of the method (containing method arguments etc) and y represents its post-state (containing the method result); we thus show that for every legal input there exists some legal output (the specification is *satisfiable*) and some output that is not legal (the specification is *not trivial*).

In case of the method `fac`, this amounts to proving the following:

$$\begin{aligned} \forall n \in \text{int}, os \in \text{STATE}: \\ n \geq 0 \wedge \text{factorial}_0(n) \leq \text{MAX_INT} \\ \Rightarrow \\ (\exists ns \in \text{STATE}: \text{value_}(ns) = \text{factorial}_0(n)) \\ \forall n \in \text{int}, os \in \text{STATE}: \\ n \geq 0 \wedge \text{factorial}_0(n) \leq \text{MAX_INT} \\ \Rightarrow \\ (\exists ns \in \text{STATE}: \text{value_}(ns) \neq \text{factorial}_0(n)) \end{aligned}$$

The corresponding proof trees are as follows:



The proof for satisfiability amounts to instantiating, for given method argument n_0 , the existential formula with a poststate derived from the prestate “now_” by setting the return value to $\text{factorial}(n_0)$; in the proof of non-triviality, the return value is set to -1 (a state, i.e. a value of type “STATE”, is a record whose field *val* denotes the state’s return value, if there is any).

4.2. Recursive Computation of Factorials

We revisit the computation of factorial numbers by discussing a directly recursive solution and an indirectly recursive solution.

The directly recursive solution implemented by a method `fac0` is straight-forward:

```
public static int fac0(int n) /*@
  requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
  ensures VALUE@NEXT = factorial(VAR n);
  decreases VAR n;
  @*/
```

```

{
  if (n <= 0) return 1;
  int n0 = fac0(n-1);
  return n*n0;
}

```

The specification is extended by a recursion measure, i.e., a term whose value must be decreased by every recursive invocation of that method.

If we switch to the “Semantics” view and select the method call `int n0 = fac0(n-1)`, the following semantics is displayed:

Pre-State Knowledge

```

if old n ≤ 0 then
  old n ≥ 0 ∧ factorial(old n) ≤ Base.MAXINT ∧ returns@now ∧ value@now = 1
else
  old n ≥ 0 ∧ factorial(old n) ≤ Base.MAXINT ∧ executes@now
endif

```

Precondition

```

Base.MININT + 1 ≤ old n ∧ old n ≤ Base.MAXINT + 1
∧
(∃ n ∈ Base.int: n + 1 = old n ∧ n ≥ 0 ∧ factorial(n) ≤ Base.MAXINT)

```

Effects

```

executes: true, continues: false, breaks: false, returns: false
variables: n0; exceptions:-

```

Transition Relation

```

∃ s2 ∈ S(Base.int):
  value@s2 = factorial(old n - 1) ∧ next ~ s2 ∧ var n0 = value@s2

```

Termination Condition

```

executes@now ⇒ old n < measure + 1

```

The transition relation of that call is derived from the method’s postcondition: it says that the poststate s_2 of the method execution ($\text{next} \sim s_2$ indicates that this state has the same variable values as the poststate `next` of the method call) returns the value of $\text{factorial}(n - 1)$ which is stored in the variable n_0 (which is the only visible variable that is modified by the statement). Since every statement is only executed in a prestate `now` with property `executes@now`, the pre-state knowledge tells us that n is positive and $\text{factorial}(n)$ is in the range of type `int`, as is required by the precondition of the method call.

From the transition relation of the method call, the transition relation of the whole method body can be derived:

Transition Relation

```

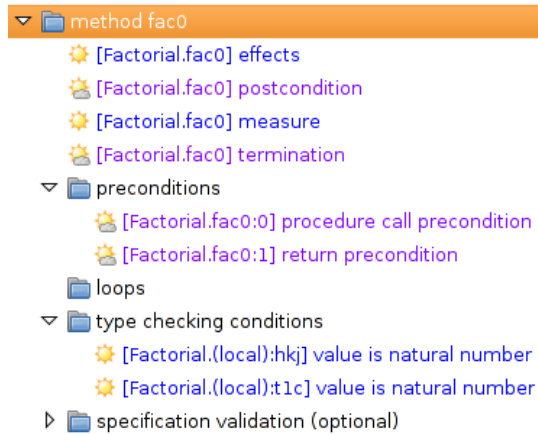
if old  $n \leq 0$  then
   $\exists \text{state}_0 \in S(\text{Base.int})$ :
    ( if executes@state0 then
      returns@state0  $\wedge$  value@state0 = 1
     $\wedge$ 
      ( $\exists \text{state} \in S(\text{Base.int}), \text{s2} \in S(\text{Base.int})$ :
        value@s2 = factorial(old  $n - 1$ )  $\wedge$  state  $\sim$  s2  $\wedge$  executes@state
       $\wedge$ 
         $\neg$  returns@state  $\wedge$  value@next = old  $n \cdot$  value@s2)
     $\wedge$ 
      returns@next
    endif )
else
  returns@state0  $\wedge$  value@state0 = 1  $\wedge$  next = state0
endif )
else
  ( $\exists \text{state} \in S(\text{Base.int}), \text{s2} \in S(\text{Base.int})$ :
    value@s2 = factorial(old  $n - 1$ )  $\wedge$  state  $\sim$  s2  $\wedge$  executes@state
   $\wedge$ 
     $\neg$  returns@state  $\wedge$  value@next = old  $n \cdot$  value@s2)
   $\wedge$ 
    returns@next
endif

```

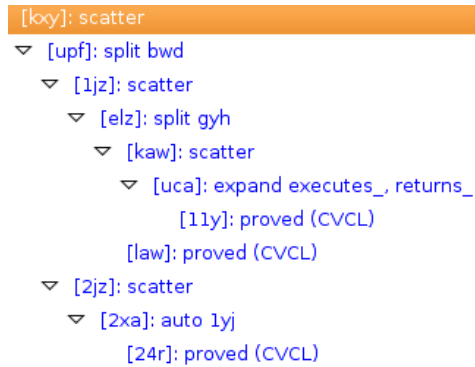
We can clearly see two cases depending on the truth value of $\text{old } n \leq 0$: if this is false, the second branch tells us that the method terminates by returning $\text{old } n \cdot \text{factorial}(\text{old } n - 1)$; if this is true the first branch tells⁴ that the return value of the method is 1.

⁴Since executes@state_0 and returns@state_0 are contradictory, the inner conditional formula can be reduced to its second branch; here the formula is not yet sufficiently simplified by the RISC ProgramExplorer.

In the “Analysis” view, the following non-trivial verification tasks are displayed:



Among these, only the task for the postcondition and the preconditions require major human interaction. As for the proof of the postcondition, by some human guidance

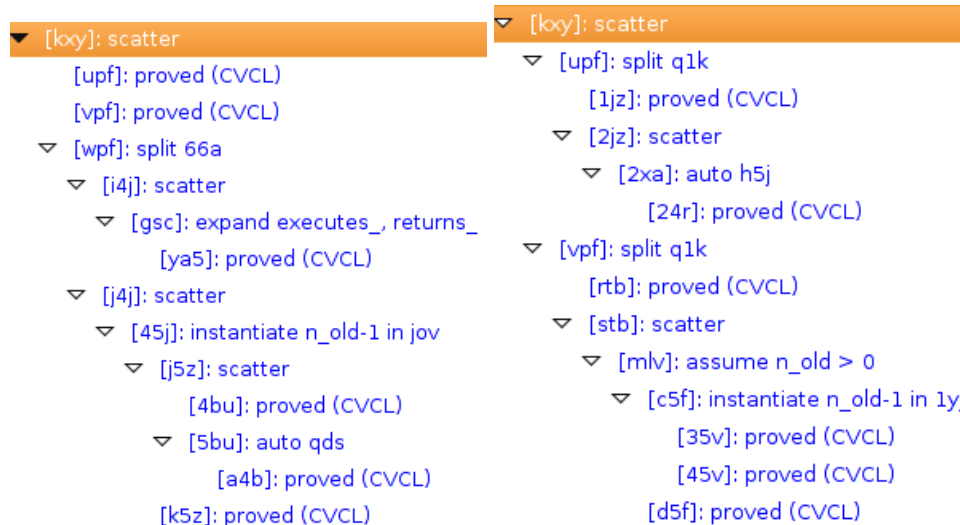


we reduce the proof to the only interesting subsituation:

1yj	$\forall n \in \mathbb{N}: \text{factorial}(n+1) = (n+1) \cdot \text{factorial}(n)$
d6p	$\text{factorial}(1) = 1$
tht	$\text{factorial}(2) = 2$
dcw	$\forall n \in \mathbb{N}: n > 2 \Rightarrow \text{factorial}(n) > n$
qds	$\forall n \in \mathbb{N}, m \in \mathbb{N}: n \geq m \Rightarrow \text{factorial}(n) \geq \text{factorial}(m)$
ind	$\forall n \in \mathbb{N}, m \in \mathbb{N}: n > m \wedge n \geq 2 \Rightarrow \text{factorial}(n) > \text{factorial}(m)$
h5j	$\forall n \in \mathbb{N}, m \in \mathbb{N}: n \cdot m \geq 0$
xiv	$\text{factorial}(n_{\text{old}}) \leq \text{MAX}_{\text{INT}}$
moq	$0 < n_{\text{old}}$
hnd	$\text{value_}(s2_0) = \text{factorial}_0(n_{\text{old}} - 1)$
6sx	$\text{similar_}(state_0, s2_0)$
isi	$\text{executes_}(state_0)$
mtq	$\neg \text{returns_}(state_0)$
ygu	$\text{value_}(now_)= n_{\text{old}} \cdot \text{value_}(s2_0)$
s4l	$\text{returns_}(now_)$
q1a	$\text{value_}(now_)= \text{factorial}_0(n_{\text{old}})$

Here it has to be shown (by appropriate instantiation of formula “1yj” derived from axiom `fac_ax2` in theory *Math*) that $\text{factorial}(n_{\text{old}}) = n_{\text{old}} \cdot \text{factorial}(n_{\text{old}} - 1)$.

The proof of the preconditions essentially have to show that the argument of the recursive factorial call and the return value are in the range of type `int`; the detailed proofs are not difficult but a a bit tedious:



The RISC ProgramExplorer also supports arbitrary mutual recursion. As an example, take the following computation of the factorial numbers performed by two methods `fac1` and `fac2` which call each other:

```
public static int fac1(int n) /*@
  requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
  ensures VALUE@NEXT = factorial(VAR n);
  decreases VAR n;
  @*/
{
  if (n <= 0) return 1;
  int n0 = fac2(n-1);
  return n*n0;
}

public static int fac2(int n) /*@
  requires VAR n >= 0 AND factorial(VAR n) <= Base.MAX_INT;
  ensures VALUE@NEXT = factorial(VAR n);
  decreases VAR n;
  @*/
{
  if (n <= 0) return 1;
  int n0 = fac1(n-1);
  return n*n0;
}
```

The methods have the same state relations and verification tasks as the directly recursive method `fac0`; we thus omit the details.

4.3. Searching in Arrays

Our next example discusses the semantics and verification of a method `search` which implements linear search of a given integer array a for a key x ; the method returns the smallest position of an occurrence of x in a , respectively -1 , if x does not occur in a .

```
public class Searching
{
  public static int search(int[] a, int x) /*@
    requires ...;
    ensures ...;
  @*/
  {
    int n = a.length;
    int r = -1;
    int i = 0;
    while (i < n && r == -1) /*@
      invariant ...;
      decreases ...;
    @*/
    {
      if (a[i] == x)
        r = i;
      else
        i = i+1;
    }
    return r;
  }
}
```

The specification of the method is as follows:

```
requires NOT (VAR a).null;
ensures
  LET result = VALUE@NEXT, n = (VAR a).length IN
  IF result = -1 THEN
    FORALL(i: INT): 0 <= i AND i < n =>
      (VAR a).value[i] /= VAR x
  ELSE
    0 <= result AND result < n AND
    (FORALL(i: INT): 0 <= i AND i < result =>
      (VAR a).value[i] /= VAR x) AND
      (VAR a).value[result] = VAR x
  ENDIF;
```

The program array a is represented by a mathematical variable $\text{var } a$ with the following type *IntArray* (declared in theory *Base*):

```

int: TYPE = [MIN_INT..MAX_INT];
nat: TYPE = [0..MAX_INT];
IntArray: TYPE =
  [#value: ARRAY int OF int, length: nat, null: BOOLEAN#];

```

The precondition states that a must not be the null pointer. The postcondition describes the two possible method results using the terms $\text{var } a.\text{length}$ to refer to the number of elements in a and $\text{var } a.\text{value}[i]$ to refer to the value at position i .

The loop is correspondingly annotated with the following invariant and termination term:

```

invariant NOT (VAR a).null AND VAR n = (VAR a).length
  AND 0 <= VAR i AND VAR i <= VAR n
  AND (FORALL(i: INT): 0 <= i AND i < VAR i =>
    (VAR a).value[i] /= VAR x)
  AND (VAR r = -1 OR (VAR r = VAR i AND VAR i < VAR n AND
    (VAR a).value[VAR r] = VAR x));
decreases IF VAR r = -1 THEN VAR n - VAR i ELSE 0 ENDIF;

```

The termination term is defined by two cases, since if x is found in a , i is not incremented but r is set to -1 .

From the specified method, the following information is displayed in the “Semantics” view of method `linsearch`:

Pre-State Knowledge

$\neg \text{old } a.\text{null}$

Effects

executes: false, **continues:** false, **breaks:** false, **returns:** true
variables: -; **exceptions:** -

Transition Relation

$$\begin{aligned}
& (\exists in \in \text{Base.int}, n \in \text{Base.int}: \\
& \quad n = \text{old } a.\text{length} \wedge (in \geq n \vee \text{value@next} \neq -1) \wedge 0 \leq in \wedge in \leq n \\
& \quad \wedge \\
& \quad (\forall i \in \mathbb{Z}: 0 \leq i \wedge i < in \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x) \\
& \quad \wedge \\
& \quad (\text{value@next} = -1 \\
& \quad \vee \\
& \quad \text{value@next} = in \wedge in < n \wedge \text{old } a.\text{value}[\text{value@next}] = \text{old } x)) \wedge \neg \text{old } a.\text{null} \\
& \quad \wedge \\
& \quad \text{returns@next}
\end{aligned}$$

Termination Condition

$\text{executes@now} \Rightarrow \text{old } a.\text{length} \geq 0$

We see that the body is executed in a state when a is not `null` and that the termination condition is ensured since the length of a is always non-negative; from the transition relation, we can deduce that up to a certain position $0 \leq in \leq \text{old } a.\text{length}$ (the value of i at the termination of `fac`) the key x does not occur in a and that one of the two cases occurs:

1. the return value is -1 and $in = n$, or
2. the return value equals in and a holds x at that position.

From this the partial correctness of the method is pretty self-evident.

The semantics of the `while` loop is depicted as follows:

Pre-State Knowledge

$$\neg \text{old } a.\text{null} \wedge \text{old } n = \text{old } a.\text{length} \wedge \text{old } r = -1 \wedge \text{old } i = 0$$

Precondition

$$\begin{aligned} & \text{Base.MIN}_{\text{INT}} \leq -1 \wedge -1 \leq \text{Base.MAX}_{\text{INT}} \wedge \neg \text{old } a.\text{null} \wedge \text{old } n = \text{old } a.\text{length} \wedge 0 \leq \text{old } i \\ & \wedge \\ & \text{old } i \leq \text{old } n \wedge (\forall i \in \mathbb{Z}: 0 \leq i \wedge i < \text{old } i \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x) \\ & \wedge \\ & (\text{old } r = -1 \vee \text{old } r = \text{old } i \wedge \text{old } i < \text{old } n \wedge \text{old } a.\text{value}[\text{old } r] = \text{old } x) \end{aligned}$$

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: r, i ; **exceptions:** -

Transition Relation

$$\begin{aligned} & (\text{var } i \geq \text{old } n \vee \text{var } r \neq -1) \wedge \neg \text{old } a.\text{null} \wedge \text{old } n = \text{old } a.\text{length} \wedge 0 \leq \text{var } i \\ & \wedge \\ & \text{var } i \leq \text{old } n \wedge (\forall i \in \mathbb{Z}: 0 \leq i \wedge i < \text{var } i \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x) \\ & \wedge \\ & (\text{var } r = -1 \vee \text{var } r = \text{var } i \wedge \text{var } i < \text{old } n \wedge \text{old } a.\text{value}[\text{var } r] = \text{old } x) \end{aligned}$$

Termination Condition

$$\text{executes@now} \Rightarrow \text{if old } r = -1 \text{ then old } n - \text{old } i \text{ else } 0 \text{ endif} \geq 0$$

In the pre-state knowledge, we can derive the values of n , r , and i when the loop starts execution. From the effect clause, we can deduce that the loop is only terminated by the loop expression and that the only modified variables are r and i . The transition relation gives essentially the same information as for the method body discussed above. The termination condition states that the termination term must be initially non-negative; from the pre-state knowledge, we know that this is the case because n equals the length of a and i is 0.

The semantics of the loop body is depicted as follows:

Pre-State Knowledge

```

old  $i < \text{old } n \wedge \text{old } r = -1 \wedge \neg \text{old } a.\text{null} \wedge \text{old } n = \text{old } a.\text{length} \wedge 0 \leq \text{old } i$ 
^
old  $i \leq \text{old } n \wedge (\forall i \in \mathbb{Z}: 0 \leq i \wedge i < \text{old } i \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x)$ 
^
(old  $r = -1 \vee \text{old } r = \text{old } i \wedge \text{old } i < \text{old } n \wedge \text{old } a.\text{value}[\text{old } r] = \text{old } x)$ 

```

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: r, i ; **exceptions:**-

Transition Relation

```

if old  $a.\text{value}[\text{old } i] = \text{old } x$  then
  var  $r = \text{old } i \wedge \text{var } i = \text{old } i$ 
else
  var  $i = \text{old } i + 1 \wedge \text{var } r = \text{old } r$ 
endif

```

We see that only the variables r and i are modified as described by the transition relation. The semantics of the body statement $i=i+1$

Pre-State Knowledge

```

old  $i < \text{old } n \wedge \text{old } r = -1 \wedge \neg \text{old } a.\text{null} \wedge \text{old } n = \text{old } a.\text{length} \wedge 0 \leq \text{old } i$ 
^
old  $i \leq \text{old } n \wedge (\forall i \in \mathbb{Z}: 0 \leq i \wedge i < \text{old } i \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x)$ 
^
(old  $r = -1 \vee \text{old } r = \text{old } i \wedge \text{old } i < \text{old } n \wedge \text{old } a.\text{value}[\text{old } r] = \text{old } x)$ 

```

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: r, i ; **exceptions:**-

Transition Relation

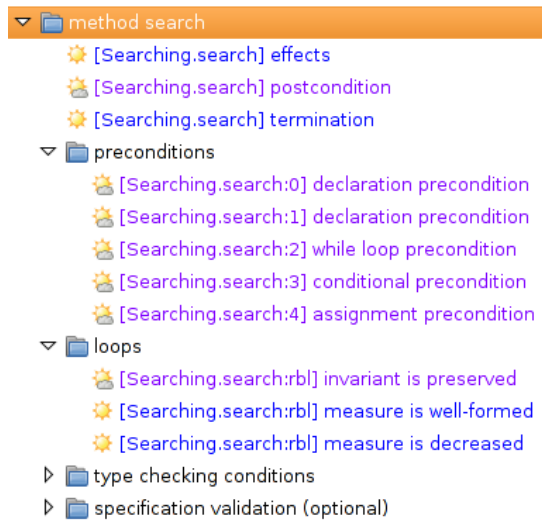
```

if old  $a.\text{value}[\text{old } i] = \text{old } x$  then
  var  $r = \text{old } i \wedge \text{var } i = \text{old } i$ 
else
  var  $i = \text{old } i + 1 \wedge \text{var } r = \text{old } r$ 
endif

```

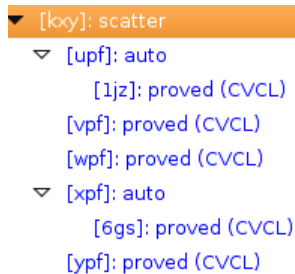
has a precondition that restricts the value of $\text{old } i + 1$ to the domain of type `int`; fortunately this can be established from the pre-state knowledge $\text{old } i \geq 0$ and $\text{old } i < \text{old } n = \text{old } a.\text{length}$.

The “Analysis” view depicts the following (non-trivial) verification tasks:

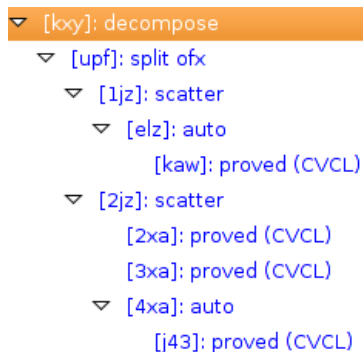


We see that the verification of the method’s effects, its termination, and that the loop measure is well-formed and decreased by every iteration have been performed automatically by the validity checker. The verification of the individual preconditions can be performed by the automatic `scatter` strategy respectively built-in simplification of the RISC ProofNavigator. The only two tasks that require interactive proofs are related to the method’s partial correctness and the preservation of the invariant.

The verification of partial correctness amounts to proving that the method’s state relation depicted above implies the post-condition; the corresponding proof proceeds with minor help:



Also the verification of the loop invariant requires only minor guidance to split the goal appropriately towards those two proof branches that can be solved by heuristic instantiation:



We see that with slightly more aggressive automation of proof search, also these proofs could have been performed fully automatically.

4.4. Program States and Control Flow Interruptions

The RISC ProgramExplorer also supports the arbitrary use of the commands `continue`, `break`, `return`, `throw` that interrupt the control flow by jumping to the next loop iteration, to the command succeeding the loop, to the caller of a method, or to the handler of an exception. The specification language takes this feature into account by supporting a type `STATE (T)` which denotes the state before/after execution of a command inside a method that returns a value of type `T` (i.e., `T` denotes the mathematical counterpart of the corresponding program type); the type `STATE` denotes the corresponding type for a method of type `void`. In a specification, the constants `NOW` and `NEXT` are of this type; `NOW` denotes the pre-state of the command and `NEXT` denotes its post-state. Specifications may be expressed with the following state functions (where `s` denotes a state):

VALUE@s which denotes the state's return value (if `s` results from the execution of the statement `return value`);

MESSAGE@s which denotes the state's exception message (if `s` results from the execution of the statement `throw new exception(message)`).

The most frequently occurring term of this kind is `VALUE@NEXT` which describes a function's return value.

Furthermore, we have the following state predicates (which are exhaustive and disjoint; i.e. a state is exactly in one of the denoted modes):

EXECUTES@s `s` is *executing*, i.e. `s` results from the execution of a command that does not interrupt the normal control flow.

CONTINUES@s `s` is *continuing*, i.e. `s` results from the execution of command `continue`; execution continues with the next iteration of the enclosing loop.

BREAKS@s `s` is *breaking*, i.e. `s` results from the execution of command `break`; execution continues with the command after the enclosing loop;

RETURNS@s `s` is *returning*, i.e. `s` results from the execution of command `return`; execution returns to the caller of the current method.

THROWS@s `s` is *throwing*, i.e. `s` results from the execution of command `throw`; execution jumps to the currently active exception handler (if any). `THROWS (E) @s` is a special version of the predicate that is only true if the exception thrown is of type (class) `E`.

Finally, we have the state predicate `s1 ~ s2` which denotes that `s1` and `s2` (which may be of different state types) are equal up to the state's return value.

After translation of a verification condition from state logic to classical logic, a state is represented by a record type


```
STATE: TYPE =
  [#mode: INT, val: ..., exception: INT, message: ...#];
```

where an integer field *mode* holds a numerical code for the execution mode, *val* holds the state's return value, *exception* holds a code for the exception's type, and *message* holds the exception's message. The state functions above are translated to corresponding functions *value_* and *message_* that extract the corresponding information from the record. The state predicates above are translated to corresponding predicates *executes_*, *continues_*, *breaks_*, *returns_*, *throws_*, and *throwsException_* that test the record's *mode* field.

As a simple example, take the following method `isSorted` which returns TRUE if and only if a given integer array is sorted in ascending order:

```
/*@
theory uses Base {
  int: TYPE = Base.int;
  intArray: TYPE = Base.IntArray;
  isSorted: PREDICATE(intArray, int) =
    PRED(a: intArray, n: int):
      FORALL(i: INT): 1 <= i AND i < n =>
        a.value[i-1] <= a.value[i];
}
@*/
class Arrays
{
  public static boolean isSorted(int[] a) /*@
    requires NOT (VAR a).null;
    ensures (VALUE@NEXT=TRUE) <=>
      isSorted(VAR a, (VAR a).length);
  @*/
  {
    int n = a.length;
    for (int i=1; i<n; i++)
      /*@ invariant ...; decreases ...; @*/
      {
        if (a[i-1] > a[i]) return false;
      }
    return true;
  }
}
```

The program loop is adequately specified by the following invariant and termination term:

```
invariant NOT (VAR a).null AND VAR n = (VAR a).length
AND 1 <= VAR i AND (VAR n >= 1 => VAR i <= VAR n)
AND (EXECUTES@NEXT => isSorted(VAR a, VAR i))
AND (RETURNS@NEXT =>
  VALUE@NEXT = FALSE AND
  VAR i < VAR n AND
```

```
(VAR a).value[VAR i-1] > (VAR a).value[VAR i]);
decreases VAR n - VAR i + 1;
```

The invariant says that, if the loop is still executing, the array is known to be sorted up to position i ; if the loop, however, executes the `return` statement, it is with a return value `false` because at position i the order is violated.

If we switch to the “Semantics” view of the method, we can consequently investigate the semantics of `return false` (which as seen by the effects always returns)

Effects

executes: false, **continues:** false, **breaks:** false, **returns:** true
variables: -; **exceptions:** -

Transition Relation

`returns@next ∧ value@next = false`

of the loop body (which as seen by the effects may or may not return)

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** true
variables: -; **exceptions:** -

Transition Relation

```
if old a.value[old i-1] > old a.value[old i] then
  returns@next ∧ value@next = false
else
  executes@next
endif
```

and of the whole loop

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** true
variables: -; **exceptions:** -

Transition Relation

```
(∃ in ∈ Base.int:
  (executes@next ⇒ in ≥ old n) ∧ 1 ≤ in ∧ (old n ≥ 1 ⇒ in ≤ old n)
  ∧
  (executes@next ⇒ isSorted(old a, in))
  ∧
  ( returns@next
    ⇒
    value@next = false ∧ in < old n
    ∧
    old a.value[in-1] > old a.value[in]) ∧ ¬ old a.null
  ∧
  old n = old a.length
```

as indicated by the loop invariant.

The semantics of the whole method body is

Transition Relation

```

 $\exists \text{state}_1 \in S(B):$ 
  ( if executes@state1 then
    ( $\exists n \in \text{Base.int}, in \in \text{Base.int}:$ 
       $n = \text{old } a.\text{length} \wedge in \geq n \wedge 1 \leq in \wedge (n \geq 1 \Rightarrow in \leq n) \wedge \text{isSorted}(\text{old } a, in)$ 
       $\wedge$ 
      ( returns@state1
         $\Rightarrow$ 
        value@state1 = false  $\wedge in < n$ 
         $\wedge$ 
        old  $a.value[in-1] > \text{old } a.value[in]$ )  $\wedge \neg \text{old } a.\text{null}$ 
      )
    )
    returns@next  $\wedge$  value@next = true
  else
    ( $\exists n \in \text{Base.int}, in \in \text{Base.int}:$ 
       $n = \text{old } a.\text{length} \wedge 1 \leq in \wedge (n \geq 1 \Rightarrow in \leq n)$ 
       $\wedge$ 
      ( returns@state1
         $\Rightarrow$ 
        value@state1 = false  $\wedge in < n$ 
         $\wedge$ 
        old  $a.value[in-1] > \text{old } a.value[in]$ )  $\wedge \neg \text{old } a.\text{null}$ 
      )
    )
    returns@state1  $\wedge$  next = state1
  endif )

```

The state relation shows that there exist two cases in the second case, the return value is “false” because a violation in the sorting order has been detected at position in ; in the first case (whose description could be considerably simplified because executes@state_1 and returns@state_1 are incompatible), the result is “true” and the array is sorted up to position $in = n = \text{old } a.\text{length}$.

For method `isSorted` the usual verification tasks are generated where only the proof of the postcondition, the loop precondition, the preservation of the loop invariant, and the decrement of the loop measure require human guidance. For instance, the postcondition proof

```

[loxy]: decompose
  [upf]: split 3x6
  [1jz]: expand isSorted
    [elz]: scatter
      [kaw]: auto 3vo
        [uca]: proved (CVCL)
    [2jz]: scatter
      [2xa]: proved (CVCL)
      [3xa]: proved (CVCL)
    [4xa]: expand isSorted
      [j43]: auto hmt
        [4kg]: proved (CVCL)

```

requires the expansion of the predicate `isSorted` and the heuristic instantiation of the resulting universal formula. The proof of the loop precondition (the loop invariant hold initially)

```

[loxy]: scatter
  [upf]: proved (CVCL)
  [vpf]: proved (CVCL)
  [wvf]: proved (CVCL)
  [xpf]: proved (CVCL)
  [ypf]: expand isSorted
    [wq1]: scatter
      [izl]: proved (CVCL)
    [zpf]: expand executes_, returns_
      [n1c]: proved (CVCL)
      [1pf]: proved (CVCL)
    [2pf]: expand executes_, returns_
      [2nt]: proved (CVCL)
    [3pf]: expand executes_, returns_
      [sx2]: proved (CVCL)

```

requires the expansion of auxiliary predicates “executes_” and “returns_” (which represent in the RISC ProofNavigator the corresponding state predicates mentioned above) to show that they cannot hold for the same state. The other proofs proceed in a similar fashion.

4.5. Objects and Method Side Effects

The RISC ProgramExplorer also supports reasoning over objects as shown in the following implementation of the abstract datatype “stack”:

```

public class Stack2 /*@
  invariant
    NOT (VAR stack).null AND
    0 <= VAR n AND VAR n <= (VAR stack).length;
  @*/

```

```
{
  private int[] stack;
  private int n;

  public Stack2() /*@
    assignable this;
    ensures VAR n = 0;
  @*/
  {
    stack = new int[10];
    n = 0;
  }

  public boolean isEmpty() /*@
    ensures VALUE@NEXT = TRUE <=> VAR n = 0;
  @*/
  {
    return n == 0;
  }

  public int top() /*@
    requires VAR n > 0;
    ensures VALUE@NEXT = (VAR stack).value[VAR n-1];
  @*/
  {
    return stack[n-1];
  }

  public int pop() /*@
    requires VAR n > 0;
    assignable n;
    ensures VALUE@NEXT = (VAR stack).value[OLD n-1]
      AND VAR n = OLD n-1;
  @*/
  {
    int result = top();
    n = n-1;
    return result;
  }

  public void push(int v) /*@
    requires (VAR n = (VAR stack).length =>
      2*(VAR stack).length+1 <= Base.MAX_INT);
    assignable stack, n;
    ensures VAR n = OLD n+1
```

```

        AND (VAR stack).value[OLD n] = VAR v;
    @*/
    {
        if (n == stack.length)
            stack = resize();
        stack[n] = v;
        n = n+1;
    }

private int[] resize() /*@
    helper;
    requires
        NOT (VAR stack).null AND
        0 <= VAR n AND VAR n <= (VAR stack).length AND
        2*(VAR stack).length+1 <= Base.MAX_INT;
    ensures LET result = VALUE@NEXT IN
        NOT result.null AND
        result.length > (OLD stack).length AND
        (FORALL(i: INT): 0 <= i AND i < VAR n =>
            result.value[i] = (OLD stack).value[i]);
    @*/
    {
        int[] stack0 = new int[2*stack.length+1];
        for (int i=0; i<n; i++) /*@
            invariant
                NOT (VAR stack).null AND
                0 <= VAR n AND VAR n <= (VAR stack).length AND
                0 <= VAR i AND VAR i <= VAR n AND
                NOT (VAR stack0).null AND
                (VAR stack0).length > (OLD stack).length AND
                (FORALL(i: INT): 0 <= i AND i < VAR i =>
                    (VAR stack0).value[i] = (OLD stack).value[i]);
            decreases VAR n - VAR i;
        @*/
        stack0[i] = stack[i];
        return stack0;
    }
}

```

The main features of this program are the following:

1. The `invariant` in the class header specifies an *object invariant*, i.e. a state condition
 - a) that is added to the postcondition of every constructor and object method,
 - b) that is added to the precondition of every object method (not constructor)
provided the method is not tagged as `helper` (as is the auxiliary method `resize()`)

above⁵

2. The assignable clauses in the method headers constrain the side effects of the method by indicating which globally visible variables may be changed by the execution of the method. A specification `assignable this` indicates that every variable in the current object may be changed; a specification of `assignable var` (with object variable `var`) specifies that only the variable `var` in the current object may be changed (all other variables in the current object remain the same); a specification `assignable object.var` specifies that in object `object` only the variable `var` may be changed.

Similar specifications are allowed for class variables (in the current class or other classes using the syntax `assignable class.var`). If no assignable clause is given, the method is “pure”, i.e., does not change any object or class variable.

An object of type `Stack2` is represented by the following record type (defined in the automatically generated theory `Stack2`):

```
Stack2: TYPE =
  [#stack:
    [#value: ARRAY Base.int OF Base.int,
     length: Base.nat, null: BOOLEAN#],
   n: Base.int, null: BOOLEAN, new: INT#]
```

The record type contains one field for every object variable as well as the automatically generated fields `null` (if set to `TRUE`, the record represents the `null` value) and `new` (set to an unknown value after the construction of the object; the results of two constructor calls are thus never equal). The specification variable `this` is a value of this type.

⁵Currently this is mainly a feature to avoid repetition in method specifications; it is e.g. not yet checked whether a helper method is private as would be required for a true modular reasoning about objects.

Based on this representation, the semantics of the object method `push` is as follows:

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false

variables: this; **exceptions:-**

Transition Relation

```

if old this.n = old this.stack.length then
  ( $\exists$  state0 ∈ S, s2 ∈ S ([#value ∈ array Base.int of Base.int, length ∈ Base.nat, null
    ∈ B#]):
    var this = old this with .stack := value@s2 with .stack.value[old this.n] :=
      old v with .n := (old this.n + 1) ∧ executes@state0
  ∧
  (let
    result = value@s2
  in
    ¬ result.null ∧ result.length > old this.stack.length
  ∧
    ( $\forall i \in \mathbb{Z}$ :
      0 ≤ i ∧ i < old this.n
    ⇒
      result.value[i] = old this.stack.value[i]))
  ∧
  state0 ~ s2) ∧ ¬ old this.stack.null ∧ 0 ≤ old this.n
  ∧
  old this.n ≤ old this.stack.length
else
  var this = old this with .stack.value[old this.n] := old v with .n := (old this.n + 1)
endif

```

We see that the effect of the method is a modification of the value of `this` (of record type `Stack2`); if some object variables are not changed, a corresponding equality is provided as part of the transition relation.

For each of the object methods, as usual a couple of tasks are generated; however, most of them can be solved automatically by the validity checker or the `scatter` strategy. Also the remaining interactive proofs are very simple and require only minor human guidance.

References

- [1] ANTLR v3 Parser Generator, 2010. <http://www.antlr.org>.
- [2] Clark Barrett. CVC Lite Homepage, April 2006. New York University, NY, <http://www.cs.nyu.edu/acsys/cvcl>.
- [3] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004.
- [4] The Java Modeling Language (JML), 2010. <http://www.jmlspecs.org>.
- [5] The RISC ProofNavigator, 2010. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, <http://www.risc.jku.at/research/formal/software/ProofNavigator>.
- [6] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008. <http://www.risc.jku.at/people/schreine/papers/ProgramCalculus2008.pdf>.
- [7] Wolfgang Schreiner. Understanding Programs. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, July 2008. <http://www.risc.uni-linz.ac.at/people/schreine/papers/Understanding2008.pdf>.
- [8] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, 2009.
- [9] Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs (Extended Abstract). In Pedro Quaresma and Ralph-Johan Back, editors, *THedu'11, CTP Components for Educational Software, Workshop associated to CADE-23*, number 2011/001 in CISUC Technical Report, pages 55–59, Wroclaw, Poland, July 31, 2011. Center for Informatics and Systems, University of Coimbra, Portugal.

A. Programs as State Relations

In this chapter (whose material is based on [9]), we sketch the formal calculus on which the RISC ProgramExplorer is founded. To simplify the presentation, we do not refer to the subset of Java which is actually used in the software but to a simple command language without control flow interruptions and method calls. In this language, a command c can be formed according to the grammar

$$c ::= x = e \mid \{\text{var } x; c\} \mid \{c_1; c_2\} \\ \mid \text{if } (e) \text{ then } c \mid \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid \text{while } (e)^{f,t} c$$

where x denotes a program variable, e denotes a program expression, and a while loop is annotated by an invariant formula f and termination term t . The semantics of a command c is defined, for a given set $Store$ of possible states (store contents), by a binary relation $\llbracket c \rrbracket \subseteq Store \times Store$ that defines the possible state transitions of the command and by a set $\langle\langle c \rangle\rangle \subseteq Store$ that defines those pre-states where the command must perform a transition to some post-state; for a definition of the semantics, see [6].

In Figures A.1, A.2, and A.3, we give rules (where the terms **old** xs and **var** xs refer to the sets of values of the program variables xs in the pre-/post-state) to derive for the commands shown above the following three kinds of judgments:

- $c : \llbracket f_r \rrbracket_{g,h}^{xs}$ denotes the derivation of a state relation f_r from command c together with the set of program variables xs that may be modified by c . The derived relation is correct if the derived state-independent condition g holds, and if the derived state condition h holds on the pre-state of c . The rationale for g is to capture state-independent conditions such as the correctness of loop invariants; the purpose of h is to capture statement preconditions that prevent e.g. arithmetic overflows. These side conditions have to be proved; they are separated from the transition relation f_r to make the core of the relation better understandable.
- $c \downarrow_{g_c} f_c$ denotes the derivation of a state condition (termination condition) f_c from c ; the derived condition is correct, if the state-independent condition g_c holds. The purpose of this side condition is to capture that the initial value of a loop's termination term is a natural number.
- $\text{PRE}(c, f_q) = f_p$ and $\text{POST}(c, f_p) = f_q$ denote derivations that compute from a command c and a condition f_q on the post-state of c a corresponding condition f_p on the pre-state, respectively from c and pre-condition f_p the post-condition f_q . The corresponding rules in Figure A.3 show that these conditions can be computed directly from the transition relation of c .

$$\begin{array}{c}
\frac{c : [f]_{g,h}^{xs} \quad x \notin xs}{c : [f \wedge \text{var } x = \text{old } x]_{g,h}^{xs \cup \{x\}}} \qquad \frac{e \simeq_h t}{x = e : [\text{var } x = t]_{\text{true},h}^{\{x\}}} \\
\frac{c : [f]_{g,h}^{xs}}{\{\text{var } x; c\} : [\exists x : f]_{g, \forall x : h[x/\text{old } x]}^{xs \setminus x}} \\
\frac{c_1 : [f_1]_{g_1,h_1}^{xs} \quad c_2 : [f_2]_{g_2,h_2}^{xs} \quad \text{PRE}(c_1, h_2) = h_3}{\{c_1; c_2\} : [\exists ys : f_1[ys/\text{var } xs] \wedge f_2[ys/\text{old } xs]]_{g_1 \wedge g_2, h_1 \wedge h_3}^{xs}} \\
\frac{e \simeq_h f_e \quad c_1 : [f_1]_{g_1,h_1}^{xs}}{\text{if } (e) \text{ then } c : [\text{if } f_e \text{ then } f_1 \text{ else var } xs = \text{old } xs]_{g_1,h \wedge (f_e \Rightarrow h_1)}^{xs}} \\
\frac{e \simeq_h f_e \quad c_1 : [f_1]_{g_1,h_1}^{xs} \quad c_2 : [f_2]_{g_2,h_2}^{xs}}{\text{if } (e) \text{ then } c_1 \text{ else } c_2 : [\text{if } f_e \text{ then } f_1 \text{ else } f_2]_{g_1 \wedge g_2, h \wedge \text{if } f_e \text{ then } h_1 \text{ else } h_2}^{xs}} \\
\frac{e \simeq_h f_e \quad c : [f_c]_{g_c,h_c}^{xs}}{g \equiv \forall xs, ys, zs : f[xs/\text{old } xs, ys/\text{var } xs] \wedge f_e[ys/\text{old } xs] \wedge f_c[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow h[ys/\text{old } xs] \wedge f[xs/\text{old } xs, zs/\text{var } xs]} \\
\frac{}{\text{while } (e)^{f,t} c : [f \wedge \neg f_e \text{var } xs/\text{old } xs]_{g_c \wedge g, h \wedge f[\text{old } xs/\text{var } xs]}^{xs}}
\end{array}$$

Figure A.1.: The Transition Rules

$$\begin{array}{c}
x = e \downarrow_{\text{true}} \text{true} \quad \frac{c \downarrow_g f}{\{\text{var } x; c\} \downarrow_g \forall x : f} \quad \frac{c_1 \downarrow_{g_1} f_1 \quad c_2 \downarrow_{g_2} f_2 \quad \text{PRE}(c_1, f_2) = f_3}{\{c_1; c_2\} \downarrow_{g_1 \wedge g_2} f_1 \wedge f_3} \\
\frac{e \simeq_h f_e \quad c \downarrow_g f}{\text{if } (e) \text{ then } c \downarrow_g f_e \Rightarrow f} \quad \frac{e \simeq_h f_e \quad c_1 \downarrow_{g_1} f_1 \quad c_2 \downarrow_{g_2} f_2}{\text{if } (e) \text{ then } c_1 \text{ else } c_2 \downarrow_{g_1 \wedge g_2} \text{if } f_e \text{ then } f_1 \text{ else } f_2} \\
\frac{e \simeq_h f_e \quad c : [f_c]_{g_c,h_c}^{xs} \quad c \downarrow_{g_t} f_t}{g \equiv \forall xs, ys, zs : f[xs/\text{old } xs, ys/\text{var } xs] \wedge f_e[ys/\text{old } xs] \wedge f_c[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow g_t[ys/\text{old } xs] \wedge f_t[ys/\text{old } xs] \wedge \text{let } n = t[zs/\text{old } xs] \text{ in } n \in \mathbb{N} \wedge n < t[ys/\text{old } xs]} \\
\frac{}{\text{while } (e)^{f,t} c \downarrow_g t \in \mathbb{N}}
\end{array}$$

Figure A.2.: The Termination Rules

$$\begin{array}{c}
\frac{c : [f]_{g,h}^{xs}}{\text{PRE}(c, f_q) = \forall xs : f[xs/\text{var } xs] \Rightarrow f_q[xs/\text{old } xs]} \\
\frac{c : [f]_{g,h}^{xs}}{\text{POST}(c, f_p) = \exists xs : f_p[xs/\text{old } xs] \wedge f[xs/\text{old } xs, \text{old } xs/\text{var } xs]}
\end{array}$$

Figure A.3.: The Pre-/Postcondition Rules

The derivations make use of additional judgments $e \simeq_{f_e} f$ and $e \simeq_{f_e} t$ which translate a boolean-valued program expression e into a logic formula f and an expression e of any other type into a term t , provided that the state in which e is evaluated satisfies the condition f_e (the rules for these judgments are omitted).

One should note that the rules presented in Figures A.1 and A.2 can be applied recursively over the structure of a command; first we determine the transition/termination formula of the subcommands, then we combine the formulas to the transition/termination formula of the whole command. Along this process, the side condition h is constructed which has to be shown separately to hold in the pre-state of the command in order to verify the correctness of the translation.

A special case is the rule for `while` loops. Here the result is only determined by the invariant formula respectively termination term by which the loop is annotated; additionally, a proof obligation g is generated to verify the correctness of the loop body with respect to invariant and termination term. In a similar way, in the full programming language calls of program methods are handled: the transition relation of the method call is derived from the specification of the method; the correctness of the implementation of the method is to be established separately. We thus yield a modular approach to the derivation of transition relations and termination conditions; the size of the derived formula is independent of the sizes of the bodies of the loops executed respectively of the methods called. Furthermore, the approach gives rise to some sort of “correct by construction” approach: we may first develop loop invariants and method preconditions (and verify the correctness of programs executing the loops and calling the methods) before we implement the bodies of the loops and methods (and consequently verify the correctness of implementations).

Formally, the derivations satisfy the following soundness constraints.

Theorem 1 (Soundness) *For all $c \in \text{Command}$, $f_r, f_c, f_p, f_q, g, h \in \text{Formula}$, and for all $xs \in \mathbb{P}(\text{Variable})$, the following statements hold:*

1. *If we can derive the judgment $c : [f_r]_{g,h}^{xs}$, then we have for all $s, s' \in \text{Store}$*

$$\llbracket g \rrbracket \wedge \llbracket h \rrbracket(s) \Rightarrow (\llbracket c \rrbracket(s, s') \Rightarrow \llbracket f_r \rrbracket(s, s') \wedge \forall x \in \text{Variable} \setminus xs : \llbracket x \rrbracket(s) = \llbracket x \rrbracket(s')).$$

2. *If we can (in addition to $c : [f_r]_{g,h}^{xs}$) derive the judgment $c \downarrow_{g_c} f_c$, then we have for all $s \in \text{Store}$*

$$\llbracket g \rrbracket \wedge \llbracket g_c \rrbracket \wedge \llbracket h \rrbracket(s) \Rightarrow (\llbracket f_c \rrbracket(s) \Rightarrow \langle\langle c \rangle\rangle(s)).$$

3. *If we can (in addition to $c : [f_r]_{g,h}^{xs}$) also derive the judgment $\text{PRE}(c, f_q) = f_p$ or the judgment $\text{POST}(c, f_p) = f_q$, then we have for all $s, s' \in \text{Store}$*

$$\llbracket g \rrbracket \wedge \llbracket h \rrbracket(s) \Rightarrow (\llbracket f_p \rrbracket(s) \wedge \llbracket f_r \rrbracket(s, s') \Rightarrow \llbracket f_q \rrbracket(s')).$$

The semantics $\llbracket f \rrbracket(s, s')$ of a transition relation f is determined over a pair of states s, s' (and a logic environment, which is omitted for clarity); the semantics of state condition g is defined

as $\llbracket g \rrbracket(s) \Leftrightarrow \forall s' : \llbracket g \rrbracket(s, s')$ and the semantics of a state independent-condition h is defined as $\llbracket h \rrbracket \Leftrightarrow \forall s, s' : \llbracket h \rrbracket(s, s')$.

In [7], the formal semantics of commands and formulas has been defined and the soundness of (a preliminary form of) the calculus has been proved. In [6], a concise definition of the semantics, judgments, and rules of (an older form of) the calculus is given.

B. Programming Language

In this appendix, we sketch the language that is used in the RISC ProgramExplorer for describing programs (its formal syntax is described in Appendix H.1). This programming language can in the following sense be considered as a “MiniJava”, i.e. as (a variant of) a subset of Java: Assume that a program can be parsed and type-checked by the RISC ProgramExplorer without error. If this program can be also compiled by the Java compiler without error, then the execution of the generated target code behaves as specified by Java¹.

Deviations In detail, MiniJava has the following *deviations* compared to Java (such that a program that can be parsed and type-checked by the RISC ProgramExplorer cannot be compiled in Java):

Visibility Modifiers The modifiers `public`, `protected`, and `private` are recognized but ignored; in fact MiniJava treats all entities as if declared with modifier `public`. Consequently, if a MiniJava program violates the specified access constraints, it cannot be compiled by a Java compiler.

Constraints The following items describe *constraints* of MiniJava (such that a program that can be compiled with Java cannot be parsed or type-checked by the RISC ProgramExplorer)².

Inheritance MiniJava does not support inheritance; every class denotes an object type that is incompatible with the object type of any other class.

Interfaces MiniJava does not support interfaces.

Method Calls A method call with a return value may only appear on the right side of a variable initialization or of a variable assignment, not as an expression within another expression.

For Loops The initialization part of a for loop header must be an initialized variable declaration or a variable assignment; the update part of the header must be a variable assignment or increment.

¹It should be noted that “MiniJava” was designed as a simple imperative programming language whose concrete syntax and semantics is immediately familiar to many programmers and can thus represent the basis for understanding formal specifications of imperative languages. It is not designed as the starting point of the specification of full Java.

²Actually, only the major constraints are listed (more constraints can be detected by investigating the syntax specified in Appendix H.1).

Throwing Exceptions An exception can be only thrown by a statement of form `throw new Exception(string)` where *string* denotes a string literal, respectively a value of type `java.lang.String`.

Null The keyword `null` may only appear on the right hand side of an initialization or assignment statement or on the right hand side of an equality or inequality expression.

Array Types Currently only one-dimensional arrays are supported (i.e. the base type of an array must not itself be an array type).

References The type system restricts a program such that that every object variable can be considered to hold the object *value* (rather than a *reference* to the value) which considerably simplifies reasoning about objects. This restriction ensures that two different references cannot denote the same object (and so an update of the object value via one reference cannot affect the object value denoted by any other reference). In particular,

- a variable of an object type may only receive the result of a constructor call or of a method call;
- a `return` statement may only return (the result of) a constructor call, a method call, or an object path `v . . .`, where *v* denotes a local variable of the current method;
- a method/constructor call may receive as an argument of an object type only (the result of) a constructor call, a method call, or an object path `v . . .` where *v* denotes a local variable or a method parameter that is not the base of an object path which appears as another argument in the same method/constructor call.

In the description above, an object path `v . . .` denotes the variable *v*, possibly trailed by a sequence of selectors of the form `.var` (an object variable selector) or `[exp]` (an array index selector).

Parameters In a method, a parameter that denotes an object or array must not be assigned a new value (but the *contents* of the object or array may be modified). The reason is that the RISC ProgramExplorer handles such parameters as “transient” (the corresponding arguments may have new values after the call of the method); the restriction ensures that this view coincides with the Java semantics of object/array parameters holding pointers.

Java Classes The RISC ProgramExplorer does not itself provide/implement the classes of the Java API (also not the classes `java.lang.String` used for character strings or the class `java.lang.System` used for standard input/output); if such classes are used in programs, the programmer must provide corresponding class stubs in (a subpackage of) a package `java` within the package hierarchy seen by the RISC ProgramExplorer (see Section E). If no class `String` is provided, strings are represented by a built-in `STRING` type.

Specification Comments The contents of program comments of the form `/*@ . . . @*/` and `//@ . . .` are interpreted as formal program specifications; the language of these specifications are explained in the following section.

C. Specification Language

In this appendix, we describe the language that is used in the RISC ProgramExplorer for specifying programs. This language is based upon the logic [language of the RISC ProofNavigator](#) as explained in Section C.1. With this language whose formal syntax is described in Appendix H.2, theories can be constructed as described in Sections C.2 and C.3. With the help of theories, we may specify programs as described in Sections C.5, C.6, and C.7.

C.1. Logic Language

The logic language of the RISC ProgramExplorer is based on the [language of the RISC ProofNavigator](#) [5, 8]. In the following, we only describe the differences respectively extensions.

C.1.1. Declarations

The logic language allows to introduce by declarations

- type constants,
- object/function/predicate constants,
- constants denoting formulas (to be proved) and axioms (assumed true).

While the language of RISC ProofNavigator considers both terms and formulas as elements of the syntactic domain (*value*) *expression* (formulas are just expressions denoting a Boolean value, mismatches between terms and formulas are detected by the type checker), the RISC ProgramExplorer decomposes *expression* into two syntactic domains *term* and *formula* (which already enables the parser to detect mismatches). Nevertheless, on the semantic level predicates are just considered as functions whose result is a Boolean value.

Object/function/predicate constants can now be defined as follows:

ident* : *type* = *term This definition introduces an object constant *ident* defined as *term*. If *type* denotes the type *BOOLEAN*, *ident* can be used as a 0-ary predicate constant.

ident* : *type* <=> *formula This definition introduces a 0-ary predicate constant *ident*; *type* must denote the type *BOOLEAN*.

ident* : *type* = LAMBDA (*params*) : *term This definition introduces a new function constant *ident* which is defined as a function that binds its concrete arguments to the parameters *params* and returns as a result the value of *term* in the environment set up by the

binding. Here *type* must denote a corresponding function type. If the domain of *type* denotes the type *BOOLEAN*, *ident* can be considered as a predicate constant.

ident* : *type*=PRED (*params*) : *formula This definition introduces a predicate constant *ident* which is defined as a predicate that binds its concrete arguments to the parameters *params* and returns as a result the truth value of *formula* in the environment set up by the binding. Here *type* must denote a corresponding function type whose domain denotes the type *BOOLEAN*; here the type PREDICATE (see the following subsection) is recommended.

C.1.2. Types

The RISC ProgramExplorer introduces the additional types

```
STRING
PREDICATE (types)
```

STRING is an unspecified type which plays a role in the mapping of program types to logical types, see the next subsection.

PREDICATE (*types*) is a synonym of

```
(types) -> BOOLEAN
```

The use of this type syntactically simplifies the definitions of predicate constants (see the previous subsection).

C.1.3. Mapping Program Types to Logical Types

The subsequent subsections describe how a logical formula may refer to the values of program variables. This requires the mapping of program values to logical values and of program types to logical types. The mapping is based on the automatically generated theory Base in the unnamed top-level package:

```
theory Base
{
  MIN_INT: INT = -2147483648;
  MAX_INT: INT = 2147483647;
  minIntAxiom: AXIOM MIN_INT < 0;
  maxIntAxiom: AXIOM MAX_INT > 0;
  int: TYPE = [MIN_INT..MAX_INT];
  nat: TYPE = [0..MAX_INT];
  char: TYPE;
  intTrunc: REAL -> INT;
  intTruncAxiom: AXIOM FORALL(x: REAL):
    (IF x >= 0 THEN x-1 < intTrunc(x) AND intTrunc(x) <= x
     ELSE x <= intTrunc(x) AND intTrunc(x) < x+1 ENDIF);
  intDivOverflow: (int, int) -> int;
```

```

intDiv0: (int, int) -> INT = LAMBDA(x: int, y: int):
  (IF y /= 0 THEN intTrunc(x/y) ELSE intDivOverflow(x, y) ENDIF);
zero: int = 0;
nullChar: char;
nullString: STRING;
newString: INT -> STRING;
IntArray: TYPE =
  [#value: ARRAY int OF int, length: nat, null: BOOLEAN#];
nullIntArray: IntArray =
  (#value:=ARRAY(y: int): zero, length:=zero, null:=TRUE#);
newIntArray: nat -> IntArray = LAMBDA(x: nat):
  (#value:=ARRAY(y: int): zero, length:=x, null:=FALSE#);
CharArray: TYPE =
  [#value: ARRAY int OF char, length: nat, null: BOOLEAN#];
nullCharArray: CharArray =
  (#value:=ARRAY(y: int): nullChar, length:=zero, null:=TRUE#);
newCharArray: nat -> CharArray = LAMBDA(x: nat):
  (#value:=ARRAY(y: int): nullChar, length:=x, null:=FALSE#);
BooleanArray: TYPE =
  [#value: ARRAY int OF BOOLEAN, length: nat, null: BOOLEAN#];
nullBooleanArray: BooleanArray =
  (#value:=ARRAY(y: int): FALSE, length:=zero, null:=TRUE#);
newBooleanArray: nat -> BooleanArray = LAMBDA(x: nat):
  (#value:=ARRAY(y: int): FALSE, length:=x, null:=FALSE#);
StringArray: TYPE =
  [#value: ARRAY int OF STRING, length: nat, null: BOOLEAN#];
nullStringArray: StringArray =
  (#value:=ARRAY(y: int): nullString, length:=zero, null:=TRUE#);
newStringArray: nat -> StringArray = LAMBDA(x: nat):
  (#value:=ARRAY(y: int): nullString, length:=x, null:=FALSE#);
}

```

In detail, program types are mapped to logical types as follows:

boolean The program type `boolean` is mapped to the logical type `BOOLEAN`.

int The program type `int` is mapped to the logical type `Base.int` (which is a subrange of the type `INT` of all integer numbers).

char The program type `char` is mapped to the logical type `Base.char` (which is currently unspecified).

class C Every class `C` is automatically translated to a theory `C` that resides in the same package as the class. This theory contains a record type `C` to which the program type (class) `C` is mapped. A record of this type contains

- one field for every object variable of the class;
- a field `null` of type `BOOLEAN` which, if set to `true`, indicates that the program value represents the `null` pointer (then the other fields are meaningless);
- a field `new` of type `INT` which is set to a unique value when the corresponding object is newly created (i.e., two object allocations yield different objects).

The theory also contains an array type `Array` to which the program type (class) `C[]` is mapped (see below for the details of array representations).

Additionally the automatically generated theory contains the following constants:

null: C This constant represents a program value of type `C` that is `null`.

newObject: INT->C This function is used to represent the allocation of a value of type `C` (the argument is an unspecified integer by which the `new` field of the object is initialized).

nullArray: Array This constant represents a program value of type `C[]` that is `null`.

newArray: INT->Array This function is used to represent the allocation of an array of type `C[]` (the argument represents the length of the array); an axiom `newArrayAxiom` specifies that all elements in the range of the array are `null`.

In the translation of an object access `object.field`, the RISC ProgramExplorer uses the precondition `NOT object.null`. If an object, for which this precondition can be established is updated by an assignment to an object variable, also the new object value satisfies that condition. On the other side, `null` is just an unspecified value for which the precondition simply can not be established. As a consequence, in method preconditions not the test `object /= C.null` but the test `NOT object.null` must be applied. Therefore also a program test `object != null` is automatically translated to the logical test `NOT object.null`.

Character strings If the user provides a program class `java.lang.String`, string literals in programs are considered as values of this class which is mapped to the logical type `java.lang.String.String`.

However, if the user does not provide such a class, string literals in programs are considered as values of a pseudo-type that is mapped to the logical type `STRING`.

T[] If `T` denotes a class, then the program type `T[]` is mapped to the record type `Array` in the theory of `C`. If `T` is `boolean`, `int`, `char`, or `String`, the theory `Base` contains a record type `TArray` (the name of `T` is capitalized) to which the program type `T[]` is mapped. Other array types are currently not supported.

The record type contains the following fields:

value: ARRAY Base.int OF T' This field represents the sequence of array elements; `T'` is the type to which the program type `T` is mapped.

length: Base.nat This field indicates that in `value` only the elements at indices from 0 inclusive to `length` exclusive are defined.

null: BOOLEAN If set to `true`, this field indicates that the program value represents the `null` pointer (then the other fields are meaningless);

A class `T` contains also constants `nullTArray` and `newTArray` that represent array values that are `null` respectively result from the allocation of a new array (see above).

If T is `boolean`, `int`, `char`, or `String`, the theory `Base` contains also constants `nullTArray` and `newTArray` (again the name of T is capitalized) that represent array values that are `null` respectively result from the allocation of a new array.

As for objects, in method preconditions not the test `array /= T.null` but the test `NOT array.null` must be applied (see the explanation for objects above).

C.1.4. Program Variables

Synopsis

```
OLD var
VAR var
```

Description Within the context of a state predicate of a specification (e.g. in a class invariant or in a method precondition), both `OLD var` and `VAR var` refer to the “current” state of the program variable `var`.

Within the context of a state relation of a specification (e.g. a method postcondition or loop invariant), `OLD var` refers to the value of the program variable `var` in the prestate of the specified execution; `VAR var` refers to the value of `var` in the corresponding poststate.

More specifically, `OLD var` respectively `VAR var` denotes the logical value to which the value of the program variable is mapped. Therefore the type of `OLD var` respectively `VAR var` is the logical type to which the type of the program variable is mapped.

Pragmatics A reference to a program variable `var` in a formula is tagged with keyword `OLD` or `VAR` to explicitly distinguish it from a reference to a logical variable; we thus emphasize that its value actually results from mapping a program value to a logical value.

We choose the keywords and their interpretations in both state conditions and state relations in order to minimize the confusion of programmers:

- If there is a corresponding state relation (e.g. method postcondition), we may prefer in the precondition the use of `OLD var` since we thus refer in both the precondition and the postcondition to the same value in the same way.

However, if there is no corresponding state relation, the syntax `OLD var` in a state condition looks awkward since the condition only refers to a single state: here we may prefer `VAR var`.

- In a loop invariant (which also denotes a state relation), `VAR var` refers to the value of the variable after the execution of the loop body, while `OLD var` refers to the state of the variable in the prestate of the loop. If the invariant does not refer to the prestate (as it is often the case), the invariant can be thus expressed in terms of `VAR var` only.

C.1.5. Program States

The logic language introduces a new kind of values called *states* with corresponding types, constants, functions, and predicates.

Type STATE

Synopsis

```
STATE
STATE (type)
```

Description A type of this family denotes the set of states that may result from the execution of a command. The type `STATE` indicates that the execution of the command must not return a value (i.e. that the command is executed within a function of result type `void`); the type `STATE (type)` indicates that the command may return a value of the denoted *type*.

State Constants

Synopsis

```
NOW
NEXT
```

Description Within the context of a state predicate of a specification (e.g. a method precondition), both constants `NOW` and `NEXT` denote the “current” state.

Within the context of a state relation of a specification (e.g. a method postcondition or loop invariant), the constant `NOW` denotes the prestate of the specified execution while the constant `NEXT` denotes the corresponding poststate.

Pragmatics To simplify the semantics, `NEXT` is also defined in the context of a state predicate.

In a loop invariant, `NOW` refers to the prestate of the loop, while `NEXT` refers to the poststate of the loop body.

C.1.6. State Functions

Synopsis

```
VALUE@state
MESSAGE@state
```

Description These functions are evaluated over *state* whose type is of form `STATE` or `STATE(result)`.

If *state* results from the execution of `return value`, then the term `VALUE@next` refers to (the logical mapping of) *value*. The type of *state* must be of form `STATE(result)`; the type of `VALUE@next` is *result* (which is the logical mapping of the type of *value*).

If *state* results from the execution of `throw new exception(message)`, the term `MESSAGE@next` refers to (the logical mapping of) *message*. Its type is the logical mapping of the program type `java.lang.String` (which must be the type of *message*).

State Predicates

Synopsis

```
EXECUTES@state
CONTINUES@state
BREAKS@state
RETURNS@state
THROWS@state
THROWS(exception)@state
```

Description These predicates are evaluated over *state* whose type is of form `STATE` or `STATE(result)`:

- `EXECUTES@state` is true if and only if none of the following four predicates is true.
- `CONTINUES@state` is true if and only if *state* results from executing `continue`.
- `BREAKS@state` is true if and only if *state* results from the execution of `break`.
- `RETURNS@state` is true if and only if *state* results from the execution of `return` or `return value`.
- `THROWS@state` is true if and only if *state* results from the execution of `throw new exception(message)` (for any *exception* type and string *message*).
- `THROWS(exception)@state` is true if and only if *state* results from the execution of `throw new exception(message)` (for any character string *message*).

State Equality

Synopsis

```
state1~state2
```

Description This predicate is evaluated over two states *state1* and *state2* which may be of different state types `STATE(type1)` and `STATE(type2)` (respectively `STATE`). The result is true if and only if both states are equal except for their values `VALUE@state1` respectively `VALUE@state2` (if applicable).

Pragmatics The predicate may be required to express the relationship between the post-state of a called method and the poststate of the calling method (which may have different return types).

State Pair Predicates

Synopsis

```
READONLY
WRITESONLY var, ...
```

Description These formulas are evaluated in the context of a pair of execution states (e.g. a method postcondition or loop invariant) called the “prestate” and the “poststate” of the execution.

READONLY is true if and only if the value of every program variable is in the poststate of the execution the same as in the prestate.

WRITESONLY *name*, ... is true if and only if the value of every program variable that is not listed in “*var*, ...” is in the poststate of the execution the same as in the prestate.

C.2. Theory Definitions

Synopsis

```
package package ;
import package.* ;
import package.theory ;
...
theory theory uses theories
{ declarations }
```

Description A theory definition introduces by a list of *declarations* a “theory” i.e. a collection of logic entities that may be used in other theories or for the specification of programs.

The clause *theory theory* states the name of the theory as *theory*. The optional clause *package package* states that the new theory resides in *package* and may be referenced elsewhere by the long name *package.theory*; likewise any *entity* this is introduced by *declarations* may be referenced elsewhere by *package.theory.entity*. If the *package* clause is omitted, the theory resides in the unnamed top-level package.

An *import* clause imports theories from other packages such that they may be referenced from the current theory not only by their long names of form *package.theory* by also by their short names of form *theory*. A clause

```
import package.*;
```

imports all theories from *package*; a clause

```
import package.theory;
```

imports from *package* only *theory*. If multiple *package.** import theories with the same name, these theories can be only referenced by their long name unless one of the packages is also imported as *package.theory*; then this theory can also be referenced by the short name. Multiple *package.theory* imports of different theories with the same short name *theory* are prohibited.

Every theory referenced by declarations in the current theory must be listed in the clause *uses theory, . . .*, either by the long name of the theory or, if the theory was imported, by its short name.

Pragmatics A theory with long name *package.theory* must reside in a file with name *theory.theory* in a subdirectory *package* of a directory that is considered as a root of the package hierarchy. The name *package* may have form *p1.p2 . . . pn*; the corresponding directory path is then *p1/p2/ . . . /pn*.

The clause *import . . .* is modeled after the semantics of the corresponding Java clause but imports theories rather than classes.

The clause *uses theory, . . .* was introduced to simplify the computation of dependencies between classes and theories; in a subsequent version of the language, this clause may be well dropped.

C.3. Class Specifications

Synopsis

```
/*@
  import package.*;
  import package.theory;
  ...
  theory uses theory, ...
  { declarations }
  @*/
classheader { ... }
```

Descriptions A class specification introduces by a list of *declarations* the “local theory” of a class i.e. a theory of those entities that may be referenced by their short names in the specification of methods, loops, and commands of the class (the entities introduced in other theories may be always referenced by the long name *package.theory.entity*). If a

class has no such specification, the local theory is empty; the specifications in this class may therefore only refer to entities introduced in other theories.

An `import` clause imports theories from other packages, see Section C.2.

Every theory referenced by declarations in the local theory (respectively by the specifications of methods, loops, statements in the current class) must be listed in the clause `uses theory, ...`, either by the long name of the theory or, if the theory was imported, by its short name.

Pragmatics The clause `import ...` is modeled after the semantics of the corresponding Java clause but imports theories rather than classes.

The clause `uses theory, ...` was introduced to simplify the computation of dependencies between classes and theories; in a subsequent version of the language, this clause may be well dropped.

C.4. Class Invariants

Synopsis

```
classheader {
  /*@ invariant formula ; */
  ...
}
```

Descriptions A class invariant denotes a *formula* that is implicitly added

- to every precondition of every (non-constructor) object method, and
- to every postcondition of every constructor and object method.

As an exception, if the constructor or object method is marked as a helper (see Section C.5), then the formula is added neither to the precondition nor to the postcondition.

Pragmatics A class invariant ensures that, after the allocation of an object, during its full life-time, at every call/return from a (non-helper) method invoked on that object, the stated formula holds.

C.5. Method Specifications

Synopsis

```

methodheader
/*@
  helper;
  assignable vars ;
  signals exceptions ;
  requires formula ;
  diverges formula ;
  ensures formula ;
  decreases term, ... ;
@*/
{ statements }

```

Description This specification describes the observable behavior of a given method (class method, object method, or constructor) by the following clauses:

helper This optional clause may be given for a constructor or an object method. It indicates that the method is a helper method that must not assume the class invariant as its precondition and need not ensure the class invariant as its postcondition (see Section C.4).

assignable vars This optional clause lists the variables *vars* that are visible in the scope of the declaration of the method (object and class variables of the current class, class variables of other classes, respectively variables that represent components of such variables, but *not* local variables of the method) and whose values may be changed by the execution of the method.

If the clause is omitted, the method must not modify any variable that is visible in the scope of the method declaration.

If the variable `this` is added to the clause, it indicates that *all* object variables of the current class may be modified.

A *parameter* of the current method may be only listed in the clause, if it denotes an array or an object; the clause then indicates that the *contents* of the array/object may be changed by the call of the method.

signals exceptions This optional clause lists the types of the *exceptions* that may be thrown by the execution of the method (excluding “runtime exceptions” such as “division by zero” that may be thrown by the execution of primitive operations).

If the clause is omitted, the method must not throw any exception.

requires formula This optional clause states that it is only legal to call the method in a state (the method’s “prestate”) that satisfies the given *formula*.

If the clause is omitted, the *formula* is considered as “true”, i.e. it is legal to call the method in any state.

diverges formula This optional clause states that the method will terminate (by returning normally or by throwing an exception) when called in any legal state that satisfies

also the negation of *formula* (i.e. the method is allowed to run forever when called in any legal state that satisfies *formula*).

If the clause is omitted, the *formula* is considered as “false”, i.e. the method must terminate when called in any legal prestate.

ensures *formula* This optional clause states that, for every legal prestate of the method, every state in which the method terminates is only legal if it is related to the method’s prestate by *formula*.

If the clause is omitted, the *formula* is considered as “true”, i.e. the method may terminate with any poststate.

decreases *term*, ... This optional clause states that, for every call of the method in a legal state, the value of the a given term sequence decreases according to a well founded ordering. If the sequence consists of a single *term*, the term denotes a non-negative integer number which is decreased in every (directly or indirectly) recursive call of the method (such that chain of recursive method calls must eventually end). If the sequence has more than one elements, then the values of some $term_1, \dots, term_i$ may remain the same while $term_{i+1}$ is decreased as described above (decreasing lexicographical ordering). If the clause is omitted, no default is assumed.

Pragmatics This specification is in essence modeled after the “light-weight” specification format of JML, the Java Modeling Language [4]; however, a fixed order is required and specific default values for missing clauses are given. Furthermore, the specification follows (not precedes) the method’s declaration header to emphasize that the specification appears in the scope of the parameters of the method.

If the clause `decreases term` is missing in a (directly or indirectly recursive) method, the termination of the method can be probably not proved.

If in an `assignable` clause specific components of an object (e.g. specific fields of the current object) are listed, the “frame condition” itself just verifies that only the object (e.g. `this`) is modified; however, the “postcondition” is appropriately extended to ensure that only the specified components of the object are modified.

In an `assignable` clause, parameters may be listed that denote objects or arrays. The RISC ProgramExplorer handles such parameters not as input parameters but as transient parameters that may have a new value after the call of the method (methods are allowed to update the components of such parameters, not to assign new values to them). Therefore e.g. also a method that sorts a given array in place may be appropriately analyzed and verified.

C.6. Loop Specifications

Synopsis

```

while (exp)                for (forheader)
/*@                       /*@
  invariant formula ;      invariant formula ;
  decreases term, ... ;    decreases term, ... ;
@*/                        @*/
body                       body

```

Description The optional clause `invariant formula` states that the state in which the loop checks the value of `exp` for the first time (the loop’s “prestate”) is related by `formula`

1. to the loop’s prestate itself and
2. to every state that arises immediately after the execution of the loop’s `body` (the body’s “poststate”).

If the clause is omitted, the formula is assumed to be “true”.

The optional clause `decreases term` states that

1. the value of `term` in the loop’s prestate and in every poststate of the loop’s `body` denotes a non-negative integer number, and that
2. the value of `term` immediately before the execution of the loop’s `body` is greater than the value of `term` after the execution of the loop’s `body`.

Consequently the loop cannot perform an infinite number of iterations.

If the clause has form `decreases term1, ..., termn` with $n > 1$, then then the values of some `term1, ..., termi` may remain the same while `termi+1` with $i + 1 \leq n$ is decreased as described above (decreasing lexicographical ordering).

Pragmatics It should be noted that the formulation of the invariant above relates the loop’s prestate to the body’s poststate which, due to the existence of state functions and state predicates in the formula language, may be considered as different from the prestate of the subsequent loop iteration, respectively, if the loop terminates, from the loop’s poststate. For instance, if the body executes a `break` statement, the loop’s prestate is related to the body’s poststate by the formula `BREAKS@NEXT` but to the loop’s poststate by `EXECUTES@NEXT`. The first formula is more precise since it describes that the loop terminates from the execution of the loop body which the second formula does not. Our formulation therefore allows to express stronger invariants.

C.7. Statement Specifications

Synopsis

```

/*@ assert formula ';'
statement

```

Description The specification states that immediately before the execution of *statement* (i.e. in the statement's "prestate") *formula* holds.

Pragmatics The specification creates an additional proof obligation but then also more information for the verification of *statement* and its successors.

D. New RISC ProofNavigator

The RISC ProgramExplorer includes an updated version of the RISC ProofNavigator; as shown in Appendix E, the RISC ProgramExplorer may also be invoked in a mode that exposes only the RISC ProofNavigator interface and can be used very much like the original RISC ProofNavigator.

However, there are various differences between the new version of the RISC ProofNavigator and the original one:

Bug Fixes The new version fixes various bugs of the original version; these fixes have *not* been propagated to the original version, i.e. the original version is not maintained any more. It is therefore strongly recommended to switch to the new version included in the RISC ProgramExplorer.

Interaction On some occasions, the RISC ProofNavigator asks the user for input (e.g., whether the user really wants to quit). In the original version, the input was provided by text input from the command line. In the new version, a corresponding interaction window pops up.

Commands The original RISC ProofNavigator supported the commands `type`, `formula`, `value` for printing the definition of a type, formula, or value; the command names could thus not be used as identifiers.

The new version calls these commands `printt`, `printv`, and `printf`.

Context Directories The contents of the context directories have been stream-lined (see Appendix G).

Since these changes are only minor, we still refer to the original manual [5] for the documentation of the RISC ProofNavigator. However, in the future, the new version may further diverge from the original one.

E. Software Invocation

The shell script `ProgramExplorer` is the main interface to the program i.e. the program is typically started by executing

```
ProgramExplorer &
```

However, if the script is copied/renamed/linked to `ProofNavigator` and executed as

```
ProofNavigator &
```

the program starts with a standalone interface to the RISC ProofNavigator [5] (which is part of the RISC ProgramExplorer).

Invoking the script as

```
ProgramExplorer -h
```

gives the following output which lists the available startup options and the environment variables used:

```
RISC ProgramExplorer Version 1.0 (September XX, 2011)
http://www.risc.jku.at/research/formal/software/ProgramExplorer
(C) 2008-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "ProgramExplorer -h" to see the options available.
```

```
-----
Usage: ProgramExplorer [OPTION]...
OPTION: one of the following options:
  -h, --help: print this message.
  -cp, --classpath [PATH]: directories representing
                           the top package.
```

Environment Variables:

```
PE_CLASSPATH:
    the directories (separated by ":") representing
    the top package
    (default the current working directory)
PE_CVCL
    the command for executing the cvcl checker
    (default "cvcl")
```

```

PE_JAVAC
    the command for compiling java programs
    (default "javac")
PE_JAVA
    the command for executing java programs
    (default "java")
PE_CWD
    the directory used for compiling/executing
    (default the current working directory)
PE_MAIN
    the name of the main class of the program
    (default "Main")

```

The command accepts the following startup options:

- h, --help** With this option, the description shown above is printed and the program terminates.
- cp, --classpath *Path*** This option expects as *Path* a sequence of directories separated by the colon character “:”. The program considers these directories to jointly represent the root of the package hierarchy; by default, the current working directory (path “.”) alone represents the root. The directories in *Path* must not have different class files (extension `.java`), theory files (extension `.theory`), or subdirectories of the same name.

The program uses the values of the following environment variables.

- PE_CLASSPATH** If the program is started without the option `-cp/-classpath Path`, the value of this variable is considered as the *Path*, see the description of the option given above.
- PE_CVCL** The value of this environment variable is considered as the path to the executable of the Cooperating Validity Checker (CVC) Lite version 2.0; by default, the path `cvcl` is assumed.
- PE_JAVAC** The value of this environment variable is considered as the path to the executable of the Java compiler; by default, the path `javac` is assumed.
- PE_JAVA** The value of this environment variable is considered as the path to the executable of the Java runtime environment; by default, the path `java` is assumed.
- PE_CWD** The value of this environment variable is considered as the path of the directory used for compiling/executing respectively creating subdirectories; by the default the current working directory “.” is used.
- PE_MAIN** The value of this environment variable is considered as the name of the main class of the program to be compiled and executed; by default the value `Main` is used.

F. Software Installation

The installation of the program is thoroughly described in the files README and INSTALL of the distribution; we include these files verbatim below.

F.1. README

README

Information on the RISC ProgramExplorer.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Copyright (C) 2008-, 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 ProgramExplorer

<http://www.risc.jku.at/research/formal/software/ProgramExplorer>

This is the RISC ProgramExplorer, an interactive program reasoning environment developed at the Research Institute for Symbolic Computation (RISC). This software is freely available under the terms of the GNU General Public License, see file COPYING. The RISC ProgramExplorer runs on computers with x86-compatible processors under the GNU/Linux operating system. For learning how to use the software, see the file "main.pdf" in the directory "manual"; examples can be found in the directory "examples".

The current version is a release candidate that

- * provides the overall technological and semantic framework (programming language and formal specification language),
- * translates annotated programs into the semantic model (programs commands as state relations) which is open for human investigation,
- * generates from the semantic model the verification conditions which can be semi-automatically proved with the help of

the RISC ProofNavigator, an interactive proof assistant which is integrated into the RISC ProgramExplorer.

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 the RISC ProgramExplorer

On the RISC ProgramExplorer web site, you can find a virtual GNU/Linux machine that has the RISC ProgramExplorer 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 RISC ProgramExplorer; if you choose this option, see the web site for further instructions.

Running the RISC ProgramExplorer Examples

The installation of the RISC ProgramExplorer contains a subdirectory "examples" with a number of specified and verified example programs. To (re)run the examples, go to the directory, unzip the PETASKS.tgz file and start the RISC ProgramExplorer:

```
cd examples
tar xzf PETASKS.tgz
ProgramExplorer &
```

Select the tab "Symbols" and double-click e.g. on "Sum" to see the file "Sum.java". Right-click in the "Symbols" tab the method "sum" and select "Show Semantics" to see the method semantics. Right-click in the "Tasks" tab any task displayed in purple and select "Execute Task" to replay the corresponding proof.

Third Party Software

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

CVC Lite 2.0
<http://www.cs.nyu.edu/acsys/cvcl>

This is a C++ library/program for validity checking in various theories.

The RISC ProgramExplorer currently only works with CVCL 2.0, not the newer CVC3 available from <http://www.cs.nyu.edu/acsys/cvc3>. To download the CVCL 2.0 source, go to the RISC ProofNavigator web site (URL see above), Section "Third Party Software", and click on the link "CVCL 2.0 local copy".

RIACA OpenMath Library 2.0
<http://www.riaca.win.tue.nl/products/openmath/lib>

This is a library for converting mathematical objects to/from the OpenMath representation.

Go to the link "OMLib 2.0" and then "Downloads". Download one of the "om-lib-src-2.0-rc2.*" files.

General Purpose Hash Function Algorithms Library
<http://www.partow.net/programming/hashfunctions>

A library of hash functions implemented in various languages.

Go to the link "General Hash Function Source Code (Java)" to download the corresponding zip file.

ANTLR 3.2
<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.

On a Debian 6.0 GNU/Linux distribution, just install the package "antlr3" by executing (as superuser) the command

```
apt-get install antlr3
```

ANTLR 2.7.6b2
<http://www.antlr.org>

This is a framework for constructing parsers and lexical analyzers used for processing the logic language of the RISC ProofNavigator.

On a Debian 6.0 GNU/Linux distribution, just install the package "antlr" by executing (as superuser) the command

```
apt-get install antlr
```

The Eclipse Standard Widget Toolkit 3.7
<http://www.eclipse.org/swt>

This is a widget set for developing GUIs in Java.

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

Mozilla Firefox 3.* or SeaMonkey 2.* (or higher)
<http://www.mozilla.org>

See the question "What do I need to run the SWT browser in a standalone application on Linux GTK or Linux Motif?" in the FAQ at <http://www.eclipse.org/swt/faq.php>.

Chances are that the SWT browser will work with the Firefox included in your Linux distribution (but it will *not* work with the Firefox downloaded from the Mozilla site). For instance, on a Debian 6.0 GNU/Linux distribution, just install Firefox by executing (as superuser) the command

```
apt-get install iceweasel
```

If the SWT browser does not work with the Firefox included in your GNU/Linux distribution, go to the page <http://www.mozilla.org/projects/seamonkey> to download and install the SeaMonkey 2.* browser instead. You might have to set the environment variable MOZILLA_FIVE_HOME in the "ProgramExplorer" script to "/usr/lib/mozilla".

The GIMP Toolkit GTK+ 2.X (or higher)
<http://www.gtk.org>

This library is required by "Eclipse Linux (x86/GTK2)" and by "Mozilla 1.7.8 GTK2".

On a Debian 6 GNU/Linux distribution, the package is automatically installed, if you install the "mozilla-browser" package (see above).

On another GNU/Linux distribution, go to the GTK web package, section "Download", to download GTK+.

Java Development Kit 6 (or higher)
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Go to the "Downloads" section to download the JDK 6.

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

Go to the section "Base Icon Library", subsection "Download", to download the icons used in the ProgramExplorer.

End of README.

F.2. INSTALL

----- INSTALL

Installation notes for the RISC ProgramExplorer.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Copyright (C) 2008-, 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/>>.

----- Installing the RISC ProgramExplorer

The RISC ProgramExplorer is available for computers with x86-compatible processors (32 bit as well as 64 bit) running under the GNU/Linux operating system. The core of the RISC ProgramExplorer is written in Java but it depends on various third-party open source libraries and programs that are acknowledged in the README file.

To use the RISC ProgramExplorer, you have three options:

- A) You can just use the distribution, or
- B) you can compile the source code contained in the distribution, or
- C) you can download the source from a Subversion repository and compile it.

The procedures for the three options A-C are described below, but please read the following remark first.

Mathematical Fonts in the RISC ProgramExplorer

 After an installation of the RISC ProgramExplorer, the mathematical fonts displayed by the RISC ProgramExplorer (i.e. by the embedded Mozilla browser) may not look nice. For an aesthetically pleasing display, proceed as described on

Fonts for MathML-enabled Mozilla
https://developer.mozilla.org/en/Mozilla_MathML_Project/Fonts

In a nutshell, create in your home directory a subdirectory ".fonts", download into this directory the STIX font archive "STIX-mozilla1.9.zip", and unzip the archive (which will create a number of font files .fonts/*.otf). Then (re)start the RISC ProgramExplorer.

A) Using the Distribution

We provide a distribution for computers with ix86-compatible processors running under the GNU/Linux operating system (the software has been developed on the Debian 6.0 "squeeze" distribution, but any other distribution will work as well). If you have such a computer, you need to make sure that you also have

- 1) A Java 6 or higher runtime environment.

You can download a JRE 6 from
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

- 2) The Mozilla Firefox or SeaMonkey browser.

On a Debian 6.0 GNU/Linux system, just install the package "iceweasel" by executing (as superuser) the command

```
apt-get install iceweasel
```

On other Linux distributions, first look up the FAQ on

<http://www.eclipse.org/swt/faq.php>

for the question "What do I need to run the SWT browser in a standalone application on Linux GTK or Linux Motif?" The RISC ProgramExplorer uses the SWT browser, thus you have to install the software described in the FAQ.

See the README file for further information.

- 3) The GIMP Toolkit GTK+ 2.6.X or higher.

On a Debian 6.0 GNU/Linux system, GTK+ is automatically installed, if you install the Mozilla browser as described in the previous paragraph.

On other Linux distributions, download GTK+ from <http://www.gtk.org>

For installing the RISC ProgramExplorer, first create a directory INSTALLDIR (where INSTALLDIR can be any directory path). Download from the website the file

ProgramExplorer-VERSION.tgz

(where VERSION is the number of the latest version of the ProofNavigator) into INSTALLDIR, go to INSTALLDIR and unpack by executing the following command:

```
tar xzf ProgramExplorer-VERSION.tgz
```

This will create the following files

README	... the readme file
INSTALL	... the installation notes (this file)
CHANGES	... the change history
COPYING	... the GNU Public License

```

bin/
  ProgramExplorer ... the main script to start the program
  cvcl            ... CVC Lite, a validity checker used by the software.
doc/
  index.html     ... API documentation
examples/
  README         ... short explanation of examples
  *.theory       ... some example theories
  *.java         ... some example program specifications
  PETASKS.tgz    ... an archive of sample program verifications
lib/
  Screenshot.png ... startup splash screen
  *.jar          ... Java archives with the program classes
  swt32/        ... SWT for GNU/Linux computers with 32 bit processors
    swt.jar
  swt64/        ... SWT for GNU/Linux computers with 64 bit processors
    swt.jar
manual/
  main.pdf       ... the PDF file for the manual
  index.html     ... the root of the HTML version of the manual
src/
  fmrisc/       ... the root directory of the Java package "fmrisc"
  ProgramExplorer/
    Main.java    ... the main class for the RISC ProgramExplorer
  ProofNavigator/
    *.java       ... the sources for the RISC ProofNavigator
  External/
    *.java       ... third-party sources

```

Open in a text editor the script "ProgramExplorer" in directory "bin" and customize the variables defined for several locations of your environment. In particular, the distribution is configured to run on a 32-bit processor. If you use a 64-bit processor, uncomment the line "SWTDIR=\$LIBDIR/swt64" (and remove the line "SWTDIR=\$LIBDIR/swt32").

Put the "bin" directory into your PATH

```
export PATH=$PATH:INSTALLDIR/bin
```

You should now be able to execute

```
ProgramExplorer
```

to run the RISC ProgramExplorer. If you rename/copy/link the script to "ProofNavigator" and execute

```
ProofNavigator
```

the program starts with a standalone interface to the RISC ProofNavigator.

B) Compiling the Source Code

To compile the Java source, first make sure that you have the Java 6 SE development environment installed. You can download the Java 6 SE from

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

Furthermore, on a GNU/Linux system you need also the Mozilla Firefox or SeaMonkey browser, GTK2 and the GIMP toolkit GTK+ installed (see Section A).

Now download the distribution and unpack it as described in Section A.

The RISC ProgramExplorer distribution contains an executable of the validity checker CVC Lite for GNU/Linux computers with x86-compatible processors. To compile the validity checker for other systems, you need to download the CVC Lite source code (see the README file) and compile it with a C++ compiler. See

the CVC Lite documentation for more details.

To compile the Java source code, go to the "src" directory and execute from there

```
javac -cp "...\lib/*:...\lib/swt32/*" fmrisc/ProgramExplorer/Main.java
```

(replace "swt32" by "swt64" on a 64bit system).

You may ignore the warning about "unchecked" or "unsafe" operations, this refers to Java files generated automatically from ANTLR grammars.

Then execute

```
jar cf ../lib/fmrisc.jar fmrisc/**/*.class fmrisc/**/*.class fmrisc/**/*.class
```

Finally, you have to customize the "ProgramExplorer" script in directory "bin" as described in Section A. You should then be able to start the program by executing the script.

C) Downloading the Source Code from the Subversion Repository

You can now download the source code of any version of the ProofNavigator directly from the ProofNavigator Subversion repository.

To prepare the download, first create a directory SOURCEDIR (where SOURCEDIR can be any directory path).

To download the source code, you need a Subversion client, see http://en.wikipedia.org/wiki/Comparison_of_Subversion_clients for a list of available clients. On a computer with the Debian 6.0 distribution of GNU/Linux, it suffices to install the "svn" package by executing (as superuser) the command

```
apt-get install svn
```

which will provide the "svn" command line client.

Every ProofNavigator distribution has a version number VERSION (e.g. "0.1"), the corresponding Subversion URL is

```
svn://svn.risc.jku.at/schreine/FM-RISC/tags/VERSION
```

If you have the "svn" command-line client installed, execute the command

```
svn export
  svn://svn.risc.jku.at/schreine/FM-RISC/tags/VERSION SOURCEDIR
```

to download the source code into SOURCEDIR. With other Subversion clients, you have to check the corresponding documentation on how to download a directory tree using the URL svn://... shown above.

After the download, SOURCEDIR will contain the files of the distribution as shown in Section A; you can compile the source code as explained in Section B.

End of INSTALL.

G. Task Directories

The system generates in the current working directory (respectively the directory specified by the environment variable `PE_CWD`, see the previous section) two subdirectories named `.ProofNavigator` and `.PETASKS.Tag.0`.

The directory `.ProofNavigator` represents a context directory of essentially that form that is described in the [manual of the RISC ProofNavigator \[5\]](#); it is used, if the user enters in the *Analyze* view directly (not in the context of any task as described below) commands for the RISC ProofNavigator. In order to save disk space, the format of the entries has been slightly changed: for every declaration of a logical entity *name* of kind *kind*, rather than three separate files `kind_name_hash.txt`, `kind_name_time.txt`, and `kind_name_refs.xml`, a single file `kind_name_info.xml` is generated that combines the information of the three files; furthermore, the file `kind_name_decl.xgz` is not generated any more.

The directory `.PETASKS.Tag.0` represents the persistent store for the task tree of the program; *Tag* is a number that denotes the time when the program was started that created this directory. The content of the directory is a hierarchy of subdirectories that corresponds to the hierarchy of task folders and tasks of the program. Each directory is named `Name.Tag.Cntr` where *Name* is derived from the name of the task folder respectively task, *Tag* denotes the time when the program was started that created this directory, and *Cntr* represents an automatically generated sequence counter.

The content of each task directory depends on the particular kind of the task. Currently the directory may contain the following items:

File `goal_` This file contains the log of an attempt to perform the task fully automatically by translating it to a CVCL query and invoking CVCL.

Directory `ProofNavigator` This represents a [context directory of the RISC ProofNavigator \[5\]](#) that contains all information related to an attempt to perform the task by a computer-assisted manual proof (also here the contents have been slightly changed as described above).

The directory `.PETASKS.Tag.0` and each of its subdirectories contains a file `.PEDIR`; if the directory contains also a file `FREED` this indicates that the directory was freed and may be reused. If a new directory is to be created, it is first attempted to reuse a directory with the same basic *Name* from a previous invocation of the program (as indicated by *Tag*) or a freed directory of the same invocation (as indicated by *Tag* and *FREED*); in both cases, thus previously created RISC ProofNavigator proofs of tasks with the same names will be retained. Otherwise, a new directory is created; if a directory of the desired name already exists, the value of *Cntr* is incremented to yield a new directory name.

H. Grammars

In this appendix, we describe the concrete syntax of the programming language and of the specification language. The grammars are given in the notation of the parser generator ANTLR v3 [1] used for the implementation of the parser and of the lexical analyzer. Non-determinism in grammatical rules is resolved by extra means provided by ANTLR (in particular semantic predicates) which are omitted from this presentation. On the level of the programming language described in Section H.1, every specification annotation is lexically parsed as a comment yielding the token *ANNOTATION*; the actual grammar of the various kinds of annotations is described in Section H.2 under the header “specifications” by the syntactic domains *unitspec*, *methodspec*, *loopspec*, and *statementspec*. The grammar of theory declarations is specified there by the syntactic domain *theorydecl*.

H.1. Programming Language

```
// -----  
// classes and methods  
// -----  
  
// a compilation unit  
unit: clasdecl ;  
  
// a class declaration  
clasdecl:  
    ( 'package' name ';' )?  
    ( 'import' name ( '.' '*' )? ';' )*  
    ( ANNOTATION )?  
    ( 'abstract' | 'final' | 'public' )* 'class' IDENT  
    ( 'extends' name 'implements' names )?  
    ( ANNOTATION )?  
    '{' ( topdecl )* '}'  
    EOF ;  
  
// a top-level declaration  
topdecl:  
    objectvar | classvar | constructor | objectmethod | classmethod ;  
  
// an object variable, possibly with initialization  
objectvar:  
    modifiers typeexp IDENT ( '=' valexprnull )? ';' ;
```

```

// a class variable, possibly with initialization
classvar:
  modifiers 'static' modifiers typeexp IDENT
  ( '=' valexpnull )? ';' ;

// a constructor declaration
constructor:
  visibility IDENT '(' ( params )? ')'
  throwdecls ( ANNOTATION )? block ;

// declaration of an object method
objectmethod:
  modifiers ( typeexp | 'void' ) IDENT '(' ( params )? ')'
  throwdecls ( ANNOTATION )? block ;

// declaration of a class method
classmethod:
  modifiers 'static' modifiers
  ( typeexp | 'void' ) IDENT '(' ( params )? ')'
  throwdecls ( ANNOTATION )? block ;

// -----
// statements
// -----

// an execution statement
statement:
  ( an=ANNOTATION )?
  ( emptystat | block | assignment | methodcall | localvar
  | conditional | whileloop | forloop
  | continuestat | breakstat | returnstat | throwstat
  | trycatch | assertion ) ;

// an empty statement
emptystat: ';' ;

// a statement block
block: '{' ( statement )* '}' ;

// an assignment or method call with return value
assignment: assigncore ';' ;

// the core of an assignment statement
assigncore:
  lval
  ( '='
  ( valexpnull | name '(' valexps ')' | 'new' name '(' valexps ')' )
  | '++'
  | '+=' valexp
  | '--'
  | '-=' valexp
  ) ;

```

```

// a method call without return value
methodcall: name '(' valexps ')' ';' ;

// a local variable declaration, possibly with initialization
localvar: localvarcore ';' ;

// the core of a local variable declaration
localvarcore:
  ( 'final' )? typeexp IDENT
  ( '='
    ( valexpnull
      | name '(' valexps ')'
      | 'new' name '(' valexps ')' )
  )? ;

// a conditional statement with one or two branches
conditional: 'if' '(' valexp ')' statement ( 'else' statement )? ;

// a while loop
whileloop: 'while' '(' valexp ')' ( ANNOTATION )? statement ;

// a for loop
forloop:
  'for' '('
    ( assigncore | localvarcore )? ';' ( valexp )? ';' ( assigncore )?
  ')' ( ANNOTATION )? statement ;

// a continue statement
continuestat: 'continue' ';' ;

// a break statement
breakstat: 'break' ';' ;

// a return statement, possibly with return value
returnstat: 'return' ( valexpnull )? ';' ;

// a throw statement
throwstat: 'throw' 'new' name '(' valexp ')' ';' ;

// a try catch block
trycatch: 'try' block ( 'catch' '(' param ')' block )+ ;

// an assertion
assertion: 'assert' valexp ';' ;

// -----
// value expressions
// -----

// a value expression that also includes "null"
valexpnull: valexp | 'null' ;

```

```
// value expressions
valexp: valexp3 ;

// disjunctions
valexp3: valexp4 ( '||' valexp4 )* ;

// conjunctions
valexp4: valexp8 ( '&&' v1=valexp8 )* ;

// equalities/inequalities
valexp8:
  valexp9
  ( '==' valexp9 | '!=' valexp9 | '==' 'null' | '!=' 'null' )* ;

// relations
valexp9:
  valexp11
  ( '<' valexp11 | '<=' valexp11 | '>' valexp11 | '>=' valexp11 )* ;

// sums and differences
valexp11: valexp12 ( '+' valexp12 | '-' valexp12 )* ;

// products and quotients
valexp12: valexp13 ( '*' valexp13 | '/' valexp13 | '%' valexp13 )* ;

// array creation
valexp13: 'new' typeexp '[' valexp ']' | valexp14 ;

// unary operators
valexp14: '+' valexp14 | '-' valexp14 | '!' valexp14 | valexp15 ;

// selector operations
valexp15: valexp16 ( rselector ( rselector )* )? ;

// atoms
valexp16:
  IDENT | INT | 'true' | 'false' | STRING | CHAR | '(' valexp ')' ;

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

// class-level modifiers
modifiers: visibility ( 'final' visibility )? ;

// visibility modifiers
visibility: ( 'private' | 'protected' | 'public' )? ;

// throw declarations
throwdecls: ( 'throws' names )? ;
```

```

// a method's parameter list
params: param ( ',' param )* ;

// a method parameter
param: typeexp IDENT ;

// a type expression
typeexp: typeexpbase ( '[' ']' )? ;

// a type expression
typeexpbase: 'int' | 'boolean' | 'char' | name | IDENT ;

// a value expression list
valexps: ( valexpnull ( ',' valexpnull )* )? ;

// a name
name: ( IDENT '.' )* IDENT ;

// a sequence of names
names: name ( ',' name )* ;

// a location of a variable
lval: IDENT ( lselector )* ;

// an lvalue selector
lselector: '[' valexp ']' | '.' IDENT ;

// an rvalue selector
rselector: '[' v=valexp ']' | '.' 'getMessage' '(' ')' | '.' IDENT ;

// -----
// lexical syntax
// -----

IDENT : REALLETTER ( LETTER | DIGIT )* ;
INT : DIGIT ( DIGIT )* ;
STRING : '"' ( ~('"' | '\\\' | EOL) | ESCAPED )* '"' ;
CHAR : '\'' ( ~('\'' | '\\\' | EOL) | ESCAPED ) '\'' ;
WS : ( ' ' | '\t' | EOL ) ;
ANNOTATION:
  ( '//\' ( '@' .* EOL | .* EOL ) WS? )+
  | '/*\'' ( '@' .* '@*/' | .* '*/*' ) ;
REALLETTER : ( 'a'..'z' | 'A'..'Z' ) ;
LETTER : ( 'a'..'z' | 'A'..'Z' | '_' ) ;
DIGIT : ( '0'..'9' ) ;
EOL : ( '\n' | '\r' | '\f' | '\uffff' ) ;
ESCAPED : '\\\'
  ( '\\\' | '"' | '\'' | 'n' | 't' | 'b' | 'f' | 'r' |
    ( 'u' HEX HEX HEX HEX ) ) ;
HEX : '0'..'9' | 'a'..'f' | 'A'..'F' ;

```

H.2. Specification Language

```

// -----
// specifications
// -----

// a unit specification
unitspec:
  imports 'theory' ( 'uses' names )? '{' declarations '}' EOF ;

// a class specification
classspec: ( 'invariant' formula ';' )? EOF ;

// a method specification
methodspec:
  ( 'helper' ';' )?
  ( 'requires' formula ';' )?
  ( 'assignable' names ';' )?
  ( 'signals' names ';' )?
  ( 'ensures' formula ';' )?
  ( 'diverges' formula ';' )?
  ( 'decreases' term ( ',' term )* ';' )?
  EOF ;

// a loop annotation
loopspec:
  ( 'invariant' formula ';' )?
  ( 'decreases' term ( ',' term )* ';' )?
  EOF ;

// a command pre-state annotation
statementspec: ( 'assert' formula ';' )? ;

// a theory declaration
theorydecl:
  ( 'package' name ';' )? imports
  ( 'public' )* 'theory' IDENT ( 'uses' names )?
  '{' ( ( declaration )? ';' )* '}' EOF ;

imports: ( 'import' name ( '.' '*' )? ';' )* ;

declarations: ( ( declaration )? ';' )* ;

declaration:
  IDENT ':'
  ( 'TYPE'
  | 'TYPE' '=' typeExp
  | 'FORMULA' formula
  | 'AXIOM' formula
  | typeExp ( '=' term |
              '=' 'PRED' paramList ':' formula |
              '<=>' formula )?

```

```

    ) ;

typeExp:
( typeExpBase '->' typeExp
| '(' typeExp ( ',' typeExp )+ ')' '->' typeExp
| 'ARRAY' typeExpBase 'OF' typeExp
| typeExpBase
) ;

typeExpBase:
( name
| 'BOOLEAN'
| 'INT'
| 'NAT'
| 'REAL'
| 'STRING'
| 'STATE' ( '(' typeExp ')' )?
| '[' typeExp ( ',' typeExp )+ ']'
| '[' typeExp ']'
| '[' IDENT ':' typeExp ( ',' IDENT ':' typeExp )* '#]'
| 'SUBTYPE' '(' term ')'
| '[' term '..' term ']'
| 'PREDICATE' ( '(' typeExp ( ',' typeExp )* ')' )?
| '(' typeExp ')' ) ;

// -----
// formulas
// -----

// quantifiers bind weakest
formula:
( 'FORALL' paramList ':' formula
| 'EXISTS' paramList ':' formula
| formula10
) ;

// lets
formula10:
( 'LET' vdefinition ( ',' vdefinition )* 'IN' formula10
| formula20
) ;

// implications, equivalences, exclusive ors (= non-equivalences)
formula20:
( formula30 '=>' formula20
| formula30 ( '<=>' formula30 | 'XOR' formula30 )?
) ;

// disjunctions
formula30: formula40 ( 'OR' formula40 )* ;

// conjunctions

```

```

formula40: formula50 ( 'AND' formula50 )* ;

// logical negations
formula50: 'NOT' formula50 | formula60 ;

// equality and inequality and relations
formula60:
  term
  ( '=' term | '/=' term | '<' term | '<=' term | '>' term | '>=' term )
| formula70 ;

// atomic predicates
formula70:
  name '(' term ( ',' term )* ')'
| 'EXECUTES' '@' statearg | 'CONTINUES' '@' statearg
| 'BREAKS' '@' statearg | 'RETURNS' '@' statearg
| 'THROWS' '@' statearg | 'THROWS' '(' name ')' '@' statearg
| formula100 ;

// argument to state predicate
statearg: 'NOW' | 'NEXT' | name ;

// atoms
formula100:
  ( name | '(' 'OLD' name ')' | '(' 'VAR' name ')' )
  ( '.' ( NUMBER | IDENT ) | '[' term ']' )*
| 'TRUE' | 'FALSE'
| 'IF' formula 'THEN' formula
  ( 'ELSIF' formula 'THEN' formula )* 'ELSE' formula 'ENDIF'
| 'WRITESONLY' names | 'READSONLY'
| '(' formula ')'
;

// -----
// terms
// -----

// quantifiers bind weakest
term: 'LAMBDA' paramList ':' term | 'ARRAY' paramList ':' term | term10 ;

// lets
term10: 'LET' vdefinition ( ',' vdefinition )* 'IN' term10 | term20 ;

// sums and differences
term20: term30 ( '+' term30 | '-' term30 )* ;

// products and quotients
term30: term40 ( '*' term40 | '/' term40 )* ;

// power terms
term40: term50 ( '^' term50 )* ;

```



```

// unary arithmetic operators
term50: '+' term50 | '-' term50 | term60 ;

// updates
term60:
  term70
  ( 'WITH' ( '.' ( NUMBER | IDENT ) | '[' term ']' )+ ':= ' term70 )* ;

// selections
term70: term80 ( '.' ( NUMBER | IDENT ) | '[' term ']' )* ;

// applications
term80:
  'VALUE' '@' term100 | 'MESSAGE' '@' term100
  | term100 ( '(' term ( ',' term )* ')' )* ;

// atoms
term100:
  name | NUMBER | STRING | 'TRUE' | 'FALSE'
  | 'OLD' name | 'VAR' name | 'NOW' | 'NEXT'
  | '(' term ( ',' term )* ')'
  | '# IDENT := term ( ',' IDENT := term )* #'
  | 'IF' formula 'THEN' term
  ( 'ELSIF' formula 'THEN' term )* 'ELSE' term 'ENDIF' ;

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

paramList: '(' param[params] ( ',' param[params] )* ')' ;

param: IDENT ( ',' IDENT )* ':' typeExp ;

// value definition
vdefinition: IDENT ':' typeExp '=' term | IDENT '=' term ;

// a name
name: IDENT ( '.' IDENT )* ;

// a sequence of names
names: name ( ',' name )* ;

// -----
// lexical syntax
// -----

IDENT: REALLETTER (LETTER | DIGIT)* ;
NUMBER: DIGIT (DIGIT)* ;
STRING : '"' (~('"' | '\\\ ' | EOL) | ESCAPED )* '"' ;
REALLETTER: ('a'..'z' | 'A'..'Z') ;
LETTER: ( REALLETTER | '_' ) ;
DIGIT: ('0'..'9') ;

```

```
WS: (' ' | '\t' | EOL | COMMENT) { $channel=HIDDEN; };
EOL: ('\n' | '\r' | '\f');
COMMENT : '//' .* EOL | '/*' .* '*/' ;
ESCAPED : '\\\
  ( '\\\ | '\"' | '\'' | 'n' | 't' | 'b' | 'f' | 'r' |
    ('u' HEX HEX HEX HEX) ) ;
HEX : '0'..'9' | 'a'..'f' | 'A'..'F' ;
```