ProofPower

Compliance Tool — User Guide

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

List of Figures

ABOUT THIS PUBLICATION

0.1 Purpose

This document describes the use of the Compliance Tool supplied with ProofPower.

0.2 Readership

This document is intended to be read by users of the Compliance Tool. It contains both introductory material for the new user and reference material for more experienced users.

0.3 Related Publications

A bibliography is given on page 77 of this document.

- Installation of the Compliance Tool is described in:
  Compliance Tool — Installation and Operation [13]
- Advice on proving verification conditions generated by the tool is given in:
  Compliance Tool — Proving VCs [14]
- The syntax and semantics of the Compliance Notation as supported by the Compliance Tool is described in:
  Compliance Notation — Language Description [15]
- An overview of the of the Compliance notation can be found in the DRA document:
  A commentary on the Specification of the Compliance Notation for SPARK and Z [6].
- A description of ProofPower may be found in:
  ProofPower *Description Manual* [8],
  which also contains a full list of other ProofPower documentation.

0.4 Area Covered

Once the Compliance Tool is installed on the user's workstation, by following the procedure described in *Compliance Tool Installation and Operation [13]*, this User Guide should enable the user to undertake the following tasks:

1. Loading a Compliance Notation script into the tool;

2. Generation of the Ada program from a script;
3. Generation of the Z document from a script and reloading the Z document into ProofPower;
4. Preparation and checking of new scripts;
5. Using ProofPower facilities to work with VCs, e.g., to begin an attempt to prove a VC;
6. Evaluation of the rigour of a compliance argument by identifying the informal and formal parts of the argument and recognising potentially anomalous ways of using the facilities of the tool.

0.5 Prerequisites

This User Guide is *not* intended to be an introduction to the Z language, or to the Compliance Notation, or to the use of ProofPower to prepare the Z parts of a script.

Familiarity with the Compliance Notation and some familiarity with ProofPower are very desirable, although not essential for simple use of the tool to process existing scripts. Some familiarity with the use of ProofPower for developing Z specifications is required to use the tool to develop new scripts.

The Compliance Notation is described in *Compliance Notation — Language Description* [15]. There is also a formally specification of the notation in the DRA technical report [5]. The ProofPower-Z tutorial, [9], gives an introduction to the use of ProofPower for specification and proof in Z.

The ProofPower user documentation is supplied as part of the ProofPower release included with the Compliance Tool and is available for on-line reference.

The user interface to the tool described in this document is a Motif application running under the X Windows System. Users who are unfamiliar with X or Motif should consult local or supplier's documentation or expertise for further guidance (e.g. see [17]).

0.6 Acknowledgements

INTRODUCTION TO THE COMPLIANCE TOOL

1.1 The Compliance Notation

The Compliance Tool supports a notation for demonstrating the compliance of Ada programs with Z specifications. The notation is described in detail in *Compliance Notation — Language Description* [15]. The Ada parts of the notation belong to the Compliance Notation subset of Ada and will be referred to as Compliance Notation Ada in the rest of this document. The Compliance Notation was designed by the Defence Research Agency, Malvern and the ideas behind its design are discussed in the DRA document [6].

Compliance Notation is prepared and presented in literate scripts containing a mixture of narrative text, Z and Compliance Notation Ada. Special constructs are provided for two main purposes:

1. to allow the Ada program to be presented and documented in an order independent of that prescribed by the Ada syntax (in a similar fashion to Knuth's Web system [2]);
2. to allow the behaviour of the Ada code to be formally specified in Z (using a style based on Morgan's refinement calculus [3]).

These constructs allow the user to assert formal connections between the Z parts and the Ada parts in a Compliance Notation script and permit the compliance of the Ada program against its Z specifications to be reduced to a set of conjectures known as verification conditions, whose truth guarantees compliance. The VC generation algorithm is formally specified in the DRA technical report [5] and explained by example in *Compliance Notation — Language Description* [15]. The algorithm maps a Compliance Notation script to a Z document containing the VCs formulated with Z conjectures together with supporting Z definitions.

1.2 The Compliance Tool

The Compliance Tool described in this document supports the use of the Compliance Notation. It performs the following principal functions:

1. Checking the syntax of a Compliance Notation script;
2. Generating the Z document including VCs from a script;
3. Extracting the Ada program from a script.

The Compliance Tool is implemented as an application of the ProofPower specification and proof tool. The tool provides all the facilities offered by ProofPower for developing specifications and proofs in HOL and Z and for preparing high quality printed output via the LaTeX system. The tool

includes custom facilities for working with Compliance Notation including, in some installations an interactive tool called the VC browser and some customised proof procedures.

The use of the tool in the development of a compliance argument will involve several stages, typically including the following:

1. Initial preparation of the literate scripts using the editing and interactive checking facilities of the tool. During this stage, the Z documents and Ada program are produced and inspected as required to assist in the development.
2. Batch processing of the complete literate scripts. If the conventions suggested in this User Guide are followed, the Z documents and Ada program are produced automatically. The tool can be used interactively if required, e.g., to help diagnose errors.
3. Further analysis of the Z documents. Depending on circumstances and on the level of formality required, this might involve either or both of: *(a)*, inspection of the documents, on paper or using a viewer; *(b)*, use of the VC browser supplied with the tool; and, *(c)*, machine-checked proof of some or all of the VCs.

The Compliance Tool offers a range of facilities to help in all of these activities.

GETTING STARTED

The task of using of the Compliance Tool to process a script is very similar to that of using ProofPower to process a Z specification. Indeed, the Z paragraphs in a Compliance Notation script are handled using the usual ProofPower mechanisms and the constructs peculiar to the Compliance Notation are treated in a very similar way.

Simple use of the tool to process existing scripts does not require any great familiarity with ProofPower. Using the tool to create new scripts or edit existing ones requires some familiarity with the preparation of Z specifications using ProofPower (as described in [10]). The rest of this chapter explains how to make a start with the tool for either of these levels of use. Section 2.1 describes set-up procedures which are common to batch and interactive working; section 2.2 describes how to use the tool to process existing scripts using UNIX commands; section 2.3 explains how to start an interactive session with the tool and gives an overview of its main graphical features.

2.1 ProofPower Databases

As always with a ProofPower application, to use the Compliance Tool you must first pick the ProofPower database with which you wish to work. A ProofPower database is a file which holds code and data recording the results of work with ProofPower. These databases are organised in a hierarchy. To run the Compliance Tool, you use a database which is a descendant in this hierarchy of the database `pp_daz`. This database, held in a file called `sun4bin/sun4pp_daz.db` in the ProofPower installation directory, contains the code and data required to process Compliance Notation. If you are starting from scratch, you would create a suitable database, say called `mydatabase`, using a UNIX command line such as:

```
pp_make_database -p daz mydatabase
```

This creates an empty database, which you will generally refer to as '`mydatabase`' when using the system. The database is held in a file called '`sun4mydatabase.db`' and this is the name you would use to carry out UNIX file operations on the database, e.g., to delete it with the UNIX command '`rm`'.

2.2 Batch Working

Providing the conventions of chapter 3 have been followed in the development of the scripts, one or more literate scripts may be processed using the programs `docsml` and `pp` to produce the Z documents and Ada program. For example, assuming a database `mydatabase` has been set up as described above, the following UNIX commands will process the script in file `wrk501.doc`.

```
docsml wrk501
pp -f wrk501 -d mydatabase >wrk501.run.log
```

Here the first command produces a file called `wrk501.sml`, containing the formal material from `wrk501.doc` without the narrative text. The second command causes the Compliance Tool proper to process `wrk501.sml`, updating the database `mydatabase` and producing the Z document and Ada program for the script. If the conventions of chapter 3 have been followed in the development of the script, the Z document will be in a file called `wrk501.zdoc` and the Ada program will be in `wrk501.ada`. The log produced as the standard output from `pp` has been directed to a file `wrk501.run.log`, since it is likely to contain non-ASCII characters.

In a compliance argument involving more than one script, the `docsml-pp` command sequence above is entered in turn for each script. Note that the order of the scripts is important, and providers of compliance arguments must indicate the required order (e.g., by supplying a make file or a shell script containing the necessary sequence of commands).

The Ada program is an Ada source file which may be examined with any text editor. The Z document contains non-ASCII characters and is best examined either using the `xpp` editor or using the program `docpr` to print it. The log file is best examined in the same way. The following UNIX command will print the Z document in our example:

```
docpr wrk501.zdoc
```

Basic use of the `xpp` editor is described in section 2.3.1. To use it to examine the Z document in this example, one would use the UNIX command:

```
xpp -file wrk501.zdoc
```

2.3 Interactive Working

2.3.1 Starting `xpp`

To start an interactive session with the Compliance Tool, you use the UNIX command `xpp`. `xpp` comprises a custom editor for working on your scripts together with a command interface for executing ProofPower commands and for loading Z and Compliance Notation constructs into the database.

The `xpp` command has options to identify the file containing the script you want edit and to give the ProofPower command options (including the database to use). To begin work on the script '`wrk501.doc`' with the database '`mydatabase`', you might use the UNIX command line:

```
xpp -file wrk501.doc -command pp -d mydatabase
```

This will create a Motif window similar to the one shown in figure 2.1. This is referred to as the `xpp` Main Window. If you omit the `-command` option, then you will get a so-called edit-only session, in which the lower part of the window will be missing.

2.3.2 Window Layout

This section describes briefly the main elements of the `xpp` Main Window. Further information may be obtained either from the tool's help system or from the ProofPower documentation.

In overview, the top part of the window acts as an editor for literate scripts. Material from the editor can be transferred into the ProofPower-ML system for processing (e.g. to type check a specification paragraph or initiate a proof step). The journal of the transactions with ProofPower occupies the bottom part of the window. Figure 2.1 also shows a popup window (in this case a palette of special symbols). The size and position of all the windows can be adjusted using standard Motif techniques.

Sections 2.3.2.1 to 2.3.2.4 below describe the principal features of the xpp Main Window.

2.3.2.1 Menu Bar

The menu bar is at the top of the window shown in figure 2.1. The menus are all pulldown menus operated in the usual Motif fashion. The functions they perform are briefly described below.

File Menu This menu is for common file operations including: loading and saving the literate script; creating a new literate script; deleting a file.

Tools Menu This menu is used to create popup windows to perform various tasks. An example is the palette window shown in figure 2.1.

Edit Menu This menu provides script editing operations: 'Cut', 'Copy', 'Paste' and 'Undo'.

Command Menu This menu is primarily used to cause text from the script window to be executed by the ProofPower-ML system. It also provides various control functions for the ProofPower-ML system.

Help Menu This menu provides help on various topics. Its 'Tutorial' item may be consulted for more information on basic operation of xpp.

2.3.2.2 File Name Bar

This displays the name of the file containing the script which is being edited in the Script Window.

2.3.2.3 Script Window

The Script Window is the upper large Motif text window in the xpp Main Window. It provides a general purpose editor for viewing and modifying scripts. In figure 2.1 on page 12, a Compliance Notation script containing definitions of a Z global variable, '*primed*', and an Ada procedure, '*primes*', is being displayed in the Script Window.

Part of the Script Window in figure 2.1 is hidden behind a window containing a 'palette' of symbols. This palette has been selected from the Tools Menu and can be used to enter the symbols into the literate script. Pushing one of the buttons on the palette causes the symbol to appear in the document as if it had been typed at the keyboard.

Note that the Z and Ada constructs in the Script Window are delimited using special mark-up sequences: 'Ⓢ*Z*", 'Ⓢ*CN*' and '■'. A tool called the Templates Tool is available from the Tools Menu allowing easy entry of these mark-up sequences.

2.3.2.4 Journal Window

The Journal Window is the lower large Motif text window in the xpp Main Window. When ProofPower commands are executed the resulting responses from the ProofPower system are displayed

in the Journal Window. The Journal Window is a read-only text window — you cannot alter its contents by typing into it.

In 2.1, the user has just issued a command to process the procedure *primes*. This is done by selecting the procedure and its delimiting mark-up sequences in the Script Window and then using the 'Execute' item in the Command Menu. The absence of error messages in the Journal Window and the prompt (':>') indicate that the procedure has been processed and accepted.

Figure 2.1: An Example Compliance Tool Session

LITERATE SCRIPT CONVENTIONS

Literate scripts are held in UNIX files, each file holding one or more scripts. Typically, each script contains exactly one compilation unit (such as a package specification or package body) together with any Z definitions needed to support the specification of that compilation unit. A script need not contain any Ada code at all, in which case the script just provides Z definitions for use in other scripts. A script may not contain more than one compilation unit.

Processing a script has three main goals: production of a Z document; production of the Ada program (i.e., the source code contained in the script); and modifications to the state of a database maintained by the tool and used in processing subsequent scripts. The literate scripts making up a complete compliance argument must be presented to the tool in some order, compatible with any dependencies between the scripts.

In addition to narrative text, Z paragraphs and Compliance Notation clauses, each script contains commands which direct the tool, e.g., to tell it to output the Ada program to a particular file. In this chapter, we recommend some conventions for these commands which are intended to make the script easy to process both interactively and in batch. A complete example of a script following these conventions is given in chapter 5.

The conventions described below are not enforced by the tool, and some users may wish to adapt them to their own needs. However, other sections of this user guide are written on the assumption that these conventions have been followed (e.g., section 2.2). The conventions are designed to make compliance arguments easy to assess. If you need to adopt different conventions, please refer to section 6 for further guidance on assessing a compliance argument.

3.1 Initialisation

The formal material in each literate script should be preceded by an initialisation section giving a name to the script in which it is to be processed. This is done with a ProofPower-ML command, *new_script*. Execution of *new_script* introduces a new ProofPower theory whose name is the name of the script and prepares the Compliance Tool to accept an Ada compilation unit. For example:

```
new_script{name = "Utils", unit_type = "spec"};
```

Here the name is the name of the compilation unit and *unit_type* indicates its type, which must be one of "*spec*" (package specification), "*body*" (package body), "*proc*" (procedure) or "*func*" (function).

Dependencies on ProofPower theories that come from Ada constructs such as context clauses are handled automatically by the tool. However, in some circumstances, you may need to inform the tool that a script depends on one or more theories in the initialisation for the script. This might happen if you have an existing library of Z theories containing definitions that you want to use in the script. A variant of *new_script*, called *new_script1* allows you to give a list of *library theories* that become parents of the script theory. For example, if *script3* depends on the Z paragraphs stored in *mytheory1* and *mytheory2*, then you can use the command:

```
new_script1{name = "Utils", unit_type = "spec",
        library_theories=["mytheory1", "mytheory2"]};
```

In some circumstances, you may want to write Z paragraphs referring to the types, constants or functions in a package, *P* say, before giving the Ada compilation unit containing the context clause identifying *P* as a dependency. In this case, you can identify the package specification theory, *Pspec*, as a library theory in the script initialisation.

3.2 Theory Hierarchy

For each compilation unit, the corresponding *new_script* (or *new_script1* command introduces a ProofPower theory. For example, associated with a package body `utils` there will be a theory called *UTILS'body*.

For each each subprogram body inside a package or subprogram body, and for each package or subprogram stub, the Compliance Tool automatically creates a ProofPower theory whose name is derived from the expanded name of the package or subprogram. These theories are referred to as subprogram or stub theories.

A subprogram or stub theory is created with an automatically generated parent theory which is a duplicate of the theory associated with the enclosing body as it was at the point where the subprogram body or stub was processed (but with any VCs removed). This parent theory is called the context theory for the subprogram or stub. For example, if a package body *Utils* contains a procedure *sort*, a context theory called *UTILSoSORT'context* is created as a duplicate of the the script theory *UTILS'body*

The subprogram theories are used to hold the Z paragraphs representing the types and constants defined in the declarative part of the subprogram body and to hold VCs associated with the statement part. The Compliance Tool requires you to work in the appropriate theory when processing Compliance Notation clauses. The command *open_scope*, which takes as its argument the expanded name of the subprogram or package, is used to do this. For example, to refine a specification statement in the body of the procedure *sort* in the example mentioned above, you would need to make the following call:

```
open_scope "Utils.sort";
```

Stub theories are used to provide a context for processing the corresponding subunit. When a subunit is processed, the stub theory is automatically made a parent of the script theory.

For example, if the package `utils` contains the body of a procedure called `sort`, the tool will create a subprogram theory called *UTILSoSORT'proc*. If the package `utils` contains a stub for a package called `strings`, the tool will create a stub theory called *UTILSoSTRINGS'stub*. The list of all the theories associated with a script can be found using the function *get_script_theories*. For example:

```
get_script_theories "UTILS'body";
```

might result in the following output:

Compliance Tool Output

```
val it = ["UTILS'body", "UTILSoSORT'proc", "UTILSoSTRINGS'stub"] : string list
```

3.3 Output Commands

The formal material in a literate script should be followed by directions for the production of the Z document and of the Ada program. This should be as in the following example:

```
output_z_document{script="UTILS'spec", out_file="utils_ads.zdoc"};
output_ada_program{script="UTILS'spec", out_file="utils_ads.ada"};
```

Here *UTILS'spec* is the script name, and conventional file name suffixes have been added to it for the Z document and the Ada program.

COMPLIANCE TOOL FUNCTIONS

Sections 4.1 to 4.7 below describe basic use of the main functions of the Compliance Tool.

You invoke many of these functions by executing ProofPower-ML commands. This may conveniently be done interactively using the Command Line Tool which can be started using the Tools Menu. An instance of the Command Line Tool may be seen in figure 4.1 on page 20. This tool allows single line ProofPower-ML commands to be entered and executed without changing the script which is being edited. The tool also has a scrollable list which you can use as a memory for common commands.

Some functions are provided as UNIX commands or as a combination of UNIX and ProofPower-ML commands.

4.1 Loading Scripts

To load a script, the UNIX command `docsml` must first be used to extract the formal material. For example, the UNIX command:

```
docsml wrk501
```

copies the formal material from the file `wrk501.doc` into a file called `wrk501.sml`.

The `.sml` file produced by `docsml` may then be loaded using the ProofPower-ML command:

```
use_file "wrk501";
```

By default, the tool will raise an exception and stop processing a script if it detects an error. However, while preparing a compliance argument for a large program, it can be convenient to have the tool continue processing when errors are detected. This can be controlled using the flag *cn_stop_on_exceptions*. The tool maintains a record of the errors that have been detected which can be manipulated using the ProofPower-ML commands, *print_exception_log*, *output_exception_log* and *delete_exception_log*. See section 7 for more information.

4.2 Generating the Ada Program

The Ada program is output to a file using the ProofPower-ML command *output_ada_program*. This command is normally included at the end of each literate script as described in section 3.3 and may also be called at any time during the development of the script.

A function *print_ada_program* is also provided for displaying the compilation units os the Ada program on the standard output. It takes a string parameter giving the name of the script in which the compilation unit was introduced. A minus sign may be used as a short-hand for the name of the current script. For example, in an interactive session with `xpp`, the ProofPower-ML command:

```
print_ada_program"-"
```

will cause the Ada program for the current script to be displayed in the journal window.

Some errors, in particular illegal redeclaration of names, are not detected until the Ada program is generated, so generation of the Ada program is an important part of processing a script. Even if these errors are detected, the Ada program is displayed or output to the file before the errors are reported. This allows a script containing informally developed code in which redeclaration of names is required to be processed by the tool (some redeclarations are legal Ada, but not legal Compliance Notation Ada).

When loading scripts, detection of an error will normally cause the tool to raise an exception and stop processing. You can override this behaviour selectively by using a ProofPower-ML exception handler. For example, the following ProofPower-ML command will output the Ada program to the file `SCRIPT.ada` and continue without raising an exception even if the program contains illegal redeclarations:

```
output_ada_program{script="-"=, out_file="SCRIPT.ada"}
        handle Fail _ => ();
```

The subset of Ada accepted by the Compliance Tool includes a number of features, e.g., use clauses, that are not supported in the SPARK subset of Ada (see [1] for the definition of the SPARK subset of Ada). You may ask for Ada comments highlighting constructs that do not conform to the SPARK syntax to be inserted in the Ada program uinsg the flag *cn_spark_syntax_warnings*. See section 7 for more information.

4.3 Generating and Reloading the Z document

The Z document is output to a file using the ProofPower-ML command *output_z_document*. This command is normally called at the end of each literate script as described in section 3.3 and may also be called at any time during the development of the script.

A function *print_z_document* is also provided for displaying the Z document on the standard output. For example, in an interactive session with `xpp`, the ProofPower-ML command:

```
print_z_document "SCRIPT";
```

will cause the Z document for the script called *SCRIPT* to be displayed in the journal window.

In many situations, there is no actual need to reload the Z document, since it is automatically loaded as a a side-effect of processing the literate script. Thus work on proof of VCs, for example, can begin immediately after the script is processed. The Z document can be reloaded into the tool, if required, simply by treating it in the same way as a `.sml` file produced by running `docsml` on an ordinary Z specification. Thus one can use either a ProofPower-ML command such as:

```
use_file"SCRIPT.zdoc";
```

or a UNIX command such as:

```
pp -f SCRIPT.zdoc -d database
```

to load in the Z document. The following points should be noted in connection with reloading Z documents produced by the Compliance Tool:

1. the Z document requires the presence of the theory *cn*, which is not provided as standard in ProofPower-Z databases so that `database` in the example above should refer to a Compliance Tool database;

2. loading the Z document attempts to create ProofPower theories which will already exist in a database in which the literate script giving rise to the Z document has already been processed (so that these theories must be deleted, or a fresh Compliance Tool database created, if the loading is to succeed);

3. loading the Z document does not recreate the information about the Ada program which is computed and stored when the literate script is processed (so that facilities such as *print_ada_program* which require this information will no longer be useful).

4.4 Inspecting the VCs

Figure 4.1 on page 20 gives another example of an `xpp` Main Window. In this example, the user has used the Command Line Tool to issue commands to load an entire script into the tool and then display the Z document in the Journal Window. The Search-and-Replace Tool has been used to carry out a textual search in the Script Window for the refinement step which introduced the VC at the end of the Z document.

Using an editor to correlate the script with the VCs can be a useful technique but is often time consuming. The VC Browser is an interactive tool for examining the VCs and relating them to the Compliance Notation clauses which produced them. The VC Browser is provided in the ProofPower database `xdaz` on implementations of the Compliance Tool built with the Poly/ML compiler. To use it you must ensure that X Windows and Motif support is compiled into the Poly/ML driver program `poly`. See the Poly/ML documentation for more information.

To use the VC browser, you carry out your Compliance Notation work in an X Windows environment using the `xdaz` database rather than the `daz` database. I.e., using `pp -d xdaz` rather than `pp -d xdaz`. The VC Browser is started by running the ProofPower-ML command '*browse_vcs*()'. This brings up a window which can be used to inspect the VCs that have been generated and the Compliance Notation clauses which gave rise to them. Press the VC Browser's "Help" button for more information.

Figure 4.1: Inspecting VCs with `xpp`

4.5 Editing and Checking a Script

The xpp Script Window provides a general purpose editor for use in preparing scripts. The ProofPower document [7] or the xpp help system may be consulted for an introduction to its use. The easiest way to begin constructing a new script is to adapt an existing one, and a number of examples are supplied with the Compliance Tool for this purpose.

Just as with a ProofPower-Z specification, a Compliance Notation script may be entered a clause at a time so that errors can be detected and corrected interactively. Note however, that the initialisation section of the script as described in section 3.1 has to be entered first of all, and that clauses will generally need to be entered in the correct order.

If a Compliance Notation clause has been rejected because of an error, the clause can normally be re-entered once the error has been corrected. However, if a clause has been accepted, it cannot normally be re-entered. If you have loaded a script and wish to modify and reload it, the function *delete_script* may be used to avoid having to start again from scratch. *delete_script* takes as its parameter the name of the script to be deleted. It prints out a report of what it has deleted to help you determine what must be reloaded. A call of *delete_script* would normally be followed by a *new_script* command to begin work on the revised version of the script.

To make entry of Compliance Notation constructs easier, the Templates Tool in xpp has been customised so that you can enter a template for any of the following constructs with one button-press:

1. Replacement Step;
2. Refinement Step;
3. Compilation Unit;
4. Specification Statement;
5. Each of the 8 Z Paragraph forms.

The customised Templates Tool may be seen in figure 4.2 on page 22. The four buttons at the bottom are for the four Compliance Notation constructs listed above. In the figure the user has just pressed the refinement step button and has started to fill in the template at the bottom of the Script Window — the label has been filled in and the ellipsis, '...', has been selected ready to be replaced by the statement part of the refinement.

4.6 Accessing and Proving VCs

VCs are represented in the tool using Z's conjecture paragraph form. A conjecture paragraph associates a name with a Z predicate, and is used in specifications to record properties which the specifier believes to be true of the objects specified. In ProofPower the function *get_conjecture* retrieves a conjecture from a specification by name. The conjecture is retrieved as a Z term which can be used, e.g., to form a goal for proof with the ProofPower Subgoal Package.

The tool assigns a name to each Compliance Notation clause in the script being processed and this name is used as the basis for the names of any VCs generated by the clause. In the case of clauses containing a package or similar named Ada object, the name is taken from the object name; for other clauses the name is derived from the implicit or explicit numeric label associated with the clause.

Examples of VC names can be seen in 4.1 on page 20. Here the clause name *5* on the left refers to the refinement step refining label *5* as shown in the following extracts from the file wrk501.doc:

Figure 4.2: Compliance Notation Templates Tool

Compliance Notation

$$\sqsubseteq$$
$$\Delta\ MULT[mult\_inv,\ mult\_inv \wedge MULT\ (N) \geq J]$$
$$\Delta\ JPRIME\ [mult\_inv \wedge MULT\ (N) \geq J,$$
$$mult\_inv \wedge MULT\ (N) \geq J \wedge JPRIME = MULT\ (N)\ noteq\ J] \quad (5)$$

Compliance Notation

$$(5) \sqsubseteq$$

$$jprime := mult(n)\ /= j;$$

The VCs corresponding to the clause named *5* are named *vc5_1*, *vc5_2*, etc. (in this case there is only one). A clause name like *4_11* is used for the *11*-th unlabelled refinement step in the script, with the *4* indicating that this refinement step comes after the one labelled *4*.

To access the VC in the example, one would use the ProofPower-ML function call:

```
get_conjecture "PRIMES'proc" "vc5_1";
```

Here the first parameter gives the name of the literate script which gave rise to the VC. Proof of a VC will normally need to be carried out within the theory corresponding to the literate script in question. Thus, to begin work on a proof of the VC with the Subgoal Package, the following commands would be used:

```
open_theory "PRIMES'proc";
set_goal([], get_conjecture "PRIMES'proc" "vc5_1");
```

As ordinary Z goals, VCs may be proved using all of the normal facilities provided by ProofPower for proof in Z. Some extensions to the Z toolkit are used in many VCs. These extensions are contained in the theory *cn*. Some custom support is provided in the Compliance Tool to assist with reasoning in this theory, most notably the proof contexts *cn1* and *cn1_ext*. These proof contexts (amongst others) and the other custom proof tools are described in section 7.2 of this document. The user interface to the conversions etc. described there is via the proof contexts, so that most users will only need to be familiar with the proof contexts. An example of a complete proof script for a literate script is included in chapter 5

4.7 Printing and Previewing Scripts

You can produce a LaTeX document from a Compliance Notation script using ProofPower's document preparation facilities, which are described in detail in ProofPower *Tutorial* [7]. The UNIX command `doctex` is used to produce the LaTeX file which you can then run through LaTeX using the UNIX command `texdvi`. You can then print or preview the resulting DVI file using the programs provided with your LaTeX installation.

So for example, to print the script in file `wrk507.doc` on a PostScript printer, you might use the following UNIX commands:

```
doctex wrk507
texdvi wrk507
dvips wrk507
```

You can also generate a DVI file with embedded hypertext links, which help you relate k-slots and specification statements with the corresponding replacement steps and refinement steps. To do this, you use a Compliance Tool function which generates an edit script which can be used by `doctex` to make the links. For example, the example script `wrk507.doc`, contains the following ProofPower-ML command at the end:

```
output_hypertext_edit_script{out_file="wrk507.ex"};
```

When the script has been loaded, the DVI file with hypertext links can be produced using the UNIX commands:

```
doctex -e wrk507.ex wrk507
texdvi wrk507
```

The resulting DVI file, `wrk507.dvi`, can then be viewed with a hypertext viewer such as `xhdvi`. You can navigate around the script by clicking on the underlined arrows which appear in the margins near k-slots, specification statements, refinement steps and replacement steps.

EXAMPLE SCRIPT

In this chapter, we give an example literate script, following the conventions recommended in chapter 3. The script shows the initial part of the compliance argument for an Ada procedure for computing integer square roots.

Section 5.1 contains the script and associated ProofPower commands. Sections 5.2 and 5.3 shows the corresponding Z document and Ada Program. Section 5.4 contains proofs of the VCs produced by the script and section 5.5 shows the listing of the resulting ProofPower theory.

5.1 The Literate Script

5.1.1 Initialisation Commands

We give the script the name *SQRT'proc* following the conventions for a script containing a compilation unit comprising a function named *SQRT*:

SML

```
new_script{name = "SQRT", unit_type = "proc"};
```

5.1.2 The Compliance Argument

For simplicity, we present the square root function as a top-level procedure. The procedure has a specification statement requiring that for non-negative input values of X, the output value of Y is the integer part of the square root of X.

Compliance Notation

```
procedure SQRT (X : INTEGER; Y : out INTEGER)

Δ Y [X ≥ 0, Y ** 2 ≤ X < (Y + 1) ** 2]

is
   LO : INTEGER;

   ⟨ local vars ⟩                (2)

begin
   LO := 0;

    Δ LO [X ≥ 0 ∧ LO = 0, LO ** 2 ≤ X < (LO + 1) ** 2]

```

$Y := LO;$
end SQRT;

Compliance Notation

$(2) \equiv$

$HI : INTEGER;$

Compliance Notation

$\sqsubseteq$
$\Delta\ LO, HI\ [X \geq 0 \wedge LO = 0, LO ** 2 \leq X < (LO + 1) ** 2]$

Compliance Notation

$\sqsubseteq$

$HI := X + 1;$

$\$till\ [\![LO ** 2 \leq X < (LO + 1) ** 2]\!]$

loop

$\Delta\ LO, HI\ [LO ** 2 \leq X < HI ** 2, LO ** 2 \leq X < HI ** 2]$

end loop;

Compliance Notation

$\sqsubseteq$

exit when $LO + 1 = HI$;

$\Delta\ LO, HI\ [LO ** 2 \leq X < HI ** 2, LO ** 2 \leq X < HI ** 2]$

Note the development is not complete at this point. However, we can still output the Z document and the Ada program as if it were, and begin to do proofs. The fact that the development is not complete is made manifest in the Ada program which will contain null statements corresponding to the omissions, see section 5.3.

5.1.3 Output Commands

SML

$output\_z\_document\{script="SQRT'proc",\ out\_file="usr501.zdoc"\};$
$output\_ada\_program\{script="SQRT'proc",\ out\_file="usr501.ada"\};$

5.2 The Z Document

The following shows the Z document produced by the example script.

SML

```
new_theory "SQRT'spec";
```

z

$$vcSQRT\_1 \ ?\vdash \forall X : INTEGER \mid X \geq 0 \bullet X \geq 0 \land 0 = 0$$

z

$$vcSQRT\_2 \ ?\vdash \forall LO : INTEGER; X : INTEGER \mid X \geq 0 \land LO ** 2 \leq X \land X < (LO + 1) ** 2 \bullet LO ** 2 \leq X \land X < (LO + 1) ** 2$$

z

$$vc2\_1\_1 \ ?\vdash \forall LO : INTEGER; X : INTEGER \mid X \geq 0 \land LO = 0 \bullet X \geq 0 \land LO = 0$$

z

$$vc2\_1\_2 \ ?\vdash \forall LO, LO_0 : INTEGER; X : INTEGER \mid (X \geq 0 \land LO_0 = 0) \land LO ** 2 \leq X \land X < (LO + 1) ** 2 \bullet LO ** 2 \leq X \land X < (LO + 1) ** 2$$

z

$$vc2\_2\_1 \ ?\vdash \forall LO : INTEGER; X : INTEGER \mid X \geq 0 \land LO = 0 \bullet LO ** 2 \leq X \land X < (X + 1) ** 2$$

z

$$vc2\_2\_2 \ ?\vdash \forall HI : INTEGER; LO, LO_0 : INTEGER; X : INTEGER \mid (X \geq 0 \land LO_0 = 0) \land LO ** 2 \leq X \land X < HI ** 2 \bullet LO ** 2 \leq X \land X < HI ** 2$$

z

$$vc2\_2\_3 \ ?\vdash \forall LO, LO_0 : INTEGER; X : INTEGER \mid (X \geq 0 \land LO_0 = 0) \land LO ** 2 \leq X \land X < (LO + 1) ** 2 \bullet LO ** 2 \leq X \land X < (LO + 1) ** 2$$

z

$$
\begin{array}{l}
\mid\ vc2\_3\_1\ ?\vdash \\
\mid\quad \forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \\
\mid\qquad \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = TRUE \\
\mid\qquad \bullet\ LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2
\end{array}
$$

z

$$
\begin{array}{l}
\mid\ vc2\_3\_2\ ?\vdash \\
\mid\quad \forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \\
\mid\qquad \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = FALSE \\
\mid\qquad \bullet\ LO\ **\ 2 \leq X \wedge X < HI\ **\ 2
\end{array}
$$

z

$$
\begin{array}{l}
\mid\ vc2\_3\_3\ ?\vdash \\
\mid\quad \forall\ HI,\ HI_0 : INTEGER;\ LO,\ LO_0 : INTEGER;\ X : INTEGER \\
\mid\qquad \mid (LO_0\ **\ 2 \leq X \wedge X < HI_0\ **\ 2) \wedge LO\ **\ 2 \leq X \wedge X < HI\ **\ 2 \\
\mid\qquad \bullet\ LO\ **\ 2 \leq X \wedge X < HI\ **\ 2
\end{array}
$$

SML

$$
\mid (*\ Number\ of\ VCs\ in\ theory\ "SQRT'spec"\ :\ 10\ *)
$$

5.3 The Ada Program

The following shows the Ada program produced by the example script. Note the *NULL* statement the tool has introduced because the development was not complete. The comment on the *NULL* statements means that the unlabelled specification statement to which the tool has assigned label *2_4* has not been refined.

```
PROCEDURE SQRT (X : IN INTEGER; Y : OUT INTEGER)
  -- Spec ...
IS
  LO : INTEGER;
  HI : INTEGER;
BEGIN
  LO := 0;
  HI := X + 1;
  -- ...
  LOOP
    EXIT WHEN LO + 1 = HI;
    NULL; -- 2_4
  END LOOP;
  Y := LO;
END SQRT;
```

5.4 The Proofs

To embark on the proofs of the VCs, we first open the theory for the literate script:

SML
```
open_theory"SQRT'spec";
```

We will work in the proof context *cn1* most of the time:

SML
```
set_pc"cn1";
```

The statements of the VCs may be seen in the Z document in section 5.2 or in the theory listing in section 5.5.

All but one of the VCs are little more than tautologies, and will be proved by repeating *strip_tac*.

SML
```
set_goal([], get_conjecture "SQRT'spec" "vcSQRT_1");
a(REPEAT strip_tac);
val vcSQRT_1_thm = save_pop_thm"vcSQRT_1_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vcSQRT_2");
a(REPEAT strip_tac);
val vcSQRT_2_thm = save_pop_thm"vcSQRT_2_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_1_1");
a(REPEAT strip_tac);
val vc2_1_1_thm = save_pop_thm"vc2_1_1_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_1_2");
a(REPEAT strip_tac);
val vc2_1_2_thm = save_pop_thm"vc2_1_2_thm";
```

Before proving the next VC, we need to prove some simple facts about exponientation, namely: $x ** 1 = x$ and $x ** 2 = x * x$. The proofs require little more than specialising the definition of $(\_**\_)$ appropriately.

SML
```
set_goal([], ⌜∀x: ℤ• x ** 1 = x⌝);
a(REPEAT strip_tac);
a(rewrite_tac[rewrite_rule[](
    z_∀_elim⌜(x ≙ x, y ≙ 0)⌝ (∧_right_elim(z_get_spec⌜(_**_)⌝)))]);
val star_star_1_thm = pop_thm();
```

SML
```
set_goal([], ⌜Z∀x: ℤ•  x ** 2 = x * x⌝);
a(REPEAT strip_tac);
a(rewrite_tac[star_star_1_thm, rewrite_rule[](
    z_∀_elim⌜Z(x ≙ x, y ≙ 1)⌝ (∧_right_elim(z_get_spec⌜Z(_**_)⌝))))]);
val star_star_2_thm = pop_thm();
```

The meat of the next VC is that for non-negative X, $X \leq (X + 1) ** 2$. Induction is used to prove this, together with the above lemmas and the automatic prover for linear arithmetic.

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_2_1");
a(REPEAT strip_tac THEN asm_rewrite_tac[star_star_2_thm]);
a(DROP_NTH_ASM_T 2 ante_tac THEN DROP_ASMS_T discard_tac THEN strip_tac);
a(z_≤_induction_tac⌜ZX⌝ THEN PC_T1 "z_lin_arith" asm_prove_tac[]);
val vc2_2_1_thm = save_pop_thm"vc2_2_1_thm";
```

The remaining VCs are straightforward:

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_2_2");
a(REPEAT strip_tac);
val vc2_2_2_thm = save_pop_thm"vc2_2_2_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_2_3");
a(REPEAT strip_tac);
val vc2_2_3_thm = save_pop_thm"vc2_2_3_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_3_1");
a(rewrite_tac[]);
a(REPEAT strip_tac);
a(all_var_elim_asm_tac1);
val vc2_3_1_thm = save_pop_thm"vc2_3_1_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_3_2");
a(REPEAT strip_tac);
val vc2_3_2_thm = save_pop_thm"vc2_3_2_thm";
```

SML
```
set_goal([], get_conjecture "SQRT'spec" "vc2_3_3");
a(REPEAT strip_tac);
val vc2_3_3_thm = save_pop_thm"vc2_3_3_thm";
```

5.5 The Theory Listing

The following is the listing of the theory *usr501* produced by the example script and the proofs in the previous section.

5.5.1 Parents

cache′daz *cn*

5.5.2 Conjectures

vcSQRT_1

$$\forall\ X : INTEGER \mid X \geq 0 \bullet X \geq 0 \wedge 0 = 0$$

vcSQRT_2

$$\forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_1_1

$$\forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO = 0 \bullet X \geq 0 \wedge LO = 0$$

vc2_1_2

$$\forall\ LO, LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_2_1

$$\forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO = 0 \bullet LO\ **\ 2 \leq X \wedge X < (X + 1)\ **\ 2$$

vc2_2_2

$$\forall\ HI : INTEGER;\ LO, LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < HI\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

vc2_2_3

$$\forall\ LO, LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_3_1

$$\forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = TRUE \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_3_2

$$\forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = FALSE \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

vc2_3_3

$$\forall\ HI, HI_0 : INTEGER;\ LO, LO_0 : INTEGER;\ X : INTEGER \mid (LO_0\ **\ 2 \leq X \wedge X < HI_0\ **\ 2) \wedge LO\ **\ 2 \leq X \wedge X < HI\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

5.5.3 Theorems

vcSQRT_1_thm

$$\vdash \forall\ X : INTEGER \mid X \geq 0 \bullet X \geq 0 \wedge 0 = 0$$

vcSQRT_2_thm

$$\vdash \forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_1_1_thm

$$\vdash \forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO = 0 \bullet X \geq 0 \wedge LO = 0$$

vc2_1_2_thm

$$\vdash \forall\ LO,\ LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_2_1_thm

$$\vdash \forall\ LO : INTEGER;\ X : INTEGER \mid X \geq 0 \wedge LO = 0 \bullet LO\ **\ 2 \leq X \wedge X < (X + 1)\ **\ 2$$

vc2_2_2_thm

$$\vdash \forall\ HI : INTEGER;\ LO,\ LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < HI\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

vc2_2_3_thm

$$\vdash \forall\ LO,\ LO_0 : INTEGER;\ X : INTEGER \mid (X \geq 0 \wedge LO_0 = 0) \wedge LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_3_1_thm

$$\vdash \forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = TRUE \bullet LO\ **\ 2 \leq X \wedge X < (LO + 1)\ **\ 2$$

vc2_3_2_thm

$$\vdash \forall\ HI : INTEGER;\ LO : INTEGER;\ X : INTEGER \mid (LO\ **\ 2 \leq X \wedge X < HI\ **\ 2) \wedge LO + 1\ eq\ HI = FALSE \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

vc2_3_3_thm

$$\vdash \forall\ HI,\ HI_0 : INTEGER;\ LO,\ LO_0 : INTEGER;\ X : INTEGER \mid (LO_0\ **\ 2 \leq X \wedge X < HI_0\ **\ 2) \wedge LO\ **\ 2 \leq X \wedge X < HI\ **\ 2 \bullet LO\ **\ 2 \leq X \wedge X < HI\ **\ 2$$

EVALUATION GUIDELINES

6.1 Introduction

The Compliance Notation gives considerable flexibility in the degree of formal rigour applied at each stage in the presentation of a compliance argument. This chapter gives some guidelines for assessing a compliance argument.

6.2 Scope of a Compliance Argument

A compliance argument will contain one or more specification statements acting as the starting points of the formal argument. These are the specification statements that give the top level statement of critical properties of the code.

For a subprogram library, the formal starting points might comprise the specification statements for each of the subprograms in a suite of package specifications defining the external interface of the library.

For a complete application, the formal starting point might comprise a single specification statement appearing in the body of the main subprogram of the application. The formal starting points will generally be explicitly identified as such in the narrative parts of the script.

6.3 Conformance with the Ada Standard

While the Compliance Tool detects many types of error, it does not detect all the types of error that an Ada implementation is required to detect during the compilation process. A compliance argument is only valid if the Ada code it contains can be compiled without error using an Ada implementation that conforms either to the 1995 version of the standard [12] or to the 1983 version [11]. Providing there are no compilation errors, the checks made by the tool ensure that the formally developed code is not sensitive to the difference between the two versions of the standard.

The predefined type *CHARACTER* was changed in the 1995 standard: the value *CHARACTER'LAST* was increased from 127 to 255. The tool takes 127 as a lower bound on *CHARACTER'LAST*. If the compliance argument includes an axiom that further constrains the value of *CHARACTER'LAST*, then the argument should include informal justification that the axiom is compatible with the intended Ada implementation.

6.4 Formal Development Steps

The formal argument for each starting point will comprise a chain, or tree, of formal steps each of which may produce one or more verification conditions (VCs). Type definitions may also give rise to

VCs. The various sorts of VC that may be generated are described in detail in *Compliance Notation — Language Description* [15]. There are several sorts of formal development step:

- Explicit refinements: in these the refinement symbol ($\sqsubseteq$) is used to assert that a sequence of statements correctly implements a specification statement.
- Implicit refinements: these occur when the semantics of the Compliance Notation require one specification statement to refine another (e.g., this happens when a formal subprogram in a package body is implemented: the specification statement of the formal subprogram in the package body must correctly implement the specification statement for the subprogram given in the package specification).
- Declaration replacements: these occur when a k-slot used as a declaration is expanded using the replacement symbol ($\equiv$).

A complete formal development comprises a sequence of steps of the above three sorts in which all specification statements have been refined to code. Given such a complete formal development, the VC generation algorithm is designed so that truth of the VCs entails "partial" or "algorithmic" correctness of the implementation, i.e., so that truth of the VCs implies that the code meets its specification providing it terminates and does not raise any exceptions. Truth of the VCs does not of itself guarantee either that the code terminates or that the code will not raise exceptions.

Note that the tool can only check the correctness of code that it has actually processed. For example, the VC generation algorithm assumes that the subprograms in an external package meet their specifications in the package specification. A formal development of the package body must be processed to ensure that this is the case.

6.5 Informal Development Steps

Other sorts of development step break the chain of formality. These informal development steps are as follows:

- Arbitrary replacements: these are introduced by the arbitrary replacement symbol ($!\equiv$) and enable a k-slot or refinement step to be expanded to an arbitrary fragment of Ada code in an unchecked fashion.
- Statement replacements: these occur when a k-slot as a statement or specification statement as a statement is expanded using the statement replacement symbol ($!\sqsubseteq$) rather than the refinement symbol ($\sqsubseteq$).

If either of these sorts of step occurs in a script then the VCs do not give a formal guarantee that the code is correct. An informal justification of the correctness of the informal step must be provided and evaluated in this case.

6.6 Consistency

An inconcistent axiom allows any property to be proved. ProofPower supports two modes of dealing with a Z axiomatic description: in the conservative mode, the defining property is replaced by a property which is guaranteed consistent and is equivalent to the desired property if the latter

is provably consistent; in the axiomatic mode, the defining property is asserted as an axiom. If consistency is a concern, then ProofPower should be put into its conservative mode of working by setting the system flag *z_use_axioms* to *false*. This may be done by:

```
set_flag("z_use_axioms", false);
```

This will ensure that all axiomatic descriptions include a consistency caveat. Consistency proofs may then be carried out using the usual ProofPower mechanisms.

6.7 Checking for Errors

When an error is detected during loading of a script, the normal behaviour of the tool is to raise an exception and stop processing. To check that a script has been processed without error, the output from loading the script can be examined. Alternatively, the function *print_exception_log* can be used to print a log of all the errors and potential soundness problems that have been detected in a script. ProofPower-ML exception handlers and certain control flags can be used to make the tool continue processing after an error has been detected. There is also a flag *cn_stop_on_exceptions* defined as part of the Compliance Tool that may be used for this purpose (see section 7 below). The script may be checked for occurrences of the *handle* keyword or the *set_flag* command to see if error reporting has been suppressed inappropriately. See ProofPower *Description Manual* [8] for a description of the ProofPower control flags.

Some errors are only detected when the Ada program is generated (see section 4.2). Thus each script should include a command to generate the Ada program. This command should come after all the Ada code in the script. Errors detected during Ada program generation are logged even if the command is protected by a ProofPower-ML exception handler. In some cases, these errors may refer to informal parts of the Ada program and so may be considered acceptable.

The process of generating and reloading the Z document as described in section 4.3 is not intended to give rise to errors except in unusual circumstances. However, it does provide an extra layer of checking (by passing all the generated Z through the Z parser and type inferrer) and should increase confidence that the script has been processed correctly by the tool. For example, it ensures that all the names of Z objects that have been generated conform to the lexical rules for Z identifiers.

6.8 Treatment of Real Types

Ada floating point and fixed types are mapped by the Compliance Tool to the real numbers of pure mathematics as represented in the Z theory of real numbers. This is an idealisation and means outside the scope of the tool must be used to justify the approximations inherent in the use of computer arithmetic when the code is executed.

For fixed point types, the accuracy of the idealisation is heavily dependent on careful use of the Ada features for controlling the accuracy of the arithmetic. If used carefully, a fixed point type will represent a subset of the mathematical real numbers with complete accuracy (e.g., by using a representation clause, if necessary, to make "small" the precise value needed by the application and by restricting operations to addition, subtraction and multiplication by integers). On the other hand, a fixed point type whose "delta" is specified as $2^n - 1$ for a large integer n and which has no representation clause to define "small" will give a particularly bad approximation.

For floating point types, the accuracy of the idealisation can be heavily dependent on the compiler used if the Ada features for writing portable floating code are not used carefully.

ProofPower-ML COMPLIANCE TOOL REFERENCE

The following sections give reference documentation for the principal ProofPower-ML commands which are specific to the Compliance Tool, followed by the theory listing of "cn". Consult the ProofPower reference manual for further information on ProofPower-ML commands mentioned in this document which are not mentioned here.

7.1 Controlling the Tool

SML

signature **ComplianceTool** = *sig*

Description This is the signature for the commands which a user requires to operate the compliance tool.

SML

val **print_exception_log** : *string* −> *bool*;
val **output_exception_log** : {*script* : *string*, *out_file* : *string*} −> *bool*;
val **delete_exception_log** : *string* −> *unit*;

Description *print_exception_log* causes the exception log for the named script to be printed to standard output. *output_exception_log* prints the exception log to a file. Both of these functions return true if some exceptions have been logged for the script. If no exceptions have been logged a message to that effect is output and the functions return false. Both functions will raise an exception if the named script has not been processed.

delete_exception_log deletes the exception log for the named script.

SML

```
val new_script : {name : string, unit_type : string} -> unit
val new_script1 :
        {name : string, unit_type : string,
                library_theories : string list} -> unit
val new_continuation_script : {name : string, unit_type : string} -> unit
val new_continuation_script1 :
        {name : string, unit_type : string,
                library_theories : string list} -> unit
val delete_script : {name : string, unit_type : string} -> unit;
val delete_deferred_subprogram : string -> unit;
```

Description *new_script* takes as an argument the name of an Ada compilation unit and a string indicating the type of compilation unit, which must be one of "*spec*" (package specification), "*body*" (package body), "*proc*" (procedure) or "*func*" (function). It creates a new ProofPower theory, into which the Z document generated by the subsequent literate script is placed. The name of the theory is derived from the compilation unit name and type. The data structure containing the SPARK program is set to its initial empty value.

new_script1 acts the same as *new_script* except that its list of library theories will be made the parents of the script theory, and any theory produced during processing the script.

new_continuation_script and *new_continuation_script* are just like *new_script* and *new_script1*, except that the data structure containing the SPARK program is retained.

delete_script removes a script from the Compliance Tool state together with any associated theories and other items in the state dependent on the script so that the script can be reloaded. It prints a report of what has been deleted.

delete_deferred_subprogram allows the part of a script associated with a deferred subprogram to be re-entered. It deletes all theories associated with the deferred subprogram and undoes the effects of any refinement or replacement steps associated with the deferred subprogram.

SML

```
val open_scope : string -> unit
```

Description With each declarative region in an Ada program, there is a Z theory to hold definitions and verification conditions that are needed in the scope of the declarative region. The scope is initially set by *new_script* to correspond to the declarative region of the compilation unit. The *open_scope* command is then used to navigate into the scope of nested subprogram bodies. The argument of *open_scope* is the expanded name of the package or subprogram. For example, if the body of a package named *Utils* defines a subprogram named *sort*, then before expanding a k-slot or specification statement in the body of *sort*, the following call is required:

```
open_scope "Utils.sort" ;
```

open_scope may also be used to return to the scope of the compilation unit after opening the scope of a package or subprogram contained in it.

SML

val **print_ada_program** : *string* −> *unit*
val **output_ada_program** : {*script* : *string*, *out_file* : *string*} −> *unit*

Description The call *print_spark_program name* generates and prints the Ada program contained in the script with the specified name. A minus sign may be used for the name to specify the current script. The Ada program is created by filling in the k-slots and specification statements in a literate script with their expansions. The result is printed on the standard output channel in Ada syntax such that it could be subsequently compiled using an Ada compiler.

output_spark_program does the same as *print_spark_program* but allows the user to specify a file to which the program will then be written.

Uses For example, if a literate script called "primes" has just been entered into the Compliance Tool then the following command will output the corresponding Ada to a file called *primes.ada*.

```
output_ada_program{script = "-", out_file="primes.ada"}
```

SML

val **print_z_document** : *string* −> *unit*
val **output_z_document** : {*script* : *string*, *out_file* : *string*} −> *unit*
val **get_script_theories** : *string* −> *string list*

Description *print_z_document* prints the Z document from the named script to the standard output. The listing is in a format suitable for its re-entry into a ProofPower session.

output_z_program is similar to *print_z_document*, but the the Z document is written to the named output file *out_file*.

Uses For example, if a literate script called "primes" has been entered into the Compliance Tool then the following command will output the corresponding Z document to a file called *primes.zdoc*.

```
output_z_document{script="primes", out_file="primes.zdoc"}
```

get_script_theories n returns a list of all the ProofPower theories associated with the Compliance Notation script *n*.

SML

type **EVAL_REPORT**
val **get_eval_report** :
    {*title* : *string*, *theories* : *string list*} −> *EVAL_REPORT*;
val **print_eval_report** : *EVAL_REPORT* −> *unit*;
val **output_eval_report** : {*report* : *EVAL_REPORT*, *out_file* : *string*} −> *unit*;
val **output_eval_report1** : {*report* : *EVAL_REPORT*, *out_file* : *string*} −> *unit*;

Description These functions are used to create a report for assistance in evaluating a compliance argument. The type *EVAL_REPORT* is an abstract type representing the logical contents of the report. To create a report first use *get_eval_report* to give a title to the report specifying the theories to be included in the report. The report can then be output to the screen using *print_eval_report* or to a file, either in *LaTeX* format using *cn_output_eval_report* or in raw ProofPower text format using *cn_output_eval_report1*.

SML

val **browse_vcs** : *unit* −> *unit*

Description On systems with support for X/Windows and Motif enabled, this command invokes an interactive browsing tool for mapping VCs onto web clauses, and vice versa.

Errors

516201 The VC browser is not available on this ProofPower system

SML

val **output_hypertext_edit_script** : {*out_file* : *string*} −> *unit*

Description This function produces an edit script from a processed literate script. The edit script is placed in a file with name *out_file*.

The edit script will introduce hypertext links between specification statements or K-Slots, and their corresponding expansion paragraphs.

SML

(∗ *Flag* **cn_use_let_in_vcs** − *boolean control*, *default false* ∗)

Description This is the name of flag (see *set_flag* in ProofPower *Reference Manual* [16]). The flag controls the way substitution of expressions for variables is treated during VC generation.

If the flag is false (the default), then the substitution is carried out using the HOL *subst* function and then conversions are used to transform the result into Z: this results in a Z term in which the variables have actually been replaced by their substitutes, and declarations and other constructs have been adjusted as necessary to avoid variable capture.

If the flag is set true (using *set_flag*), then a Z *let* construct is used to give the semantics of substitution without actually replacing any variables with their substitutes. This can help to abbreviate the VC and make its structure clearer.

SML

(∗ *Flag* **cn_show_typing_context** − *boolean control*, *default false* ∗)

Description This is the name of a flag (see *set_flag* in ProofPower *Reference Manual* [16]). It controls a diagnostic display which may be of help in understanding problems with type-checking.

When set to true the flag will cause the tool to display the typing context used when type-checking each Z expression encountered in Compliance Notation clauses. The display gives a list of the theories that are in scope (omitting *z_library* and its ancestors) and a list of the variables that are have their type fixed by the context (e.g., variables corresponding to SPARK program variables).

This flag is primarily a diagnostic aid for developers of the tool, but may be of help to a general user in some difficult cases.

SML

(∗ **cn_case_of_ada_keywords** − *string control*, *default* `"lower"` ∗)

Description This is the name of a string control (see *set_string_control* in ProofPower *Reference Manual* [16]). It controls the case of Ada reserved words in the output from *print_ada_program* and *output_ada_program*. There are three allowed values for the string control: `"upper"`, `"lower"` and `"as input"`. The values `"upper"` and `"lower"` makes the tool generate upper-case and lower-case reserved words respectively; the value `"as input"` makes the tool use the same case for a reserved word as was used at its first appearance in the original Compliance Notation script.

SML

```
(* cn_automatic_line_splitting – integer control declared by new_int_control, default 80 *)
(* cn_tab_width – integer control declared by new_int_control, default 2 *)
(* cn_left_margin – integer control declared by new_int_control, default 0 *)
```

Description These controls define parameters for output from *format_web_clause*, *print_web_clause*, *print_spark_program*, *output_spark_program* and *strings_from_fmt1*.

These functions always split lines at logical break points (e.g. at semicolons at the end of statements). However, they will also split lines that are longer than the value set by the control *cn_automatic_line_spliting* (if it is non-zero), chosing to split at a syntactically allowed location. If the control is set to 0 then no automatic line-splitting is done (except in Z terms, which will be split during formatting to match the current line length setting). The default value of the control is 80.

cn_tab_width controls the number of spaces used to indent a nested structure such as the statement in the then part of an if statement. *cn_left_margin* specifies an indentation to be applied on every line and is given in the units specified by *cn_tab_width*. E.g., if *cn_tab_width* is 2 and *cn_left_margin* is 4, every line will be indented by at least 8 spaces, and each nested structure will be indented 2 spaces more than the structure immediately containing it.

SML

```
(* Flag cn_compactify_terms – boolean control, default true *)
```

Description This is the name of a flag (see *set_flag* in ProofPower *Reference Manual* [16]). It controls a feature for optimising the memory space occupied by the Z terms stored in the internal data structures of the Compliance Tool.

Since this space optimisation involves some processing overhead, it is optional, but it is enabled by default. The space optimisation can be disabled by setting the flag *cn_compactify_terms* to *false* and enabled by setting it to *true*.

SML

```
(* Flag cn_syntax_check_only – boolean control, default false *)
```

Description This is the name of a flag (see *set_flag* in ProofPower *Reference Manual* [16]). It allows you to carry out check that a script complies with the syntax of the Compliance Notation.

When the flag is set false, all stages of processing are carried out on each Compliance Notation clause entered.

When the flag is set true, all processing relating to the production of VCs and the Z paragraphs that support them is suppressed. The Ada program may still be produced and all the literate programming language features may be used.

Note that this flag only controls processing of Compliance Notation clauses. Z paragraphs will be processed regardless of the value of this flag.

SML

```
(* Flag cn_stop_on_exceptions – boolean control, default true *)
```

Description This flag controls the handling of exceptions when a Compliance Notation clause is processed. If the flag is true, then if processing a Compliance Notation clause will cause ML exceptions when errors are detected. If the flag is false, the an error message will be printed but an exception will not be raised. In either case, the error message will be recorded in a log that can be inspected using *print_exception_log*, q.v.

The processing of Z paragraphs is not affected by this flag.

SML

```
(* cn_domain_conds: integer control; 3 values allowed as follows *)
val cn_no_domain_conds : int;
val cn_standard_domain_conds : int;
val cn_all_domain_conds : int;
```

Description *cn_domain_conds* is the name of an integer control (see *set_int_control* in ProofPower *Reference Manual* [16]). It controls the generation of domain conditions, i.e., additional hypotheses that are added to VCs to help make them provable in cases where the compliance argument depends for its validity on the program not raising an exception. The control should be given one of the three integer values given by the three ML bindings above. The effect of the control is shown in the following table:

cn_no_domain_conds	No domain conditions are generated.
cn_standard_domain_conds	Only domain conditions corresponding to exceptions that are guaranteed by ALRM [4] are generated.
cn_all_domain_conds	Domain conditions are generated as for *cn_standard_domain_conds* together with domain conditions corresponding to real arithmetic exceptions.

SML

```
(* cn_spark_syntax_warnings – flag declared by new_flag; default false *)
```

Description This flag controls whether or not Ada comments are to be inserted in the output produced by the functions *format_web_clause*, *print_web_clause*, *print_ada_program*, and *output_ada_program* to warn about uses of Ada syntax that is not in the SPARK subset. Note that the checks are only made on the Ada concrete syntax, ignoring comments, there is no checking that required SPARK annotations are present or correct and no context-sensitive checks are made. The checks are based on the syntax given in John Barnes' book *High Integrity Ada The SPARK Approach* ISBN 0-201-17517-7.

SML

```
(* Flag cn_ignore_spark_annotations – boolean control, default false *)
```

Description This flag controls the treatment of Ada comments having the form of SPARK annotations, i.e., comments beginning with "`--#`" (but see also *cn_spark_annotation_char* below). If the flag is false, the default, then such comments are treated as SPARK annotations and may only appear in syntactic positions where the SPARK syntax accepts an annotation. The annotations are remembered for inclusion in the output when the program is printed. If the flag is true, then such comments are treated as ordinary Ada comments and ignored.

SML

```
(* Flag cn_spark_annotation_char – string control, default "#" *)
```

Description By default SPARK annotations are Ada comments beginning with "`--#`" This string control may be used to specify an alternative to "`#`" for the character that distinguishes SPARK annotation from other Ada comments.

SML

val **array_agg_def** : *int -> unit*

Description This function is used to generate the definitions that support multidimensional array aggregates. These definitions are built-in for arrays of up to 20 dimensions, so you do not need to use this function unless you have array aggregates of more than 20 dimensions. The function should be called after first opening either the theory "cn" or the cache theory for your database.

Errors

508059 the argument to array_agg_def must be at least 2

7.2 Custom Proof Facilities

SML

signature **CNToolkitExtensions** = *sig*

Description This is the signature for the toolkit extensions required by the Compliance Tool. It is specified in DRA/CIS/CSE3/TR/94/27/3.0.

SML

val **cn_boolean_cases_thm** : *THM*;
val **cn_boolean_clauses** : *THM*;
val **cn_boolean_clauses1** : *THM*;
val **cn_boolean_clauses2** : *THM*;
val **cn_and_then_or_else_clauses** : *THM*;
val **cn_boolean_pos_thm** : *THM*;
val **cn_boolean_pred_thm** : *THM*;
val **cn_boolean_succ_thm** : *THM*;
val **cn_boolean_thm** : *THM*;
val **cn_boolean_∈_boolean_thm** : *THM*;
val **cn_boolean_val_thm** : *THM*;
val **cn_relational_clauses** : *THM*;
val **cn_relational_clauses1** : *THM*;
val **cn_¬_true_eq_false_thm** : *THM*;
val **cn_intdiv_0_thm** : *THM*;
val **cn_intdiv_thm** : *THM*;
val **cn_rem_thm** : *THM*;
val **cn_intmod_thm** : *THM*;
val $\mathbf{z\_succ^n\_}\mathbb{Z}\mathbf{\_thm}$: *THM*;
val **cn_integer_to_real_thm** : *THM*;

Description These are the ML names for the theorems in the theory "*cn*", which contains extensions to the Z toolkit required to support the Compliance Notation.

SML

```
val mk_cn_intdiv : TERM * TERM -> TERM;
val mk_cn_rem : TERM * TERM -> TERM;
val mk_cn_intmod : TERM * TERM -> TERM;
val mk_cn_star_star : TERM * TERM -> TERM;
val dest_cn_intdiv : TERM -> TERM * TERM;
val dest_cn_rem : TERM -> TERM * TERM;
val dest_cn_intmod : TERM -> TERM * TERM;
val dest_cn_star_star : TERM -> TERM * TERM;
val is_cn_intdiv : TERM -> bool;
val is_cn_rem : TERM -> bool;
val is_cn_intmod : TERM -> bool;
val is_cn_star_star : TERM -> bool;
```

Description These are constructor, destructor and discriminator functions for the operators which support the numeric operations of the Compliance Notation.

Errors

```
509001 ?0 does not have type ℤ
509002 ?0 is not of the form ⌜ℤ i intdiv j⌝
509003 ?0 is not of the form ⌜ℤ i rem j⌝
509004 ?0 is not of the form ⌜ℤ i intmod j⌝
509005 ?0 is not of the form ⌜ℤ i ** j⌝
```

SML

```
val cn_intdiv_conv : CONV;
val cn_rem_conv : CONV;
val cn_intmod_conv : CONV;
val cn_star_star_conv : CONV;
```

Description These conversions perform evaluation of expressions with constant operands formed using the operators which support the numeric operations of the Compliance Notation.

Each conversion expects an expression of the form $i\ op\ j$ where *op* is one of *intdiv*, *rem*, *intmod*, or $**$, and where i and j are signed integer literals (i.e., either numeric literals or of the form $\sim k$, where k is a numeric literal). The resulting theorem has conclusion $i\ op\ j\ =\ r$, where r is the signed literal resulting from evaluating the expression.

Errors

```
509011 ?0 is not of the form ⌜ℤ i intdiv j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509012 ?0 is not of the form ⌜ℤ i rem j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509013 ?0 is not of the form ⌜ℤ i intmod j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509014 ?0 is not of the form ⌜ℤ i ** j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
```

SML

```
val cn_mod_and_conv : CONV;
val cn_mod_or_conv : CONV;
val cn_mod_xor_conv : CONV;
val cn_mod_not_conv : CONV;
```

Description These conversions perform evaluation of expressions with constant operands formed using the operators which support the numeric operations of the Compliance Notation.

Each conversion expects an expression of the form $i\ op\ j$ or $mod\_not\ i$, where op is one of mod_and, mod_or, mod_xor and where i and j are signed integer literals (i.e., either numeric literals or of the form $\sim k$, where k is a numeric literal). The resulting theorem has conclusion $i\ op\ j\ =\ r$, where r is the signed literal resulting from evaluating the expression.

Errors

```
509021 ?0 is not of the form ⌜ℤ i mod_and j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509022 ?0 is not of the form ⌜ℤ i mod_or j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509023 ?0 is not of the form ⌜ℤ i mod_xor j⌝ where ⌜ℤ i⌝ and ⌜ℤ j⌝ are numeric literals
509024 ?0 is not of the form ⌜ℤ mod_not i⌝ where ⌜ℤ i⌝ is a numeric literal
```

SML

```
(* Proof Context: 'cn *)
(* Proof Context: 'cn1 *)
```

Description Component proof context for the theory *cn* which supports the Compliance Notation. *'cn1* is a slightly improved version of the original *'cn*.

The main purpose of these proof contexts is to automate the elimination of the vocabulary of the theory *cn* in favour of plain Z toolkit constructs wherever this is possible without introducing excessive complexity.

Expressions and predicates treated by the proof contexts are constructs formed from:

```
not _, _ and _, _ or _, _ xor _, , _ and_then _, _ or_else _,
_ mem _, _ notmem _, _ eq _, _ noteq _,
_ less _, _ less_eq _, _ greater _, _ greater_eq _,
_ intdiv _, _ rem _, _ ** _, _ mod_and _, _ mod_or _,
_ mod_xor _, _mod_not
```

Contents

Rewriting:

```
cn_boolean_thm (cn only), cn_boolean_succ_thm, cn_boolean_pred_thm,
cn_boolean_pos_thm, cn_boolean_val_thm,
cn_boolean_clauses, cn_relational_clauses,
cn_intdiv_conv, cn_rem_conv, cn_intmod_conv, cn_star_star_conv
cn_boolean_∈_boolean_thm (cn1 only)
z_size_dot_dot_conv (cn1 only)
cn_and_then_or_else_clauses
```

Stripping theorems: (none)

See Also *cn*, *cn_ext*

SML

(∗ *Proof Context*: ′**cn_reals** ∗)

Description Component proof context for the theory *cn* which supports the Compliance Notation treatment of Ada fixed and floating point types.

The purpose of the proof context is to automate the elimination of the vocabulary of the theory *cn* concerned with real numbers in favour of plain Z toolkit constructs wherever this is possible without introducing excessive complexity.

Expressions and predicates treated by the proof contexts are constructs formed from the Z real arithmetic operators and the Compliance Notation operators *_e_*, *integer_to_real* and *integer_to_real*.

This proof context will typically be used in conjunction with one of the other Compliance Notation proof contexts and the proof context for the Z real numbers. E.g.,

set_merge_pcs["′*cn_reals*", "′*z_reals*", "*cn*"];

Contents

Rewriting:

cn_e_0_thm
cn_relational_clauses1
cn_integer_to_real_thm

Stripping theorems:

cn_relational_clauses1

See Also ′*cn*, ′*cn1*

SML

(∗ *Proof Context*: **cn** ∗)
(∗ *Proof Context*: **cn_ext** ∗)
(∗ *Proof Context*: **cn1** ∗)
(∗ *Proof Context*: **cn1_ext** ∗)

Description Complete proofs context for the theory *cn* which supports the Compliance Notation. *cn1* is the recommended proof context for normal use while reasoning about VCs generated by the compliance tool. *cn* is still provided for backwards compatibility.

cn1 is the merge of the component proof contexts ′*cn1* and *z_library1*. *cn1_ext* is the merge of the component proof contexts ′*cn* and *z_library1_ext*.

cn is the merge of the component proof contexts ′*cn* and *z_library*. *cn_ext* is the merge of the component proof contexts ′*cn* and *z_library_ext*.

See Also ′*cn1*, ′*cn*, *z_library1*, *z_library1_ext*

SML

signature **CNTactics** = *sig*

Description This is the name of a metalanguage structure containing tactics etc. supporting proof of the VCs generated by the Compliance Tool.

SML

val **cn_vc_simp_tac** : *THM list -> TACTIC*;

Description This tactic performs simplifications which are often appropriate at the beginning of the proof of a VC goal generated by the Compliance Tool. It should generally be used in the proof context *cn* or *cn1* or in some proof context incorporating one of these. It attempts to carry out the following steps (and fails if none of them succeeds in changing the goal):

1. Rewrite the conclusion of the goal with: the theorems supplied in the parameter (if any); the rewriting rules built into the current proof context; and, *z_plus_assoc_thm* and *z_times_assoc_thm*.
2. Remove any outer universal quantifiers (using *z_∀_tac*), typically leaving an implication whose operands are conjunctions.
3. Remove any redundant conjuncts from the result of step 2. At this stage, the goal will be proved automatically if the antecedents of the implication subsume the succedents.

For example, using the proof context *cn1*, *cn_vc_simp_tac*, will transform the goal:

```
?⊢    ∀     x : INTEGER; y : INTEGER; z : INTEGER
      |     (x + y) + 1 eq z = TRUE ∧ (x ≥ 0 ∧ y ≥ 0) ∧ x ≥ 0
      •     x ≥ 0 ∧ z greater_eq 0 = TRUE
```

into:

```
?⊢          x ∈ INTEGER ∧ y ∈ INTEGER ∧ z ∈ INTEGER
      ∧     x + y + 1 = z ∧ 0 ≤ x ∧ 0 ≤ y
      ⇒     0 ≤ z
```

Errors

518003 Could not simplify the goal ?0

SML

```
val current_pc_z_∈_net :  unit -> (TERM -> THM) NET
val set_pc_z_∈_rules :    (TERM * (TERM -> THM)) list ->
                          string -> unit;
val get_pc_z_∈_rules :    string ->
                          ((TERM * (TERM -> THM)) list * string) list;
val pc_z_∈_rules_of_thms :      string list ->
                          THM list -> (TERM * (TERM -> THM)) list;
```

Description These are tools used to construct proof contexts for use in conjunction with the tactic *cn_hc_tac*.

SML

val **cn_∈_type_tac** : *THM list -> TACTIC*

Description This is a tactic for proving assertions of the form $e \in T$ where T is the representation in Z of a SPARK type and e is the translation into Z of a SPARK expression.

The heuristic approach used by the tactic works best with the proof contexts that are generated by *cn_make_script_support*. It will also work with the proof context *cn1* if only predefined SPARK types are involved.

The tactic uses the assumptions of the goal, together with any theorems supplied as an argument as ground facts in an attempt to reduce assertions of the form $e \in T$ according to the structure of e. For example, an assertion of the form $f(x) \in T$ is reduced to the two assertions $f \in X \rightarrow T$ and $x \in X$ and, if the ground facts provide a suitable candidate for the variable X, then the original assertion is proved.

For example, given the SPARK declarations:

```
type ABC is (A, B, C);
type BYTE is range 0 .. 255;
type MARKED_BYTE is record B : BYTE; F : BOOLEAN; end record;
type ARR is array (ABC) of MARKED_BYTE;
```

cn_∈_type_tac will prove goals such as:

```
W ∈ ARR ; I ∈ ABC ?⊢ (W I).B ∈ BYTE
```

SML

signature **CNTheoryProofSupport** = *sig*

Description This is the signature for the Theory Proof Support Tools.

SML

val **z_norm_sig_h_schema_conv** : $CONV$;

Description Given an arbitrary horizontal schema, $\lceil_{\mathrm{Z}}[sig|body]\rceil$, this conversion will return the theorem that states that the original horizontal schema is equal to one with the signature variables placed in *z_sig_order* in a list of declarations (SDECLs) each with only a single variable. Items in the signature that are not simple declarations will be placed as a list after the sorted simple declarations, in their original order of occurrence. E.g.

$[x,\ z{:}\ X;\ y : Y;\ T;\ v,\ w :Z;\ S \mid (v,\ w,\ x,\ z,\ y,\ w) \in p\ ]$
will become
$[\ v : Z;\ w : Z;\ x : X;\ y : Y;\ z : X;\ T;\ S \mid (v,\ w,\ x,\ z,\ y,\ w) \in p\ ]$

Conversion

$$\frac{}{\vdash [\ sig \mid pred\ ] = [\ sig' \mid pred\ ]} \qquad \begin{array}{l} z\_norm\_sig\_h\_schema\_conv \\ \lceil_{\mathrm{Z}}[\ sig \mid pred\ ]\rceil \end{array}$$

and

Conversion

$$\frac{}{\vdash [\ sig\ ] = [\ sig'\ ]} \qquad \begin{array}{l} z\_norm\_sig\_h\_schema\_conv \\ \lceil_{\mathrm{Z}}[\ sig\ ]\rceil \end{array}$$

where sig' is the normalised form of sig. It is possible for this conversion to cause no change.

Errors

47940 ?0 is not a Z schema

SML

val **cn_simplify_canon** : *THM -> THM list*;

Description This does some Compliance Notation-specific simplifications, if the appropriate definitions are available. In the following examples the likely constant name suffixes are used, but are not required by the canonicalisation function.

$\vdash$ *namevSUCC* = (*name* $\setminus$ {*namevLAST*}) $\lhd$ *succ*
becomes $\vdash \forall\ i : name_1 \mathinner{..} name_2 + \sim 1 \bullet namevSUCC\ i = i + 1$
and $\vdash namevSUCC \in name_1 \mathinner{..} (name_2 + \sim 1) \rightarrow name_1 + 1 \mathinner{..} name_2$
where $name_1$ *is the bottom, and* $name_2$ *is the top expression of the range that is translated from name*

$\vdash$ *namevPRED* = *namevSUCC* $\sim$
becomes $\vdash \forall\ i : name_1 + 1 \mathinner{..} name_2 \bullet namevPRED\ i = i + \sim 1$
and $\vdash namevPRED \in name_1 + 1 \mathinner{..} name_2 \rightarrow name_1 \mathinner{..} (name_2 + \sim 1)$
where $name_1$ *is the bottom, and* $name_2$ *is the top expression of the range that is translated from name*

$\vdash$ *namevPOS* = *id name*
becomes $\vdash \forall\ i : name \bullet namevPOS\ i = i$
and $\vdash namevPOS \in name \rightarrow name$

$\vdash$ *namevVAL* = *namevPOS* $\sim$
becomes $\vdash \forall\ i : name \bullet namevVAL\ i = i$
and $\vdash namevVAL \in name \rightarrow name$

$\vdash$ *name* = $\{u : master \mid dom\ u = index\}$
becomes $\vdash name = (index \rightarrow comp) \cap master$

$\vdash$ *name* = [*sig*]
becomes $\vdash$ *name* = [*sig*$'$]
where sig$'$ *is the normalised form of sig, via z_norm_sig_z_schema_conv*

$\vdash$ *name* = [*sig* | *pred*]
becomes $\vdash$ *name* = [*sig*$'$ | *pred*]
where sig$'$ *is the normalised form of sig, via z_norm_sig_z_schema_conv*

The canonicalisation may also do some limited simplification to theorems that partially match the patterns given above. If none of simplifications apply then the canon returns the empty list.

Errors

517006 Theory cn is not an ancestor of the current theory

SML

val **cn_script_support_thms** : *string -> THM list*;
val **list_cn_script_support_thms** : *string list -> THM list*;

Description *cn_script_support_thms* takes a theory name as argument. Assuming the theory is in scope it will examine each of the definitions and axioms in the theory that declare generic variables, and perhaps save some theorems for each in the current theory, as below.

As a default Z paragraphs saved as axioms or definitions initially will be processed as if retrieved by *z_get_spec*, and rewritten according to the rules of proof context *z_library1*. Some of the paragraph forms can define more than one value, and the function will save theorems for each of the values defined. The description below assumes that a single item is defined, called "name", with th obvious extensions for multiple declarations. Non-alphanumeric names may be normalised to fit in with ML variable name rules.

Abbreviation Definitions If *cn_simplify_canon* applies, then its results will be saved. The first theorem, if any, returned by *cn_simplify_canon* will be saved with name *cn_name_thm*; the second theorem returned will be saved with name *cn_name_sig_thm*; and if it returns no theorems then the generic form of the theorem will be saved with name *cn_name_thm*.

Free Type Definitions The signature part will be saved with name *cn_name_sig_thm* (for each constructor); the body, if non-trivial, with name *cn_name_thm* (for each constructor). A theorem for free type name, of the form $name = U$, will be saved with name *cn_name_sig_thm*.

Axiomatic Box The signature part will be saved with name *cn_name_sig_thm*, the body, if non-trivial, with name *cn_name_thm*. No theorem will be saved if there is no signature.

Given Sets A theorem stating that the given set is equal to U will be saved with name *cn_name_thm*.

Constraint No theorem will be generated, as constraints do not define a generic variable (though they may well constrain an existing variable).

Saved theorems will also be bound to an ML variable of the same name. If a theorem is already saved with the chosen theorem name then it is assumed to be for an earlier use of the support tool, and will be replaced, and a warning issued. Finally, a list of the theorems generated will be returned by the function.

list_cn_script_support_thms acts like *cn_script_support_thms* mapped over a list of theory names, except that duplicate names in the list will be ignored.

See Also *cn_make_script_support*, *list_cn_spec_rule*, *cn_spec_rule*,

Errors

517001 Theorem ?0 in theory ?1 has been replaced by a new Compliance Notation support theorem
517002 Cannot force the saving of theorem ?0 in theory ?1
517006 Theory cn is not an ancestor of the current theory
517007 LHS of Abbreviation Definition, ?0, not a generic variable

SML

val **list_cn_spec_rule** : *THM list -> THM list*;

Description *list_cn_spec_rule* takes a list of theorems and analyses those that are Z paragraphs as if by *cn_script_support_thms*, saving and returning the list of created theorems.

Note that in particular this means that the function can be applied to the results of *get_spec*, but not those of *z_get_spec*.

Errors

517001 Theorem ?0 in theory ?1 has been replaced by a new Compliance Notation support theorem
517002 Cannot force the saving of theorem ?0 in theory ?1
517006 Theory cn is not an ancestor of the current theory

SML

val **cn_spec_rule** : *THM -> (string * THM) list*;

Description *cn_spec_rule* analyses its single theorem as if by *cn_script_support_thms*, except that it does not save the resulting theorem, or bind it to an ML variable. The function instead returns the name or names that *cn_script_support_thms* would have used, and the theorem or theorems it would have saved, except that unchanged theorems will result in an empty list being returned.

Errors

517004 Theorem ?0 cannot be processed by Compliance Notation simplification tools
517006 Theory cn is not an ancestor of the current theory

SML

val **cn_make_script_support** : *string -> string -> THM list*;
val **list_cn_make_script_support** : *string list -> string -> THM list*;

Description *cn_make_script_support thyname pcname* will first try to act as *cn_script_support_thms thyname*. It will then create a complete proof context called *pcname* (any previous version will be overwritten). As with *cn_make_script_support_thms*, it returns the list of theorems generated.

list_cn_make_script_support thynames pcname will first try to act as *list_cn_script_support_thms thynames*. It will then create a complete proof context called *pcname* (any previous version will be overwritten).

The proof context created will be "cn1" extended by:

Contents Rewriting: all saved support theorems (except equations with U);

Stripping theorems: bi-implications from support theorems, plus support theorem equations of sets promoted to bi-implications of memberships (except where U elimination will apply).

Stripping conclusions: as stripping conclusions.

Rewriting canonicalisation: no change;

Automatic proof procedures: appropriate support theorems will be noted as to be used in type inference tools, otherwise no change.

Existence prover: no change.

The U simplification material will be extended by appending to the equation context the one formed from the equations of the form $nm = U$, via *set_u_simp_eqn_cxt*.

Usage Notes The new proof context requires theory "cn", its creation theory and theory *thyname* to be in scope. It is not intended to be mixed with HOL proof contexts.

The new proof context is built using proof context "cn1".

Errors

517001 Theorem ?0 in theory ?1 has been replaced by a new Compliance Notation support theorem
517002 Cannot force the saving of theorem ?0 in theory ?1
517006 Theory cn is not an ancestor of the current theory

SML

val **all_cn_make_script_support** : *string -> THM list*;

Description *all_cn_make_script_support pcname* will determine all the theories that are parents of the current one, but that are not also ancestors of "cn" (i.e. this does not include "cn"), and will execute *list_cn_make_script_support* on this list.

That is, this will create supporting theorems and a proof context based on all the in scope Compliance Notation packages.

Usage Notes The new proof context requires theory "cn", its creation theory and theory *thyname* to be in scope. It is not intended to be mixed with HOL proof contexts.

The new proof context is built using proof context "cn1".

Errors

517006 Theory cn is not an ancestor of the current theory

7.3 THE Z THEORY cn

For brevity, the global variables *array_agg6*, *array_agg7*, ..., *array_agg20* have been suppressed from the theory listing.

7.3.1 Parents

z_reals z_library

7.3.2 Global Variables

FALSE $\mathbb{Z}$
TRUE $\mathbb{Z}$
BOOLEAN $\mathbb{P}\ \mathbb{Z}$
BOOLEANvFIRST $\mathbb{Z}$
BOOLEANvLAST $\mathbb{Z}$
BOOLEANvSUCC $\mathbb{Z} \leftrightarrow \mathbb{Z}$
BOOLEANvPRED $\mathbb{Z} \leftrightarrow \mathbb{Z}$
BOOLEANvPOS $\mathbb{Z} \leftrightarrow \mathbb{Z}$
BOOLEANvVAL $\mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **and** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **or** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **xor** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(**not** _) $\mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **and_then** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **or_else** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **eq** _)[**X**] $X \times X \leftrightarrow \mathbb{Z}$
(_ **noteq** _)[**X**] $X \times X \leftrightarrow \mathbb{Z}$
(_ **mem** _)[**X**] $X \times \mathbb{P}\ X \leftrightarrow \mathbb{Z}$
(_ **notmem** _)[**X**] $X \times \mathbb{P}\ X \leftrightarrow \mathbb{Z}$
(_ **array_and** _)[**X**] $(X \leftrightarrow \mathbb{Z}) \times (X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(_ **array_or** _)[**X**] $(X \leftrightarrow \mathbb{Z}) \times (X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(_ **array_xor** _)[**X**] $(X \leftrightarrow \mathbb{Z}) \times (X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(**array_not** _)[**X**] $(X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(_ **less** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **less_eq** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **greater** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **greater_eq** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$

(_ **real_less** _)
$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{Z}$
(_ **real_less_eq** _)
$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{Z}$
(_ **real_greater** _)
$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{Z}$
(_ **real_greater_eq** _)
$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{Z}$
(_ **array_less** _)
$(\mathbb{Z} \leftrightarrow \mathbb{Z}) \times (\mathbb{Z} \leftrightarrow \mathbb{Z}) \leftrightarrow \mathbb{Z}$
(_ **array_less_eq** _)
$(\mathbb{Z} \leftrightarrow \mathbb{Z}) \times (\mathbb{Z} \leftrightarrow \mathbb{Z}) \leftrightarrow \mathbb{Z}$
(_ **array_greater** _)
$(\mathbb{Z} \leftrightarrow \mathbb{Z}) \times (\mathbb{Z} \leftrightarrow \mathbb{Z}) \leftrightarrow \mathbb{Z}$
(_ **array_greater_eq** _)
$(\mathbb{Z} \leftrightarrow \mathbb{Z}) \times (\mathbb{Z} \leftrightarrow \mathbb{Z}) \leftrightarrow \mathbb{Z}$
(_ **intdiv** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **rem** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **intmod** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ ** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
integer_to_real
$\mathbb{Z} \leftrightarrow \mathbb{R}$
real_to_integer
$\mathbb{R} \leftrightarrow \mathbb{Z}$
(_ **mod_and** _)
$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **mod_or** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(_ **mod_xor** _)
$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
(**mod_not** _) $\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
INTEGER $\mathbb{P}\ \mathbb{Z}$
INTEGERvSUCC $\mathbb{Z} \leftrightarrow \mathbb{Z}$
INTEGERvPRED $\mathbb{Z} \leftrightarrow \mathbb{Z}$
INTEGERvPOS $\mathbb{Z} \leftrightarrow \mathbb{Z}$
INTEGERvVAL $\mathbb{Z} \leftrightarrow \mathbb{Z}$
INTEGERvFIRST
$\mathbb{Z}$
INTEGERvLAST $\mathbb{Z}$
NATURAL $\mathbb{P}\ \mathbb{Z}$
NATURALvFIRST
$\mathbb{Z}$
NATURALvLAST $\mathbb{Z}$
NATURALvSUCC $\mathbb{Z} \leftrightarrow \mathbb{Z}$
NATURALvPRED $\mathbb{Z} \leftrightarrow \mathbb{Z}$
NATURALvPOS $\mathbb{Z} \leftrightarrow \mathbb{Z}$
NATURALvVAL $\mathbb{Z} \leftrightarrow \mathbb{Z}$
POSITIVE $\mathbb{P}\ \mathbb{Z}$
POSITIVEvFIRST
$\mathbb{Z}$
POSITIVEvLAST
$\mathbb{Z}$

POSITIVEvSUCC
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
POSITIVEvPRED
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
POSITIVEvPOS $\mathbb{Z} \leftrightarrow \mathbb{Z}$
POSITIVEvVAL $\mathbb{Z} \leftrightarrow \mathbb{Z}$
LONG_INTEGER $\mathbb{P}\ \mathbb{Z}$
LONG_INTEGERvSUCC
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
LONG_INTEGERvPRED
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
LONG_INTEGERvPOS
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
LONG_INTEGERvVAL
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
LONG_INTEGERvFIRST
$\mathbb{Z}$
LONG_INTEGERvLAST
$\mathbb{Z}$
SHORT_INTEGER
$\mathbb{P}\ \mathbb{Z}$
SHORT_INTEGERvSUCC
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
SHORT_INTEGERvPRED
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
SHORT_INTEGERvPOS
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
SHORT_INTEGERvVAL
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
SHORT_INTEGERvFIRST
$\mathbb{Z}$
SHORT_INTEGERvLAST
$\mathbb{Z}$
universal_discrete
$\mathbb{P}\ \mathbb{Z}$
universal_discretevSUCC
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
universal_discretevPRED
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
universal_discretevPOS
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
universal_discretevVAL
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
universal_discretevFIRST
$\mathbb{Z}$
universal_discretevLAST
$\mathbb{Z}$
FLOAT $\mathbb{P}\ \mathbb{R}$
FLOATvDIGITS $\mathbb{Z}$
FLOATvFIRST$\mathbb{R}$
FLOATvLAST $\mathbb{R}$

SHORT_FLOAT $\mathbb{P}\ \mathbb{R}$
SHORT_FLOATvDIGITS
$\mathbb{Z}$
SHORT_FLOATvFIRST
$\mathbb{R}$
SHORT_FLOATvLAST
$\mathbb{R}$
LONG_FLOAT $\mathbb{P}\ \mathbb{R}$
LONG_FLOATvDIGITS
$\mathbb{Z}$
LONG_FLOATvFIRST
$\mathbb{R}$
LONG_FLOATvLAST
$\mathbb{R}$
CHARACTERvFIRST
$\mathbb{Z}$
CHARACTERvLAST
$\mathbb{Z}$
CHARACTER $\mathbb{P}\ \mathbb{Z}$
CHARACTERvSUCC
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
CHARACTERvPRED
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
CHARACTERvPOS
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
CHARACTERvVAL
$\mathbb{Z} \leftrightarrow \mathbb{Z}$
STRING $\mathbb{P}\ (\mathbb{Z} \leftrightarrow \mathbb{Z})$
Z_CHAR $\mathbb{P}\ (\mathbb{Z} \leftrightarrow \mathbb{S})$
Z_STRING $\mathbb{P}\ (\mathbb{Z} \leftrightarrow \mathbb{S})$
dest_char $\mathbb{S} \leftrightarrow \mathbb{Z}$
(**string_lit** _)
$(\mathbb{Z} \leftrightarrow \mathbb{S}) \leftrightarrow \mathbb{Z} \leftrightarrow \mathbb{Z}$
(**char_lit** _) $(\mathbb{Z} \leftrightarrow \mathbb{S}) \leftrightarrow \mathbb{Z}$
Informal_Function
$\mathbb{P}\ \mathit{Informal\_Function}$
(_ $\&_{\mathbf{2}}$ _)[**X**] $X \times (\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
(_ $\&_{\mathbf{1}}$ _)[**X**] $(\mathbb{Z} \leftrightarrow X) \times X \leftrightarrow \mathbb{Z} \leftrightarrow X$
(_ $\&_{\mathbf{0}}$ _)[**X**] $(\mathbb{Z} \leftrightarrow X) \times (\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
slide[**X**, **Y**] $(X \leftrightarrow Y) \times \mathbb{P}\ X \leftrightarrow X \leftrightarrow Y$
Boolean $\mathbb{B} \leftrightarrow \mathbb{Z}$
VRElsfInd $\mathbb{P}\ \mathit{VRElsfInd}$
VRElsfTrue $\mathit{VRElsfInd}$
VRElsfFalse $\mathit{VRElsfInd}$
VC_Route $\mathbb{P}\ \mathit{VC\_Route}$
(**VRAny** _) $\mathit{VC\_Route} \leftrightarrow \mathit{VC\_Route}$
(**VRNull** _) $\mathit{VC\_Route} \leftrightarrow \mathit{VC\_Route}$
(**VRAssign** _) $\mathit{VC\_Route} \leftrightarrow \mathit{VC\_Route}$
(**VRSpecVia** _)
$\mathit{VC\_Route} \leftrightarrow \mathit{VC\_Route}$
(**VRSpecToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

VRSpecPreIntro

VC_Route

(**VRSemi** _) $VC\_Route \leftrightarrow VC\_Route$

(**VREndSemi** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRIfThen** _) $VRElsfInd \times VC\_Route \leftrightarrow VC\_Route$

(**VRIfElse** _) $VRElsfInd \times VC\_Route \leftrightarrow VC\_Route$

(**VREndIf** _) $VC\_Route \leftrightarrow VC\_Route$

(**VRCaseBranch** _)

$\mathbb{Z} \times VC\_Route \leftrightarrow VC\_Route$

(**VRCaseOthers** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VREndCase** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRLoopUndecVia** _)

$VC\_Route \leftrightarrow VC\_Route$

VRLoopUndecPreIntro

VC_Route

VRLoopUndecPreToSide

VC_Route

(**VRLoopUndecToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRWhileVia** _)

$VC\_Route \leftrightarrow VC\_Route$

VRWhilePreIntro

VC_Route

(**VRWhileWPToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRWhileToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRForVia** _) $VC\_Route \leftrightarrow VC\_Route$

VRForPreIntro

VC_Route

VRForPreToSide

VC_Route

(**VRForToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRForExitToSide** _)

$VC\_Route \leftrightarrow VC\_Route$

VRExitTillIntro

$(\mathbb{Z} \leftrightarrow \mathbb{S}) \leftrightarrow VC\_Route$

(**VRExitVia** _)

$VC\_Route \leftrightarrow VC\_Route$

VRReturnIntro

VC_Route

(**VRProcCall** _)

$VC\_Route \leftrightarrow VC\_Route$

(**VRProcCallEnd** _)

$VC\_Route \leftrightarrow VC\_Route$

VRProcCallRngIntro

VC_Route
(**VRLogConToSide** _)
$VC\_Route \leftrightarrow VC\_Route$
VRLogConPreIntro
VC_Route
VRLogConTypeMemIntro
VC_Route
(**VRRefinementBegin** _)
$VC\_Route \leftrightarrow VC\_Route$
VRRefinementIntro
VC_Route
array_agg2[**g1**, **g2**, **g**]
$(g1 \leftrightarrow g2 \leftrightarrow g) \leftrightarrow g1 \times g2 \leftrightarrow g$
array_agg3[**g1**, **g2**, **g3**, **g**]
$(g1 \leftrightarrow g2 \leftrightarrow g3 \leftrightarrow g) \leftrightarrow g1 \times g2 \times g3 \leftrightarrow g$
array_agg4[**g1**, **g2**, **g3**, **g4**, **g**]
$(g1 \leftrightarrow g2 \leftrightarrow g3 \leftrightarrow g4 \leftrightarrow g) \leftrightarrow g1 \times g2 \times g3 \times g4 \leftrightarrow g$
array_agg5[**g1**, **g2**, **g3**, **g4**, **g5**, **g**]
$(g1 \leftrightarrow g2 \leftrightarrow g3 \leftrightarrow g4 \leftrightarrow g5 \leftrightarrow g)$
$\leftrightarrow g1 \times g2 \times g3 \times g4 \times g5 \leftrightarrow g$

7.3.3 Fixity

fun 0 rightassoc

(*char_lit* _) (*VRIfElse* _) (*VRWhileVia* _)
(*string_lit* _) (*VRIfThen* _) (*VRWhileWPToSide* _)
(*VRAny* _) (*VRLogConToSide* _) (_ *and* _)
(*VRAssign* _) (*VRLoopUndecToSide* _) (_ *and_then* _)
(*VRCaseBranch* _) (*VRLoopUndecVia* _) (_ *array_and* _)
(*VRCaseOthers* _) (*VRNull* _) (_ *array_or* _)
(*VREndCase* _) (*VRProcCallEnd* _) (_ *array_xor* _)
(*VREndIf* _) (*VRProcCall* _) (_ *mod_and* _)
(*VREndSemi* _) (*VRRefinementBegin* _) (_ *mod_or* _)
(*VRExitVia* _)(*VRSemi* _) (_ *mod_xor* _)
(*VRForExitToSide* _) (*VRSpecToSide* _) (_ *or* _)
(*VRForToSide* _) (*VRSpecVia* _) (_ *or_else* _)
(*VRForVia* _) (*VRWhileToSide* _) (_ *xor* _)

fun 10 rightassoc

(_ *eq* _) (_ *mem* _) (_ *noteq* _) (_ *notmem* _)

fun 20 rightassoc

(_ *array_greater* _) (_ *less* _)
(_ *array_greater_eq* _)(_ *less_eq* _)
(_ *array_less* _) (_ *real_greater* _)
(_ *array_less_eq* _) (_ *real_greater_eq* _)
(_ *greater* _) (_ *real_less* _)
(_ *greater_eq* _) (_ *real_less_eq* _)

fun 30 rightassoc
$(\_\ \&_0\ \_)$ $(\_\ \&_1\ \_)$ $(\_\ \&_2\ \_)$

fun 40 rightassoc
$(\_\ intdiv\ \_)$ $(\_\ intmod\ \_)$ $(\_\ rem\ \_)$

fun 50 rightassoc
$(array\_not\ \_)$ $(mod\_not\ \_)$ $(not\ \_)(\_\ **\ \_)$

7.3.4 Axioms

_ and _
_ or _
_ xor _
not _ $\vdash ((not\ \_) \in BOOLEAN \to BOOLEAN$
$\wedge \{(\_\ and\ \_), (\_\ or\ \_), (\_\ xor\ \_)\}$
$\subseteq BOOLEAN \times BOOLEAN \to BOOLEAN)$
$\wedge (\forall\ b : BOOLEAN$
$\bullet\ not\ FALSE = TRUE$
$\wedge\ not\ TRUE = FALSE$
$\wedge\ (b\ and\ FALSE = FALSE$
$\wedge\ b\ and\ TRUE = b)$
$\wedge\ (b\ or\ FALSE = b$
$\wedge\ b\ or\ TRUE = TRUE)$
$\wedge\ b\ xor\ FALSE = b$
$\wedge\ b\ xor\ TRUE = not\ b)$

_ and_then _
_ or_else _ $\vdash \{(\_\ and\_then\ \_), (\_\ or\_else\ \_)\}$
$\subseteq BOOLEAN \times BOOLEAN \to BOOLEAN$
$\wedge\ (\_\ and\_then\ \_) = (\_\ and\ \_)$
$\wedge\ (\_\ or\_else\ \_) = (\_\ or\ \_)$

_ eq _
_ noteq _
_ mem _
_ notmem _ $\vdash [X]((\{(\_\ mem\ \_)[X], (\_\ notmem\ \_)[X]\}$
$\subseteq X \times \mathbb{P}\ X \to BOOLEAN$
$\wedge\ \{(\_\ eq\ \_)[X], (\_\ noteq\ \_)[X]\} \subseteq X \times X \to BOOLEAN)$
$\wedge\ (\forall\ x, y : X; S : \mathbb{P}\ X; b : BOOLEAN$
$\bullet\ (b = (\_\ mem\ \_)[X]\ (x, S) \Leftrightarrow b = TRUE \Leftrightarrow x \in S)$
$\wedge\ (b = (\_\ notmem\ \_)[X]\ (x, S)$
$\Leftrightarrow b = TRUE$
$\Leftrightarrow x \notin S)$
$\wedge\ (b = (\_\ eq\ \_)[X]\ (x, y) \Leftrightarrow b = TRUE \Leftrightarrow x = y)$
$\wedge\ (b = (\_\ noteq\ \_)[X]\ (x, y)$
$\Leftrightarrow b = TRUE$
$\Leftrightarrow x \neq y)))$

_ array_and _
_ array_or _
_ array_xor _
array_not _ $\vdash [X](((array\_not\ \_)[X] \in (X \nrightarrow BOOLEAN) \to X \nrightarrow BOOLEAN$

$$
\begin{array}{l}
\qquad\qquad \wedge\ \{(\_\ array\_and\ \_)[X],\\
\qquad\qquad\quad (\_\ array\_or\ \_)[X],\\
\qquad\qquad\quad (\_\ array\_xor\ \_)[X]\}\\
\qquad\qquad\quad \subseteq (X \pfun BOOLEAN) \times (X \pfun BOOLEAN) \fun X \pfun BOOLEAN)\\
\qquad\qquad \wedge\ (\forall\ a,\ b : X \pfun BOOLEAN\\
\qquad\qquad\quad \bullet\ (array\_not\ \_)[X]\ a = (\lambda\ i : dom\ a \bullet not\ a\ i)\\
\qquad\qquad\quad\ \wedge\ (\_\ array\_and\ \_)[X]\ (a,\ b)\\
\qquad\qquad\quad\qquad = (\lambda\ i : dom\ a \cap dom\ b \bullet a\ i\ and\ b\ i)\\
\qquad\qquad\quad\ \wedge\ (\_\ array\_or\ \_)[X]\ (a,\ b)\\
\qquad\qquad\quad\qquad = (\lambda\ i : dom\ a \cap dom\ b \bullet a\ i\ or\ b\ i)\\
\qquad\qquad\quad\ \wedge\ (\_\ array\_xor\ \_)[X]\ (a,\ b)\\
\qquad\qquad\quad\qquad = (\lambda\ i : dom\ a \cap dom\ b \bullet a\ i\ xor\ b\ i)))
\end{array}
$$

_ less _
_ less_eq _
_ greater _
_ greater_eq _

$$
\begin{array}{l}
\qquad\qquad\quad \vdash \{(\_\ less\ \_),\\
\qquad\qquad\qquad\quad (\_\ less\_eq\ \_),\\
\qquad\qquad\qquad\quad (\_\ greater\ \_),\\
\qquad\qquad\qquad\quad (\_\ greater\_eq\ \_)\}\\
\qquad\qquad\qquad\quad \subseteq \mathbb{Z} \times \mathbb{Z} \fun BOOLEAN\\
\qquad\qquad\qquad \wedge\ (\forall\ x,\ y : \mathbb{Z};\ b : BOOLEAN\\
\qquad\qquad\qquad\quad \bullet\ (b = x\ less\ y \Leftrightarrow b = TRUE \Leftrightarrow x < y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ less\_eq\ y \Leftrightarrow b = TRUE \Leftrightarrow x \leq y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ greater\ y \Leftrightarrow b = TRUE \Leftrightarrow x > y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ greater\_eq\ y \Leftrightarrow b = TRUE \Leftrightarrow x \geq y))
\end{array}
$$

_ real_less _
_ real_less_eq _
_ real_greater _
_ real_greater_eq _

$$
\begin{array}{l}
\qquad\qquad\quad \vdash \{(\_\ real\_less\ \_),\\
\qquad\qquad\qquad\quad (\_\ real\_less\_eq\ \_),\\
\qquad\qquad\qquad\quad (\_\ real\_greater\ \_),\\
\qquad\qquad\qquad\quad (\_\ real\_greater\_eq\ \_)\}\\
\qquad\qquad\qquad\quad \subseteq \mathbb{R} \times \mathbb{R} \fun BOOLEAN\\
\qquad\qquad\qquad \wedge\ (\forall\ x,\ y : \mathbb{R};\ b : BOOLEAN\\
\qquad\qquad\qquad\quad \bullet\ (b = x\ real\_less\ y \Leftrightarrow b = TRUE \Leftrightarrow x <_R y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ real\_less\_eq\ y \Leftrightarrow b = TRUE \Leftrightarrow x \leq_R y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ real\_greater\ y \Leftrightarrow b = TRUE \Leftrightarrow x >_R y)\\
\qquad\qquad\qquad\quad\ \wedge\ (b = x\ real\_greater\_eq\ y\\
\qquad\qquad\qquad\qquad \Leftrightarrow b = TRUE\\
\qquad\qquad\qquad\qquad \Leftrightarrow x \geq_R y))
\end{array}
$$

_ array_less _
_ array_less_eq _
_ array_greater _
_ array_greater_eq _

$$
\begin{array}{l}
\qquad\qquad\quad \vdash \{(\_\ array\_less\ \_),\\
\qquad\qquad\qquad\quad (\_\ array\_less\_eq\ \_),\\
\qquad\qquad\qquad\quad (\_\ array\_greater\ \_),\\
\qquad\qquad\qquad\quad (\_\ array\_greater\_eq\ \_)\}\\
\qquad\qquad\qquad\quad \subseteq (\mathbb{Z} \pfun \mathbb{Z}) \times (\mathbb{Z} \pfun \mathbb{Z}) \fun BOOLEAN
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ (\forall\ a,\ b : \mathbb{Z} \pfun \mathbb{Z} \\
\qquad\qquad\qquad\qquad\quad \bullet\ (a\ array\_less\ b = TRUE \\
\qquad\qquad\qquad\qquad\qquad \Leftrightarrow\ (\exists\ i,\ j,\ k : \mathbb{Z} \\
\qquad\qquad\qquad\qquad\qquad\quad \bullet\ \{i,\ j\} \subseteq dom\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ i - 1 \notin dom\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ i + k - 1 \notin dom\ a \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ (\forall\ t : i\ ..\ j - 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \bullet\ t + k \in dom\ a \wedge b\ t = a\ (t + k)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ j + k \in dom\ a \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow\ a\ (j + k) < b\ j)) \\
\qquad\qquad\qquad\qquad\quad \wedge\ a\ array\_less\_eq\ b = a\ array\_less\ b\ or\ a\ eq\ b \\
\qquad\qquad\qquad\qquad\quad \wedge\ a\ array\_greater\ b = b\ array\_less\ a \\
\qquad\qquad\qquad\qquad\quad \wedge\ a\ array\_greater\_eq\ b = b\ array\_less\_eq\ a)
\end{array}
$$

_ intdiv _
_ rem _

_ intmod _ $\vdash \{(\_\ intdiv\ \_),\ (\_\ rem\ \_),\ (\_\ intmod\ \_)\}$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \subseteq \mathbb{Z} \times \mathbb{Z} \setminus \{0\} \to \mathbb{Z} \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ x,\ y : \mathbb{Z} \\
\qquad\qquad\qquad\qquad\quad \mid y \neq 0 \\
\qquad\qquad\qquad\qquad\quad \bullet\ (x * y \geq 0 \Rightarrow x\ intdiv\ y = abs\ x\ div\ abs\ y) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (x * y < 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow x\ intdiv\ y = \sim (abs\ x\ div\ abs\ y)) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ x\ rem\ y = x - (x\ intdiv\ y) * y \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (x * y \geq 0 \vee x\ rem\ y = 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow x\ intmod\ y = x\ rem\ y) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (x * y < 0 \wedge x\ rem\ y \neq 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow x\ intmod\ y = x\ rem\ y + y))
\end{array}
$$

_ ** _ $\vdash (\_\ **\ \_) \in \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ (\forall\ x : \mathbb{Z};\ y : \mathbb{N} \\
\qquad\qquad\qquad\qquad\quad \bullet\ x ** 0 = 1 \wedge x ** (y + 1) = x * x ** y)
\end{array}
$$

integer_to_real

$\vdash integer\_to\_real \in \mathbb{Z} \to \mathbb{R}$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ (\forall\ i : \mathbb{Z} \bullet integer\_to\_real\ i = real\ i)
\end{array}
$$

real_to_integer

$\vdash real\_to\_integer \in \mathbb{R} \to \mathbb{Z}$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ (\forall\ x : \mathbb{R} \\
\qquad\qquad\qquad\qquad\quad \bullet\ \sim_R 0.5 \leq_R x -_R real\ (real\_to\_integer\ x) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ x -_R real\ (real\_to\_integer\ x) \leq_R 0.5)
\end{array}
$$

_ mod_and _
_ mod_or _

_ mod_xor _ $\vdash \{(\_\ mod\_and\ \_),\ (\_\ mod\_or\ \_),\ (\_\ mod\_xor\ \_)\}$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \subseteq \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ i : \mathbb{N} \bullet i\ mod\_and\ 0 = 0) \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ i : \mathbb{N};\ j : \mathbb{N}_1 \\
\qquad\qquad\qquad\qquad\quad \bullet\ i\ mod\_and\ j \\
\qquad\qquad\qquad\qquad\qquad = 2 * (i\ div\ 2\ mod\_and\ j\ div\ 2) \\
\qquad\qquad\qquad\qquad\qquad\quad + i\ mod\ 2 * (j\ mod\ 2)) \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ i : \mathbb{N} \bullet i\ mod\_or\ 0 = i) \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ i : \mathbb{N};\ j : \mathbb{N}_1 \\
\qquad\qquad\qquad\qquad\quad \bullet\ i\ mod\_or\ j \\
\qquad\qquad\qquad\qquad\qquad = 2 * (i\ div\ 2\ mod\_or\ j\ div\ 2)
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad + max\ \{i\ mod\ 2,\ j\ mod\ 2\}) \\
\qquad \wedge\ (\forall\ i : \mathbb{N} \bullet i\ mod\_xor\ 0 = i) \\
\qquad \wedge\ (\forall\ i : \mathbb{N};\ j : \mathbb{N}_1 \\
\qquad\ \bullet\ i\ mod\_xor\ j \\
\qquad\quad = 2 * (i\ div\ 2\ mod\_xor\ j\ div\ 2) \\
\qquad\qquad + (i + j)\ mod\ 2)
\end{array}
$$

mod_not _ $\vdash (mod\_not\ \_) \in \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$\wedge\ (\forall\ i,\ modulus : \mathbb{Z}$
$\bullet\ mod\_not\ (i,\ modulus) = modulus - (i + 1))$

INTEGER $\vdash INTEGER \in \mathbb{P}\ \mathbb{Z} \wedge true$

INTEGERvSUCC
INTEGERvPRED
INTEGERvPOS
INTEGERvVAL
INTEGERvFIRST
INTEGERvLAST $\vdash (\{INTEGERvFIRST,\ INTEGERvLAST\} \subseteq \mathbb{Z}$
$\wedge\ \{INTEGERvSUCC,$
$INTEGERvPRED,$
$INTEGERvPOS,$
$INTEGERvVAL\}$
$\subseteq \mathbb{Z} \pfun \mathbb{Z})$
$\wedge\ true$

LONG_INTEGER $\vdash LONG\_INTEGER \in \mathbb{P}\ \mathbb{Z} \wedge true$

LONG_INTEGERvSUCC
LONG_INTEGERvPRED
LONG_INTEGERvPOS
LONG_INTEGERvVAL
LONG_INTEGERvFIRST
LONG_INTEGERvLAST
$\vdash (\{LONG\_INTEGERvFIRST,\ LONG\_INTEGERvLAST\} \subseteq \mathbb{Z}$
$\wedge\ \{LONG\_INTEGERvSUCC,$
$LONG\_INTEGERvPRED,$
$LONG\_INTEGERvPOS,$
$LONG\_INTEGERvVAL\}$
$\subseteq \mathbb{Z} \pfun \mathbb{Z})$
$\wedge\ true$

SHORT_INTEGER
$\vdash SHORT\_INTEGER \in \mathbb{P}\ \mathbb{Z} \wedge true$

SHORT_INTEGERvSUCC
SHORT_INTEGERvPRED
SHORT_INTEGERvPOS
SHORT_INTEGERvVAL
SHORT_INTEGERvFIRST
SHORT_INTEGERvLAST
$\vdash (\{SHORT\_INTEGERvFIRST,\ SHORT\_INTEGERvLAST\} \subseteq \mathbb{Z}$
$\wedge\ \{SHORT\_INTEGERvSUCC,$
$SHORT\_INTEGERvPRED,$
$SHORT\_INTEGERvPOS,$
$SHORT\_INTEGERvVAL\}$
$\subseteq \mathbb{Z} \pfun \mathbb{Z})$
$\wedge\ true$

universal_discrete
$\vdash$ *universal_discrete* $\in \mathbb{P}\ \mathbb{Z} \land$ *true*
universal_discretevSUCC
universal_discretevPRED
universal_discretevPOS
universal_discretevVAL
universal_discretevFIRST
universal_discretevLAST
$\vdash$ ($\{$*universal_discretevFIRST*, *universal_discretevLAST*$\}$
$\subseteq \mathbb{Z}$
$\land$ $\{$*universal_discretevSUCC*,
universal_discretevPRED,
universal_discretevPOS,
universal_discretevVAL$\}$
$\subseteq \mathbb{Z} \pfun \mathbb{Z}$)
$\land$ *true*
FLOAT $\vdash$ *FLOAT* $\in \mathbb{P}\ \mathbb{R} \land$ *true*
FLOATvDIGITS
FLOATvFIRST
FLOATvLAST $\vdash$ ($\{$*FLOATvFIRST*, *FLOATvLAST*$\} \subseteq \mathbb{R}$
$\land$ *FLOATvDIGITS* $\in \mathbb{Z}$)
$\land$ *true*
SHORT_FLOAT $\vdash$ *SHORT_FLOAT* $\in \mathbb{P}\ \mathbb{R} \land$ *true*
SHORT_FLOATvDIGITS
SHORT_FLOATvFIRST
SHORT_FLOATvLAST
$\vdash$ ($\{$*SHORT_FLOATvFIRST*, *SHORT_FLOATvLAST*$\} \subseteq \mathbb{R}$
$\land$ *SHORT_FLOATvDIGITS* $\in \mathbb{Z}$)
$\land$ *true*
LONG_FLOAT $\vdash$ *LONG_FLOAT* $\in \mathbb{P}\ \mathbb{R} \land$ *true*
LONG_FLOATvDIGITS
LONG_FLOATvFIRST
LONG_FLOATvLAST
$\vdash$ ($\{$*LONG_FLOATvFIRST*, *LONG_FLOATvLAST*$\} \subseteq \mathbb{R}$
$\land$ *LONG_FLOATvDIGITS* $\in \mathbb{Z}$)
$\land$ *true*
CHARACTERvLAST
$\vdash$ *CHARACTERvLAST* $\in \mathbb{Z} \land$ *CHARACTERvLAST* $\geq$ *127*
STRING $\vdash$ *STRING* $\in \mathbb{P}$ (*POSITIVE* $\pfun$ *CHARACTER*) $\land$ *true*
dest_char $\vdash$ *dest_char* $\in \mathbb{S} \to \mathbb{Z}$
$\land$ ($\forall$ *ch* : $\mathbb{S}$ $\bullet$ *dest_char ch* $= \ulcorner \mathbb{NZ}$ (*RepChar ch*)$\urcorner$)
string_lit _ $\vdash$ (*string_lit _*) $\in$ *Z_STRING* $\to$ *seq CHARACTER*
$\land$ ($\forall$ *str* : *Z_STRING*
$\bullet$ *string_lit str* $=$ *dest_char* $\circ$ *str*)
char_lit _ $\vdash$ (*char_lit _*) $\in$ *Z_CHAR* $\to$ *CHARACTER*
$\land$ (*char_lit _*) $=$ *head* $\circ$ (*string_lit _*)
_ $\&_{\mathbf{2}}$ _
_ $\&_{\mathbf{1}}$ _
_ $\&_{\mathbf{0}}$ _ $\vdash$ $[X](((\_\ \&_0\ \_)[X] \in (\mathbb{Z} \pfun X) \times (\mathbb{Z} \pfun X) \to \mathbb{Z} \pfun X$
$\land$ $(\_\ \&_1\ \_)[X] \in (\mathbb{Z} \pfun X) \times X \to \mathbb{Z} \pfun X$
$\land$ $(\_\ \&_2\ \_)[X] \in X \times (\mathbb{Z} \pfun X) \to \mathbb{Z} \pfun X$)

$$
\begin{array}{l}
\quad \wedge\ (\forall\ a,\ b : \mathbb{Z} \pfun X;\ m,\ n : \mathbb{Z} \\
\quad\quad \mid dom\ a \mapsto m \in max \wedge dom\ b \mapsto n \in min \\
\quad\quad \bullet\ (\_\ \&_0\ \_)[X]\ (a,\ b) \\
\quad\quad\quad = a \oplus \{i : dom\ b \bullet i + m + 1 - n \mapsto b\ i\}) \\
\quad \wedge\ (\forall\ a : \mathbb{Z} \pfun X \bullet (\_\ \&_0\ \_)[X]\ (a, \varnothing) = a) \\
\quad \wedge\ (\forall\ a : \mathbb{Z} \pfun X;\ x : X \\
\quad\quad \bullet\ (\_\ \&_1\ \_)[X]\ (a,\ x) = (\_\ \&_0\ \_)[X]\ (a,\ \langle x \rangle)) \\
\quad \wedge\ (\forall\ a : \mathbb{Z} \pfun X;\ x : X \\
\quad\quad \bullet\ (\_\ \&_2\ \_)[X]\ (x,\ a) = (\_\ \&_0\ \_)[X]\ (\langle x \rangle,\ a)))
\end{array}
$$

slide

$$
\begin{array}{l}
\vdash [X, \\
\quad Y](slide[X,\ Y] \in (X \pfun Y) \times \mathbb{P}\ X \to X \pfun Y \\
\quad\quad \wedge\ (\forall\ f : X \pfun Y;\ r : \mathbb{P}\ X \\
\quad\quad\quad \mid dom\ f = r \\
\quad\quad\quad \bullet\ slide[X,\ Y]\ (f,\ r) = f))
\end{array}
$$

Boolean

$$
\begin{array}{l}
\vdash Boolean \in \mathbb{B} \to BOOLEAN \\
\quad \wedge\ Boolean\ true = TRUE \\
\quad \wedge\ Boolean\ false = FALSE
\end{array}
$$

VRElsfTrue

VRElsfFalse

$$
\begin{array}{l}
\vdash \{VRElsfTrue,\ VRElsfFalse\} \subseteq VRElsfInd \\
\quad \wedge\ disjoint\ \langle \{VRElsfTrue\}, \{VRElsfFalse\} \rangle \\
\quad \wedge\ (\forall\ W : \mathbb{P}\ VRElsfInd \\
\quad\quad \mid \{VRElsfTrue,\ VRElsfFalse\} \subseteq W \\
\quad\quad \bullet\ VRElsfInd \subseteq W)
\end{array}
$$

VRAny _
VRNull _
VRAssign _
VRSpecVia _
VRSpecToSide _
VRSpecPreIntro
VRSemi _
VREndSemi _
VRIfThen _
VRIfElse _
VREndIf _
VRCaseBranch _
VRCaseOthers _
VREndCase _
VRLoopUndecVia _
VRLoopUndecPreIntro
VRLoopUndecPreToSide
VRLoopUndecToSide _
VRWhileVia _
VRWhilePreIntro
VRWhileWPToSide _
VRWhileToSide _
VRForVia _
VRForPreIntro
VRForPreToSide
VRForToSide _
VRForExitToSide _
VRExitTillIntro

VRExitVia _
VRReturnIntro
VRProcCall _
VRProcCallEnd _
VRProcCallRngIntro
VRLogConToSide _
VRLogConPreIntro
VRLogConTypeMemIntro
VRRefinementBegin _
VRRefinementIntro

$$
\begin{aligned}
\vdash\ & (\{VRSpecPreIntro, \\
& \quad VRLoopUndecPreIntro, \\
& \quad VRLoopUndecPreToSide, \\
& \quad VRWhilePreIntro, \\
& \quad VRForPreIntro, \\
& \quad VRForPreToSide, \\
& \quad VRReturnIntro, \\
& \quad VRProcCallRngIntro, \\
& \quad VRLogConPreIntro, \\
& \quad VRLogConTypeMemIntro, \\
& \quad VRRefinementIntro\} \\
& \quad \subseteq VC\_Route \\
& \land (VRAny\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRNull\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRAssign\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRSpecVia\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRSpecToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRSemi\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VREndSemi\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRIfThen\ \_) \in VRElsfInd \times VC\_Route \rightarrowtail VC\_Route \\
& \land (VRIfElse\ \_) \in VRElsfInd \times VC\_Route \rightarrowtail VC\_Route \\
& \land (VREndIf\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRCaseBranch\ \_) \in \mathbb{N}_1 \times VC\_Route \rightarrowtail VC\_Route \\
& \land (VRCaseOthers\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VREndCase\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRLoopUndecVia\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRLoopUndecToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRWhileVia\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRWhileWPToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRWhileToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRForVia\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRForToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRForExitToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land VRExitTillIntro \in Z\_STRING \rightarrowtail VC\_Route \\
& \land (VRExitVia\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRProcCall\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRProcCallEnd\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRLogConToSide\ \_) \in VC\_Route \rightarrowtail VC\_Route \\
& \land (VRRefinementBegin\ \_) \in VC\_Route \rightarrowtail VC\_Route) \\
& \land disjoint\ \langle ran\ (VRAny\ \_), \\
& \quad ran\ (VRNull\ \_),
\end{aligned}
$$

$$
\begin{array}{l}
\quad ran\ (VRAssign\ \_),\\
\quad ran\ (VRSpecVia\ \_),\\
\quad ran\ (VRSpecToSide\ \_),\\
\quad \{VRSpecPreIntro\},\\
\quad ran\ (VRSemi\ \_),\\
\quad ran\ (VREndSemi\ \_),\\
\quad ran\ (VRIfThen\ \_),\\
\quad ran\ (VRIfElse\ \_),\\
\quad ran\ (VREndIf\ \_),\\
\quad ran\ (VRCaseBranch\ \_),\\
\quad ran\ (VRCaseOthers\ \_),\\
\quad ran\ (VREndCase\ \_),\\
\quad ran\ (VRLoopUndecVia\ \_),\\
\quad \{VRLoopUndecPreIntro\},\\
\quad \{VRLoopUndecPreToSide\},\\
\quad ran\ (VRLoopUndecToSide\ \_),\\
\quad ran\ (VRWhileVia\ \_),\\
\quad \{VRWhilePreIntro\},\\
\quad ran\ (VRWhileWPToSide\ \_),\\
\quad ran\ (VRWhileToSide\ \_),\\
\quad ran\ (VRForVia\ \_),\\
\quad \{VRForPreIntro\},\\
\quad \{VRForPreToSide\},\\
\quad ran\ (VRForToSide\ \_),\\
\quad ran\ (VRForExitToSide\ \_),\\
\quad ran\ VRExitTillIntro,\\
\quad ran\ (VRExitVia\ \_),\\
\quad \{VRReturnIntro\},\\
\quad ran\ (VRProcCall\ \_),\\
\quad ran\ (VRProcCallEnd\ \_),\\
\quad \{VRProcCallRngIntro\},\\
\quad ran\ (VRLogConToSide\ \_),\\
\quad \{VRLogConPreIntro\},\\
\quad \{VRLogConTypeMemIntro\},\\
\quad ran\ (VRRefinementBegin\ \_),\\
\quad \{VRRefinementIntro\}\rangle\\
\wedge\ (\forall\ W : \mathbb{P}\ VC\_Route\\
\quad \mid \{VRSpecPreIntro,\\
\qquad VRLoopUndecPreIntro,\\
\qquad VRLoopUndecPreToSide,\\
\qquad VRWhilePreIntro,\\
\qquad VRForPreIntro,\\
\qquad VRForPreToSide,\\
\qquad VRReturnIntro,\\
\qquad VRProcCallRngIntro,\\
\qquad VRLogConPreIntro,\\
\qquad VRLogConTypeMemIntro,\\
\qquad VRRefinementIntro\}\\
\qquad \cup ((VRAny\ \_)\ (\!|\ W\ |\!)\\
\qquad\quad \cup ((VRNull\ \_)\ (\!|\ W\ |\!)\\
\qquad\qquad \cup ((VRAssign\ \_)\ (\!|\ W\ |\!)
\end{array}
$$

$$
\begin{array}{l}
\cup ((VRSpecVia\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad \cup ((VRSpecToSide\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad \cup ((VRSemi\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad \cup ((VREndSemi\ \_) \lhd\!\rhd \\
\quad\quad\quad\quad W \rhd\!\lhd \\
\quad\quad\quad\quad \cup ((VRIfThen\ \_) \lhd\!\rhd \\
VRElsfInd \times W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad \cup ((VRIfElse\ \_) \lhd\!\rhd \\
VRElsfInd \times W \rhd\!\lhd \\
\cup ((VREndIf\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad \cup ((VRCaseBranch\ \_) \lhd\!\rhd \mathbb{N}_1 \times W \rhd\!\lhd \\
\quad\quad \cup ((VRCaseOthers\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad \cup ((VREndCase\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad \cup ((VRLoopUndecVia\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad\quad \cup ((VRLoopUndecToSide\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad \cup ((VRWhileVia\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad\quad \cup ((VRWhileWPToSide\ \_) \lhd\!\rhd \\
\quad\quad\quad\quad\quad\quad\quad\quad W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad\quad\quad \cup ((VRWhileToSide\ \_) \lhd\!\rhd \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \cup ((VRForVia\ \_) \lhd\!\rhd \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \cup ((VRForToSide\ \_) \lhd\!\rhd \\
W \rhd\!\lhd \\
\cup ((VRForExitToSide\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad \cup (VRExitTillIntro \lhd\!\rhd Z\_STRING \rhd\!\lhd \\
\quad\quad \cup ((VRExitVia\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad \cup ((VRProcCall\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad \cup ((VRProcCallEnd\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad\quad \cup ((VRLogConToSide\ \_) \lhd\!\rhd W \rhd\!\lhd \\
\quad\quad\quad\quad\quad\quad \cup (VRRefinementBegin\ \_) \lhd\!\rhd \\
\quad\quad\quad\quad\quad\quad W \rhd\!\lhd)))))))))))))))))))))))))))) \\
\quad\quad \subseteq W \\
\quad \bullet\ VC\_Route \subseteq W)
\end{array}
$$

array_agg2

$$
\begin{array}{l}
\vdash [g1, \\
g2, \\
g](array\_agg2[g1,\ g2,\ g] \\
\quad \in (g1 \to g2 \to g) \to g1 \times g2 \to g \\
\quad \land (\forall f : g1 \to g2 \to g;\ x1 : g1;\ x2 : g2 \\
\quad\quad \bullet\ array\_agg2[g1,\ g2,\ g]\ f\ (x1,\ x2) = f\ x1\ x2))
\end{array}
$$

array_agg3

$$
\begin{array}{l}
\vdash [g1, \\
g2, \\
g3, \\
g](array\_agg3[g1,\ g2,\ g3,\ g] \\
\quad \in (g1 \to g2 \to g3 \to g) \to g1 \times g2 \times g3 \to g \\
\quad \land (\forall f : g1 \to g2 \to g3 \to g; \\
\quad\quad x1 : g1; \\
\quad\quad x2 : g2; \\
\quad\quad x3 : g3 \\
\quad\quad \bullet\ array\_agg3[g1,\ g2,\ g3,\ g]\ f\ (x1,\ x2,\ x3) \\
\quad\quad = f\ x1\ x2\ x3))
\end{array}
$$

array_agg4 $\vdash [g1,$
$g2,$
$g3,$
$g4,$
$g](array\_agg4[g1,\ g2,\ g3,\ g4,\ g]$
$\in (g1 \to g2 \to g3 \to g4 \to g) \to g1 \times g2 \times g3 \times g4 \to g$
$\land (\forall f : g1 \to g2 \to g3 \to g4 \to g;$
$x1 : g1;$
$x2 : g2;$
$x3 : g3;$
$x4 : g4$
$\bullet\ array\_agg4[g1,\ g2,\ g3,\ g4,\ g]\ f$
$(x1,\ x2,\ x3,\ x4)$
$= f\ x1\ x2\ x3\ x4))$

array_agg5 $\vdash [g1,$
$g2,$
$g3,$
$g4,$
$g5,$
$g](array\_agg5[g1,\ g2,\ g3,\ g4,\ g5,\ g]$
$\in (g1 \to g2 \to g3 \to g4 \to g5 \to g)$
$\to g1 \times g2 \times g3 \times g4 \times g5 \to g$
$\land (\forall f : g1 \to g2 \to g3 \to g4 \to g5 \to g;$
$x1 : g1;$
$x2 : g2;$
$x3 : g3;$
$x4 : g4;$
$x5 : g5$
$\bullet\ array\_agg5[g1,\ g2,\ g3,\ g4,\ g5,\ g]\ f$
$(x1,\ x2,\ x3,\ x4,\ x5)$
$= f\ x1\ x2\ x3\ x4\ x5))$

7.3.5 Definitions

FALSE $\vdash FALSE = 0$
TRUE $\vdash TRUE = 1$
BOOLEAN $\vdash BOOLEAN = FALSE\ ..\ TRUE$
BOOLEANvFIRST
$\vdash BOOLEANvFIRST = FALSE$
BOOLEANvLAST $\vdash BOOLEANvLAST = TRUE$
BOOLEANvSUCC $\vdash BOOLEANvSUCC = (BOOLEAN \setminus \{BOOLEANvLAST\}) \lhd succ$
BOOLEANvPRED $\vdash BOOLEANvPRED = BOOLEANvSUCC\ ^{\sim}$
BOOLEANvPOS $\vdash BOOLEANvPOS = id\ BOOLEAN$
BOOLEANvVAL $\vdash BOOLEANvVAL = BOOLEANvPOS\ ^{\sim}$
NATURAL $\vdash NATURAL = 0\ ..\ INTEGERvLAST$
NATURALvFIRST
$\vdash NATURALvFIRST = 0$
NATURALvLAST $\vdash NATURALvLAST = INTEGERvLAST$
NATURALvSUCC $\vdash NATURALvSUCC = INTEGERvSUCC$
NATURALvPRED $\vdash NATURALvPRED = INTEGERvPRED$
NATURALvPOS $\vdash NATURALvPOS = INTEGERvPOS$

NATURALvVAL $\vdash NATURALvVAL = INTEGERvVAL$
POSITIVE $\vdash POSITIVE = 1 \mathinner{..} INTEGERvLAST$
POSITIVEvFIRST
$\vdash POSITIVEvFIRST = 1$
POSITIVEvLAST
$\vdash POSITIVEvLAST = INTEGERvLAST$
POSITIVEvSUCC
$\vdash POSITIVEvSUCC = INTEGERvSUCC$
POSITIVEvPRED
$\vdash POSITIVEvPRED = INTEGERvPRED$
POSITIVEvPOS $\vdash POSITIVEvPOS = INTEGERvPOS$
POSITIVEvVAL $\vdash POSITIVEvVAL = INTEGERvVAL$
CHARACTERvFIRST
$\vdash CHARACTERvFIRST = 0$
CHARACTER $\vdash CHARACTER = CHARACTERvFIRST \mathinner{..} CHARACTERvLAST$
CHARACTERvSUCC
$\vdash CHARACTERvSUCC$
$= (CHARACTER \setminus \{CHARACTERvLAST\}) \lhd succ$
CHARACTERvPRED
$\vdash CHARACTERvPRED = CHARACTERvSUCC^{\sim}$
CHARACTERvPOS
$\vdash CHARACTERvPOS = id\ CHARACTER$
CHARACTERvVAL
$\vdash CHARACTERvVAL = CHARACTERvPOS^{\sim}$
Z_CHAR $\vdash Z\_CHAR = seq\ \mathbb{S}$
Z_STRING $\vdash Z\_STRING = seq\ \mathbb{S}$
Informal_Function
$\vdash Informal\_Function = \mathbb{U}$
VRElsfInd $\vdash VRElsfInd = \mathbb{U}$
VC_Route $\vdash VC\_Route = \mathbb{U}$

7.3.6 Theorems

cn_boolean_thm
$\vdash BOOLEAN = \{FALSE, TRUE\}$
cn_boolean_succ_thm
$\vdash BOOLEANvSUCC = \{FALSE \mapsto TRUE\}$
cn_boolean_pred_thm
$\vdash BOOLEANvPRED = \{TRUE \mapsto FALSE\}$
cn_boolean_pos_thm
$\vdash BOOLEANvPOS = id\ BOOLEAN$
cn_boolean_val_thm
$\vdash BOOLEANvVAL = id\ BOOLEAN$
cn_¬_true_eq_false_thm
$\vdash \neg\ TRUE = FALSE$
cn_boolean_cases_thm
$\vdash \forall\ x : BOOLEAN \bullet x = TRUE \lor x = FALSE$
cn_boolean_clauses
$\vdash TRUE = Boolean\ true$
$\land\ FALSE = Boolean\ false$
$\land\ (\forall\ p : \mathbb{U} \bullet not\ Boolean\ p = Boolean\ (\neg\ p))$

$$
\begin{array}{l}
\quad \wedge\ (\forall\ p,\ q : \mathbb{U} \\
\quad\quad \bullet\ Boolean\ p\ and\ Boolean\ q = Boolean\ (p \wedge q)) \\
\quad \wedge\ (\forall\ p,\ q : \mathbb{U} \\
\quad\quad \bullet\ Boolean\ p\ or\ Boolean\ q = Boolean\ (p \vee q)) \\
\quad \wedge\ (\forall\ p,\ q : \mathbb{U} \\
\quad\quad \bullet\ Boolean\ p\ xor\ Boolean\ q = Boolean\ (\neg\ (p \Leftrightarrow q))) \\
\quad \wedge\ (\forall\ p,\ q : \mathbb{U} \bullet Boolean\ p = Boolean\ q \Leftrightarrow p \Leftrightarrow q)
\end{array}
$$

cn_and_then_or_else_clauses

$$
\begin{array}{l}
\vdash\ \forall\ x,\ y : \mathbb{U} \\
\quad \bullet\ x\ and\_then\ y = x\ and\ y \wedge x\ or\_else\ y = x\ or\ y
\end{array}
$$

cn_boolean_∈_boolean_thm

$$
\vdash\ \forall\ x : \mathbb{U} \bullet Boolean\ x \in BOOLEAN
$$

cn_relational_clauses

$$
\begin{array}{l}
\vdash\ (\forall\ x : \mathbb{U};\ S : \mathbb{U} \bullet x\ mem\ S = Boolean\ (x \in S)) \\
\quad \wedge\ (\forall\ x : \mathbb{U};\ S : \mathbb{U} \bullet x\ notmem\ S = Boolean\ (\neg\ x \in S)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ eq\ y = Boolean\ (x = y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ noteq\ y = Boolean\ (\neg\ x = y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ less\ y = Boolean\ (x < y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ less\_eq\ y = Boolean\ (x \leq y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ greater\ y = Boolean\ (x > y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \bullet x\ greater\_eq\ y = Boolean\ (x \geq y))
\end{array}
$$

cn_relational_clauses1

$$
\begin{array}{l}
\vdash\ (\forall\ x,\ y : \mathbb{U} \bullet x\ real\_less\ y = Boolean\ (x <_R y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \\
\quad\quad \bullet\ x\ real\_less\_eq\ y = Boolean\ (x \leq_R y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \\
\quad\quad \bullet\ x\ real\_greater\ y = Boolean\ (x >_R y)) \\
\quad \wedge\ (\forall\ x,\ y : \mathbb{U} \\
\quad\quad \bullet\ x\ real\_greater\_eq\ y = Boolean\ (x \geq_R y))
\end{array}
$$

cn_integer_to_real_thm

$$
\vdash\ \forall\ x : \mathbb{Z} \bullet integer\_to\_real\ x = real\ x
$$

cn_boolean_clauses1

$$
\begin{array}{l}
\vdash\ (\forall\ x : BOOLEAN \\
\quad\quad \bullet\ not\ x = Boolean\ (\neg\ x = Boolean\ true)) \\
\quad \wedge\ (\forall\ x,\ y : BOOLEAN \\
\quad\quad \bullet\ x\ and\ y \\
\quad\quad\quad = Boolean \\
\quad\quad\quad\quad (x = Boolean\ true \wedge y = Boolean\ true)) \\
\quad \wedge\ (\forall\ x,\ y : BOOLEAN \\
\quad\quad \bullet\ x\ or\ y \\
\quad\quad\quad = Boolean \\
\quad\quad\quad\quad (x = Boolean\ true \vee y = Boolean\ true)) \\
\quad \wedge\ (\forall\ x,\ y : BOOLEAN \\
\quad\quad \bullet\ x\ xor\ y \\
\quad\quad\quad = Boolean \\
\quad\quad\quad\quad (\neg\ x = Boolean\ true \Leftrightarrow y = Boolean\ true))
\end{array}
$$

cn_boolean_clauses2

$$
\begin{array}{l}
\vdash\ (\forall\ x : BOOLEAN;\ p : \mathbb{U} \\
\quad\quad \bullet\ x\ and\ Boolean\ p \\
\quad\quad\quad = Boolean\ (x = Boolean\ true \wedge p)) \\
\quad \wedge\ (\forall\ x : BOOLEAN;\ p : \mathbb{U}
\end{array}
$$

$$
\begin{array}{l}
\quad \bullet\ Boolean\ p\ and\ x \\
\quad\quad = Boolean\ (p \wedge x = Boolean\ true)) \\
\wedge\ (\forall\ x : BOOLEAN;\ p : \mathbb{U} \\
\quad \bullet\ x\ or\ Boolean\ p \\
\quad\quad = Boolean\ (x = Boolean\ true \vee p)) \\
\wedge\ (\forall\ x : BOOLEAN;\ p : \mathbb{U} \\
\quad \bullet\ Boolean\ p\ or\ x \\
\quad\quad = Boolean\ (p \vee x = Boolean\ true)) \\
\wedge\ (\forall\ x : BOOLEAN;\ p : \mathbb{U} \\
\quad \bullet\ x\ xor\ Boolean\ p \\
\quad\quad = Boolean\ (\neg\ x = Boolean\ true \Leftrightarrow p)) \\
\wedge\ (\forall\ x : BOOLEAN;\ p : \mathbb{U} \\
\quad \bullet\ Boolean\ p\ xor\ x \\
\quad\quad = Boolean\ (\neg\ p \Leftrightarrow x = Boolean\ true))
\end{array}
$$

cn_intdiv_0_thm

$$
\begin{array}{l}
\vdash \forall\ j : \mathbb{Z} \\
\quad \mid \neg \\
\quad j = 0 \\
\quad \bullet\ 0\ intdiv\ j = 0 \wedge 0\ rem\ j = 0 \wedge 0\ intmod\ j = 0
\end{array}
$$

cn_intdiv_thm

$$
\begin{array}{l}
\vdash \forall\ i,\ j,\ k : \mathbb{Z} \\
\quad \mid \neg \\
\quad j = 0 \\
\quad \bullet\ i\ intdiv\ j = k \\
\quad\quad \Leftrightarrow (\exists\ m : \mathbb{Z} \\
\quad\quad\quad \bullet\ i = k * j + m \\
\quad\quad\quad\quad \wedge\ abs\ m < abs\ j \\
\quad\quad\quad\quad \wedge\ (0 \leq i \wedge 0 \leq m \vee i < 0 \wedge m \leq 0))
\end{array}
$$

cn_rem_thm

$$
\begin{array}{l}
\vdash \forall\ i,\ j,\ k : \mathbb{Z} \\
\quad \mid \neg \\
\quad j = 0 \\
\quad \bullet\ i\ rem\ j = k \\
\quad\quad \Leftrightarrow (\exists\ d : \mathbb{Z} \\
\quad\quad\quad \bullet\ i = d * j + k \\
\quad\quad\quad\quad \wedge\ abs\ k < abs\ j \\
\quad\quad\quad\quad \wedge\ (0 \leq i \wedge 0 \leq k \vee i < 0 \wedge k \leq 0))
\end{array}
$$

cn_intmod_thm

$$
\begin{array}{l}
\vdash \forall\ i,\ j,\ k : \mathbb{Z} \\
\quad \mid \neg \\
\quad j = 0 \\
\quad \bullet\ i\ intmod\ j = k \\
\quad\quad \Leftrightarrow (\exists\ d : \mathbb{Z} \\
\quad\quad\quad \bullet\ i = d * j + k \\
\quad\quad\quad\quad \wedge\ abs\ k < abs\ j \\
\quad\quad\quad\quad \wedge\ (0 \leq j \wedge 0 \leq k \vee j < 0 \wedge k \leq 0))
\end{array}
$$

z_succ$^{\mathbf{n}}$_$\overset{o}{9}$_thm

$$
\begin{array}{l}
\vdash \forall\ x : \mathbb{U};\ y : \mathbb{U} \\
\quad \bullet\ succ\ \overset{x}{\overset{o}{9}}\ y \\
\quad\quad = \{a : \mathbb{U};\ b : \mathbb{U} \\
\quad\quad\quad \mid (1 \leq x \wedge 0 \leq a \vee \neg\ 0 \leq x \wedge 0 - x \leq a \vee x = 0) \\
\quad\quad\quad\quad \wedge\ (a + x,\ b) \in y\}
\end{array}
$$

REFERENCES

[1] John Barnes. *High Integrity Ada — The Spark Approach*. Addison-Wesley, 1997.

[2] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[3] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[4] ANSI/MIL-STD-1815A-1983. *The Annotated Ada Reference Manual*. Karl A. Nyberg, Ada Joint Program Office, 1983.

[5] DRA/CIS/CSE3/TR/94/27. *Specification of the compliance notation for SPARK and Z. (3 Volumes)*. C.M. O'Halloran, C.T. Sennett, and A. Smith, Defence Research Agency, Malvern.

[6] DRA/CIS(SE2)/PROJ/SWI/TR/1/1.1. *A commentary on the specification of the compliance notation for SPARK and Z*. C.M. O'Halloran, C.T. Sennett, and A. Smith, Defence Research Agency, Malvern, 1st November 1995.

[7] DS/FMU/IED/USR004. *ProofPower Tutorial Manual*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[8] DS/FMU/IED/USR005. *ProofPower Description Manual*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[9] DS/FMU/IED/USR010. *ProofPower Evaluator's Guide*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[10] DS/FMU/IED/USR011. *ProofPower Z Tutorial*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[11] ISO/IEC 8652:1983. *Ada Reference Manual*. International Standards Organisation, 1983.

[12] ISO/IEC 8652:1995. *Ada Reference Manual*. International Standards Organisation, 1995.

[13] ISS/HAT/DAZ/USR502. *Compliance Tool — Installation and Operation*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[14] ISS/HAT/DAZ/USR503. *Compliance Tool — Proving VCs*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[15] ISS/HAT/DAZ/USR504. *Compliance Notation — Language Description*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[16] LEMMA1/HOL/USR029. *ProofPower HOL Reference Manual*. Lemma 1 Ltd., `rda@lemma-one.com`.

[17] *Motif 1.2 User's Guide*. Sun Microsystems, Inc.

INDEX