The RISC ProofNavigator Tutorial and Manual[1]

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

November 24, 2008

[1]The hypertext version of this document is available at `http://www.risc.uni-linz.ac.at/research/formal/ProofNavigator/manual`.

Abstract

This document describes the use of the RISC ProofNavigator, an interactive proving assistant for program and system reasoning developed at the Research Institute for Symbolic Computation (RISC).

Contents

Chapter 1

Introduction

This document describes the use of the *RISC ProofNavigator*, an interactive proving assistant for program and system reasoning developed at the Research Institute for Symbolic Computation (RISC).

Background In the last two decades, a variety of interactive proving assistants and automatic theorem provers have emerged, e.g. PVS [12, 14], Isabelle [11, 9], Coq [3, 7], or also the Theorema system developed at RISC [4, 18]; see [19] for a comparative overview. Thus naturally the question arises what exactly the motivation is to develop yet another such tool. The overall context of the work presented in this document is the long-term objective to develop a program and system exploration environment that has a formal reasoning component at its core. Based on a number of use cases derived from the demands of such an environment (some of these cases are presented in this document), the author evaluated from 2004 to 2005 several prominent systems. The results were mixed.

While we achieved some quite good success (most notably with PVS), we also encountered various problems and nuisances, especially with the navigation within proofs, the presentation of proof states, the treatment of arithmetic, and the general interaction of the user with the systems; we frequently found that the elaboration of proofs was more difficult than we considered necessary. Without any doubt, some of these problems were caused by the author's inabilities and could be overcome by more training and experience with the corresponding systems (we only spent a couple of weeks on each) but such a demand already represents a considerable hurdle e.g. in educational scenarios. We felt that the learning curve for using a proving assistant should not be so steep.

From these experiments, we also draw a couple of important conclusions for the pragmatics of using a proving assistant:

- Convenient navigation in large proof trees is essential; the user gets easily lost in large proofs.
- The aggressive simplification of proof state descriptions and their comfortable presentation is important; the user quickly looses the intuition about the high-level proving problem represented by a proof state.
- Decent automation in dealing with arithmetic is important; a subtype relationship between integers and reals simplifies some proofs considerably (compared to the necessity to construct mappings between these types).
- Automatic search for proofs based on elaborate strategies is rarely of much help; typically it is the combination of semi-automatic proof decomposition, critical hints given by the user, and the application of decision procedures for ground theories that shows practical success.

Not all of the existing proving assistants meet these demands equally well; all in all, we were most satisfied with PVS (consequently various concepts in the RISC ProofNavigator were designed after the model of PVS, also its specification language is close to that of PVS). However, the PVS user interface has apparently come of age and the software is not open source which is especially a problem for its integration into a larger context.

The RISC ProofNavigator Based on above investigations the author estimated that it would be fruitful to write from scratch a proof assistant according to his taste. Furthermore, this task should be possible with reasonable effort by making use of existing software that decides about the satisfiability of formulas over certain combinations of ground theories (and potentially performs related tasks such as formula simplification and counterexample generation); the *really* hard core logic and mathematics is in this software, not in the assistant itself. During the last couple of years, a couple of tools for solving this *SMT (satisfiability modulo theories)* problem have emerged, see the recently established SMT-LIB initiative and the associated SMT-COMP solver competition series [17]. After a (rather) short evaluation, we decided to use the *Cooperating Validity Checker Lite (CVCL)* Version 2.0 as a promising candidate [2, 1]; it is open software, supports the most important theories needed for program verification, apparently shows good results, and its specification language is already close to that of PVS.

Thus we started in started in the fall of 2005 with the development of a new proving assistant. This document describes the result of our efforts called *RISC ProofNavigator* [15, 16] that aims to meet the demands addressed above. The system is freely available under the GNU Public License at

```
http://www.risc.uni-linz.ac.at
   /research/formal/software/ProofNavigator
```

It has been reasonably well tested with (also large) verification examples but is certainly not free of bugs; error reports may be sent to the author at

```
Wolfgang.Schreiner@risc.uni-linz.ac.at
```

who commits himself to the maintenance of the software.

While most proving assistants are written in (semi-)declarative languages such as ML, Lisp, or Mathematica, the RISC ProofNavigator is implemented in Java, primarily for the following reasons: this language has free implementations with good performance on virtually every kind of machine, uses a runtime system with garbage collection (in the beginning of the 1990s still the exclusive domain of declarative languages and then a major reason to use these languages), has a rather clear semantics and supports modern programming language principles (a type system with interfaces and inheritance), is well supported by development frameworks, tools, and libraries, and has a large user community, in industry as well as in academia (many students nowadays learn programming in Java). These advantages are (for our purpose) more important than those of declarative languages, such as the simpler declarative semantics (ML) or the possibility to write programs in a rule-based style (Mathematica).

The graphical user interface of the RISC ProofNavigator is written with the help of the Eclipse Standard Widget Toolkit (SWT) which provides a Java wrapper for a native widget set of the underlying machine such that the user interface is responsive and good-looking; the SWT "browser" component is also the core of the system's presentation of proof states (which are rendered as documents containing a combination of XHTML and MathML [6]).

Document Structure The remainder of the document is split in two parts:

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

Third Party Software The RISC ProofNavigator uses the following third party software; detailed references can be found in the README file of the distribution listed on page 111:

- **CVC Lite 2.0**
- **RIACA OpenMath Library 2.0**
- **General Purpose Hash Function Algorithms Library**
- **ANTLR 2.7.6b2**
- **Eclipse Standard Widget Toolkit 3.3**
- **Mozilla Firefox 2.0.X or SeaMonkey 1.1.X**
- **GIMP Toolkit GTK+ 2.X**
- **Sun JDK 5.0**
- **Tango Icon Library**

Many thanks to the respective authors for their great work.

Chapter 2

User Interface

In the following we explain the main points of interaction with the user interface of the RISC ProofNavigator. We assume that the system is appropriately installed (see Appendix C) such that after typing

```
ProofNavigator &
```

a window pops up that displays the startup screen shown in Figure 2.1.

This window has three menus at the top:

File The menu entry "Read File" allows to read a sequence of declarations from a file; "Restart Session" resets the system to its initial state; "Quit" lets the system terminate.

Options The entry "No Automatic Simplification" switches off automatic formula simplifications (which is useful for step-by-step proof presentations), "Automatic Simplification" switches it on again. "Bigger Font" selects a larger display font (which is mainly useful for demonstrations), "Smaller Font" selects the default font again.

Help The entry "Online Manual" displays in the "Declarations" area the hypertext version of this document; the entry "About ProofNavigator" displays the copyright message.

The main area of the window is split into three areas (whose borders may be dragged by the mouse cursor):

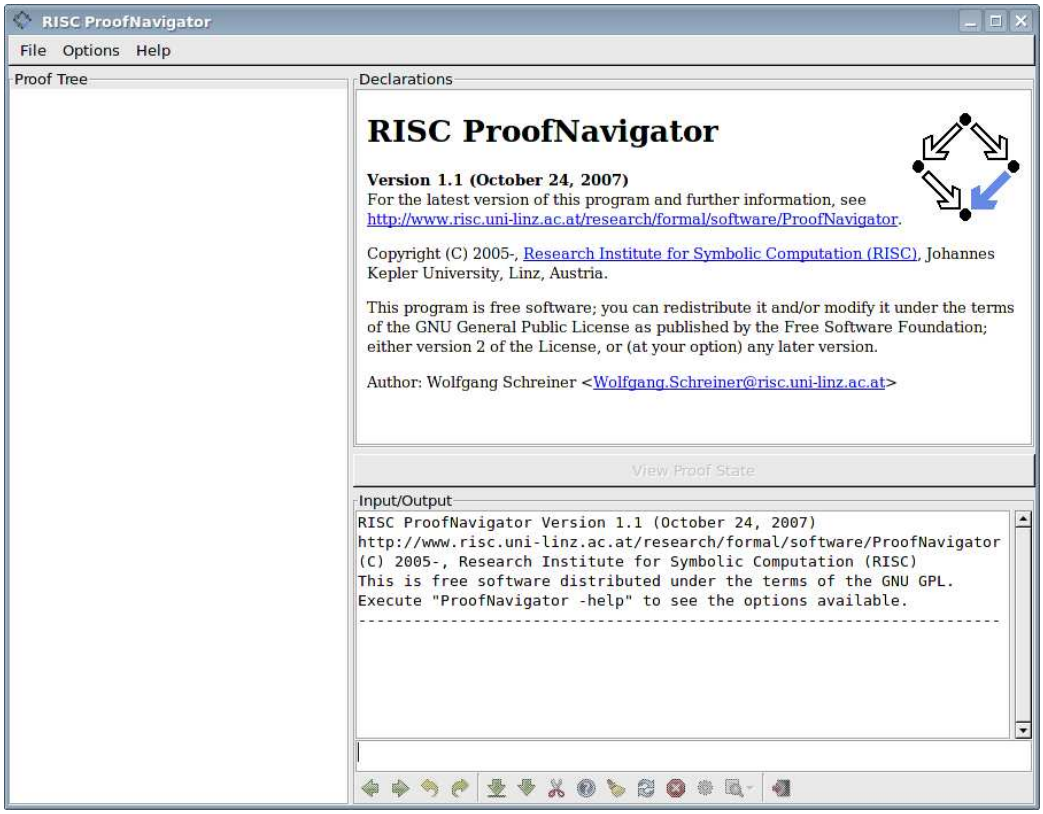


Figure 2.1: ProofNavigator Startup Window

Proof Tree This area illustrates the skeleton structure of the proof which is currently investigated respectively displayed. It mainly serves for easy navigation through a proof.

Declarations This area initially shows a copyright message. Later it displays the declarations entered by the user in a pretty-printed form which closely resembles the usual mathematical notation (while the output window below shows a corresponding plain text notation). In proving mode, this area is labelled "Proof State" and typically displays the open proof state currently investigated by the user (the button "View Declarations" below this area is then enabled to return to the declaration view).

Input/Output This area consists of an *input field* where the user may type in declarations and commands, an *output field* where the effect of the user input is shown as plain text, and a row of (possibly disabled) *buttons*.

In the input field, some keys have a special interpretation:

Arrow Up Go to the previous command in the history buffer (a cyclic buffer of the most recently entered commands).

Arrow Down Go to the next command in the history buffer.

Ctrl+a Go to the begin of the line.

Ctrl+e Go to the end of the line.

Tab Go to the next occurrence of template parameter `[]` (see page 17).

The button row consists of the following elements (from left to right):

Proof Navigation


[Previous Open State] This button triggers the command `prev` described on page 88: the previous element in the list of open proof states becomes the current state.


[Next Open State] This button triggers the command `next` described on page 88: the next element in the list of open proof states becomes the current state.


[Undo Command] This button triggers the command `undo` described on page 88: the effect of the command executed in the parent of the current state is undone.


[Redo Command] This button triggers the command `redo` described on page 88: the effect of the `undo` command that led to the current state is undone.

Proof Control


[Scatter State] This button triggers the command `scatter` described on page 92: applying various proving rules, the current state is scattered into a number of simpler proof states.


[Decompose State] This button triggers the proving command `decompose` described on page 93: applying various proving rules, the formulas in the current proof state are decomposed to yield a single simpler proof state.


[Split State] This button triggers the command `split` described on page 94: applying various proving rules to the goal of the current state, this state is split into a number of proof states with simpler goals.

 **[Generate Counterexample]** This button triggers the command `counterexample` described on page 90: a possible refutation of the current proof state is generated and displayed.

 **[Execute "auto" also in Sibling States]** This button triggers the command `autostar` described on page 97: the command `auto` (see the next button) is applied to the current state and to its subsequent siblings.

 **[Close State by Automatic Formula Instantiation]** This button triggers the command `auto` described on page 98: an attempt is made to close the current state by the automatic instantiation of the quantified formulas in the state.

 **[Simplify State]** This button is only active if "No Automatic Simplification" has been selected in the "Options" menu; it triggers the command `simplify` described on page 99 to simplify all formulas in the current state.

 **[Abort Prover Activity]** This button aborts the current activity of the prover as described on page 91.

 **[Command List]** This button lets a menu pop up that displays all available commands and command templates. By selecting a command from this menu, the command is executed in the current state. By selecting a template from this menu, the template is copied into the input area for instantiating the template parameters before execution.

Proof Exit

 **[Quit Proof]** This button triggers the command `quit` described on page 87: after confirmation by the user, the current proof is terminated and saved to file.

By invoking the system with

```
ProofNavigator --nogui
```

(see Appendix D) the system starts without the graphical user interface in plain text mode: declarations and commands are entered on the command line (i.e. read from the standard input stream) and results are printed in plain text form to the standard output stream. Most system features are also available in this mode[1].

[1] The only major exception is the functionality of the "Abort" button  .

Chapter 3

Examples

In this chapter, we are going to illustrate the features of the RISC ProofNavigator by a series of small examples of specifications and proofs. The examples (that are included in the software distribution) are designed for consecutive study and introduce language and system features on demand, i.e., at those points where they are needed. A systematic presentation of these features is given in the appendix.

3.1 An Induction Proof

Our first example describes the proof of the formula

$$\sum_{i=0}^{n} i = \frac{(n+1)\cdot n}{2}$$

using two axioms that uniquely characterize ("recursively define") the summation quantifier for every (natural number) upper bound of the summation domain:

$$\sum_{i=0}^{0} i = 0$$

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i \ (n > 0)$$

We start our proving session by typing the command

```
newcontext "sum";
```

(note the trailing semicolon) in the input field. This command described on page 83 starts a new session by erasing all previous declarations (if any) and creating a subdirectory `sum` of the current working directory in which the subsequently created proof will be persistently stored after the session. The system is now in "declaration mode" i.e. it accepts declarations that are entered by the user.

We now enter a constant declaration (note the trailing semicolon)

```
sum: NAT->NAT;
```

which introduces a function *sum* from the natural numbers to the natural numbers (denoted by the builtin atomic type *NAT*). This function will take the role of the summation quantifier: given a number n, it shall return $\sum_{i=0}^{n} i$. We correspondingly declare two axioms

```
S1: AXIOM sum(0)=0;
S2: AXIOM FORALL(n:NAT): n>0 => sum(n)=n+sum(n-1);
```

Each axiom consists of a name (e.g. *S1*) and of a formula (a boolean expression) which is from now on assumed true (e.g. *sum(0)=0*). The language for writing formulas includes builtin constants (e.g. *0* and *1*), functions (e.g. + and -), predicates (e.g. = and <), logical connectives (e.g. the implication =>) and quantifiers (e.g. the universal quantifier *FORALL*).

Finally we declare the formula

```
S: FORMULA FORALL(n:NAT): sum(n) = (n+1)*n/2;
```

where the keyword `FORMULA` (rather than `AXIOM`) indicates that the truth of this formula needs proof.

Each time a constant is declared, the "Declarations" area is updated by a pretty-printed version of the declaration such that it ultimately looks as follows:

$\text{sum} \in \mathbb{N} \to \mathbb{N}$

axiom $S1 \equiv \text{sum}(0) = 0$

axiom $S2 \equiv \forall n \in \mathbb{N}: n > 0 \Rightarrow \text{sum}(n) = n + \text{sum}(n-1)$

$S \equiv \forall n \in \mathbb{N}: \text{sum}(n) = \frac{(n+1) \cdot n}{2}$

The box to the left of each declaration is an active element; by moving the mouse cursor over it it turns blue and reveals a menu that exhibits a number of commands applicable to the declaration. In the case of function `sum` and axioms *S1* and *S2* the only commands displayed are for printing the declarations in text form in the output area. For formula *S*, the menu is more interesting:

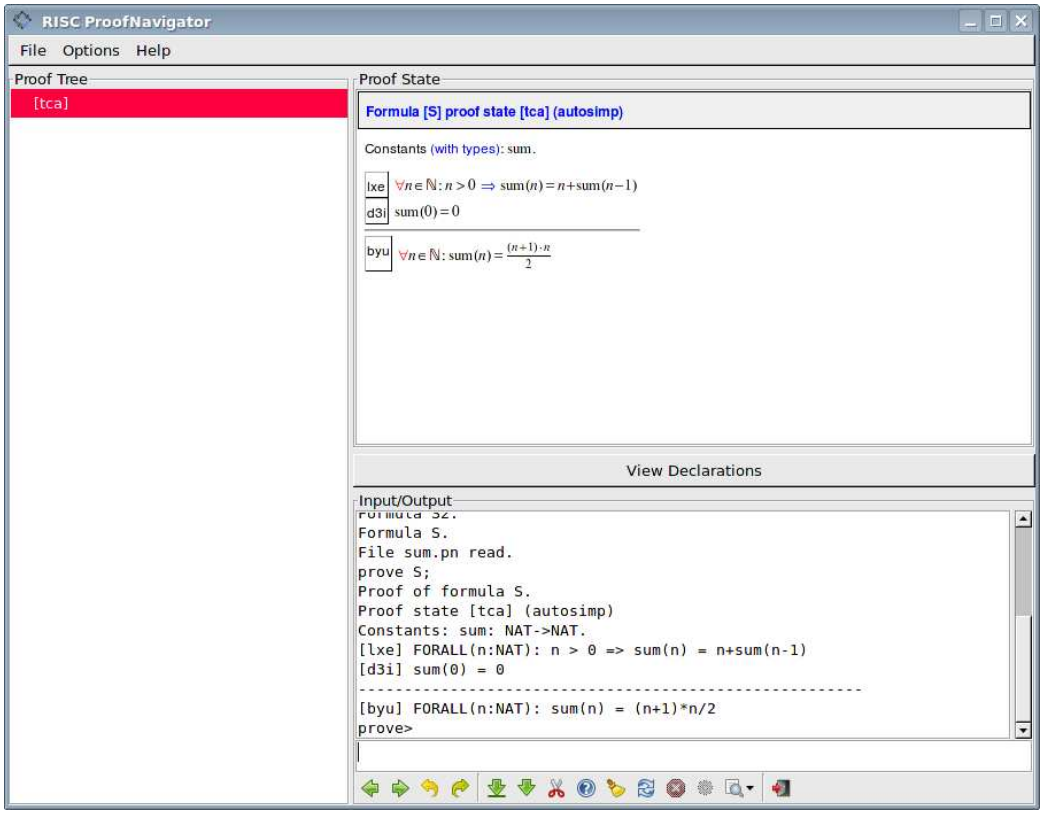
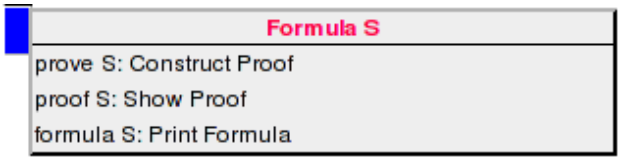


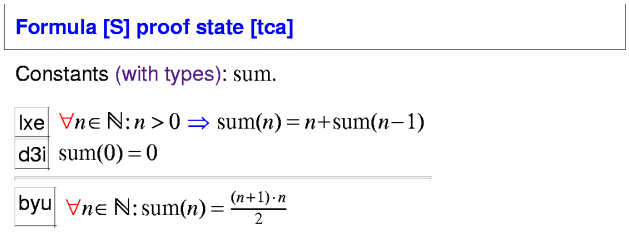
Figure 3.1: The System in Proving Mode



By selecting the menu entry “prove S” (or alternatively typing the command `prove S;` in the input field), the system switches from “declaration mode” to “proving mode” and start a proof of formula *S*. The screen now displays the content shown in Figure 3.1.

Proving the Formula The proof consists of a single state with label [tca] which is indicated as the “current state” by the red line in the “Proof Tree” area. The three-character state label helps to uniquely identify a state within a proof; it is automatically generated and does not carry any meaning.

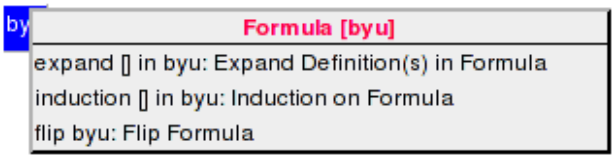
The "Proof State" area presents the state as[1]



The presentation consists of the value constants visible in the state[2] (*sum*) and of two sequences of labelled formulas: the *assumptions* (formulas [lxe] and [d3i]) and the *goals* (formula [byu]) separated by a horizontal line. The assumptions are obviously the contents of the axioms S1 and S2; the goal is the content of the formula S to be proved.

The logical interpretation of this presentation is that of a *sequent* (see page 79); simply put our task is to prove that the *conjunction of the assumptions* (the formulas above the line) logically implies the *disjunction of the goals* (the formulas below the line). In our example, we thus have to prove that from the assumptions [lxe] and [d3i] the goal [byu] follows[3].

A formula label consists typically of three characters that are automatically generated from the text of the formula (such that the same formula in different states has the same label) and does not carry any meaning. Formula labels are active elements; by moving the mouse cursor over a label, it turns blue and reveals a menu with a list of proof commands that are applicable to this formula. For instance, the following menu associated to goal [byu]



lists (among others) the command `induction [] in byu` which applies the proof technique of mathematical induction (see page 95) to formula [byu]; this formula must be a universally quantified formula with a variable from the domain of natural numbers. Actually, the menu entry is just a template with a parameter

[1] The output field displays a plain text description of the state.

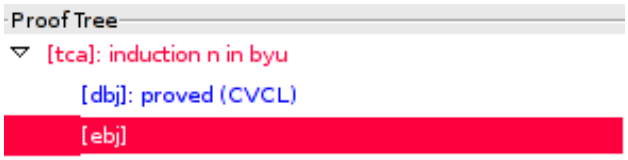
[2] By clicking on the link "with types", one may investigate the types of the constants.

[3] If there is just a single goal, this goal has to be implied. If there is no goal at all, the formula "false" has to be implied (which corresponds to a proof by contradiction).

indicated by the token `[]` that has to be instantiated by the name of the variable on which to run the induction (in general, the quantifier may bind more than one variable). Selecting the template from the menu copies the template into the input area; the cursor is already placed on the parameter[4] such that one just has to type the variable name *n* to get the full command

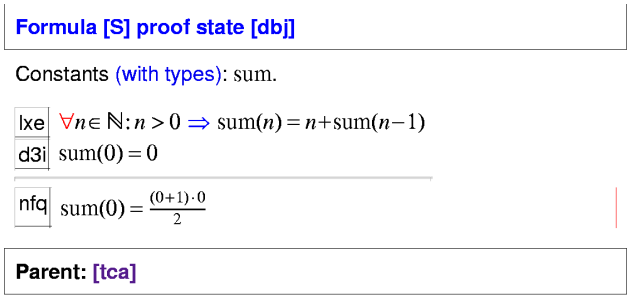
```
induction n in byu;
```

We might have typed in the command also directly on the command line; the formula menu just provided a convenient short-cut which saves us some typing effort (we will discuss later the various possibilities to enter a proof command). When the "Enter" key is pressed, the command is executed which updates the "Proof Tree" area to:



The original proof state [tca] is now labelled by the command "induction n in byu" applied in that state; furthermore it has received two children [dbj] and [ebj] which represent two proof obligations that replace the original obligation: the *induction base* and the *induction step*.

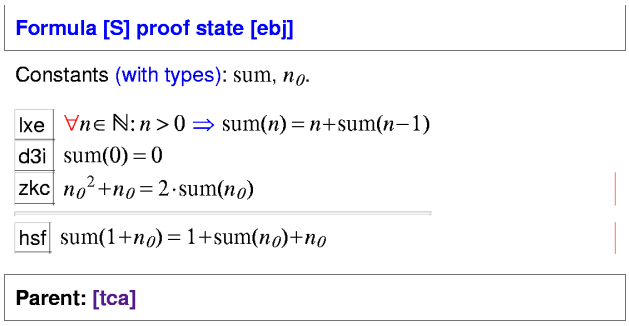
The induction base [dbj] was automatically proved by the external decision procedure CVCL which is indicated by the blue color and the comment "proved (CVCL)". Clicking on the corresponding line in the "Proof Tree" shows this already proved state in more detail:



[4] If the template has multiple occurrences of `[]`, pressing the "Tab" key moves the cursor to the next occurrence.

Apparently the new goal [nfq] is an instantiation of the original goal where the variable n is replaced by the constant 0 (a red bar is shown to the right of the goal to indicate that this is a formula that did not occur in the parent state). Since this goal is (after a bit of simplification) implied by the assumption [d3i], the system was able to prove it without human intervention.

Clicking on the line with label [ebj] in the proof tree returns our focus to the proof state that actually requires our attention:



This state represents the induction step; it contains the declaration of a new natural number constant n_0, an additional assumption [zkc] (representing the version of the original goal where the variable n is instantiated by n_0) and a new goal [hsf] (representing the version of the original goal where the variable n is instantiated by the term $1 + n_0$); red bars are shown to the right of these two formulas to indicate that they are new.

Actually, the system implicitly simplifies all formulas before it presents them to the user. Thus the formulas [zkc] and [hsf] are not syntactically equal but only logically equivalent to the instantiations

$$\begin{aligned} &(1) \quad \mathrm{sum}(n_0) = \tfrac{(n_0+1)\cdot n_0}{2} \\ &(2) \quad \mathrm{sum}(1+n_0) = \tfrac{((1+n_0)+1)\cdot(1+n_0)}{2} \end{aligned}$$

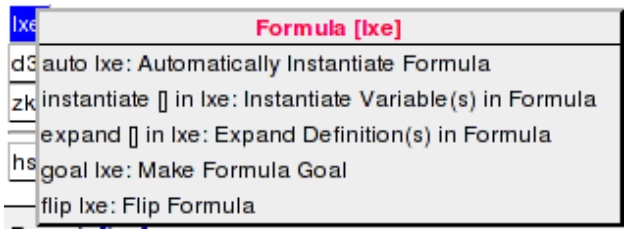
of the original goal. While it is easy to see that formula [zkc] is derived from Equation (1) by simple arithmetic, it is not so easy to see that formula [hsf] is indeed equivalent to Equation (2). The derivation

$$\begin{aligned} &\mathrm{sum}(1+n_0) = \tfrac{((1+n_0)+1)\cdot(1+n_0)}{2} = \tfrac{(n_0+2)\cdot(n_0+1)}{2} \\ &= \tfrac{n_0^2+3n_0+2}{2} = \tfrac{(n_0^2+n_0)+2n_0+2}{2} \overset{(1)}{=} \tfrac{2\mathrm{sum}(n_0)+2n_0+2}{2} \\ &= \mathrm{sum}(n_0) + n_0 + 1 = 1 + \mathrm{sum}(n_0) + n_0 \end{aligned}$$

shows that this is indeed the case under the assumption that Equation (1) holds (which is used in one step of the derivation). Such automatically performed transformations may help to considerably simplify a proof state, but their correctness

may not always be immediately obvious to the user. Later we will also show a version of the proof with automatic simplification turned off.

In order to close the resulting proof state, we have to apply the knowledge contained in assumption [lxe]. By moving the mouse cursor over the label of this formula, a menu is revealed that shows us two possibilities to achieve this:



The menu entry for the command `auto lxe` promises an automatic instantiation of formula [lxe] while the entry for the command template `instantiate [] in lxe` asks for an explicit instantiation term for n. Although we may prefer the simpler `auto lxe`, it is in our example easy to see that the required instantiation term is $n_0 + 1$ such that we may also select the template and instantiate it as follows:

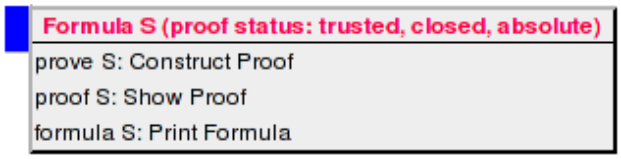
```
instantiate n_0+1 in lxe;
```

(as one can see in the output area, the plain text form of n_0 is `n_0`).

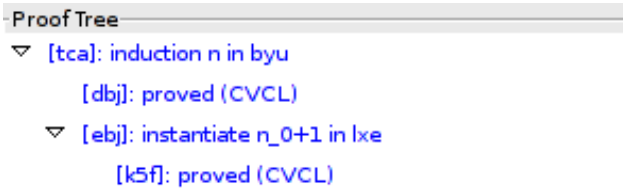
In either case (regardless of choosing `auto` or `instantiate`), the system prints in the output area

```
Proof state [k5f] is closed by decision procedure.
Formula S is proved. QED.
Proof saved (browse file S_index.xhtml).
Quit proof of formula S (use 'proof S' to see proof).
```

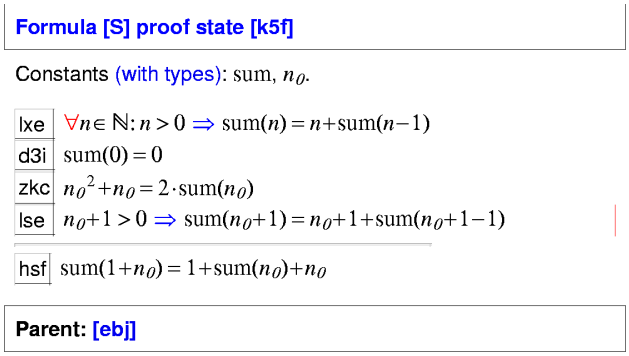
which indicates that formula S has been proved and that the system has returned to declaration mode. The menu of formula S is now



Its head line states that the formula has a proof with the following status (see page 80): the proof is *closed* (it does not contain unproved states), *trusted* (it does not depend on declarations that have been changed since the proof was created), and *absolute* (it does not depend on other unproved formulas). Selecting the command `proof S` from the menu displays in the "Proof Tree" area the structure of this proof:



All proof states are shown in blue indicating the successful completion of the proof. We see that state [ebj] corresponding to the induction step is labelled by the command `instantiate n_0+1 in lxe` which has created a single child state [k5f]; clicking on this state shows its presentation in the "Proof State" area:



This state contains a single additional assumption [lse] representing the version of [lxe] where variable n is instantiated by the term $n_0 + 1$. With the help of this assumption, the external decision procedure CVCL was able to close the proof state as indicated by the state comment "closed (CVCL)". If the command `auto lxe` is applied instead, the proof state contains a large number of automatically deduced instantiations, among them the one shown above.

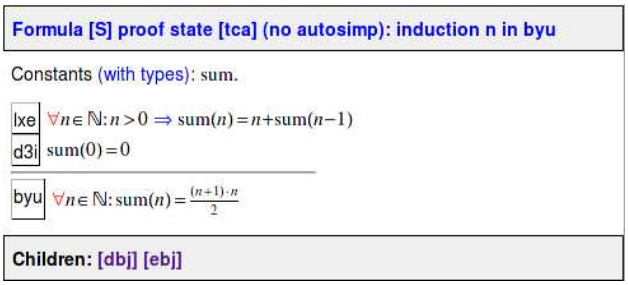
Switching Off Automatic Simplification As we have seen above, automatic transformations of formulas are handy but may sometimes be confusing. We may thus select the menu option "No Automatic Simplifications" which triggers the execution of the command

```
option autosimp="false";
```

which prevents in subsequently created proofs the automatic application of such simplifications (automatic simplification may be again switched on during a proof). We will now repeat above proof in this mode. We start by selecting in the menu of formula *S* the entry "prove S: Construct Proof" and then answer the question

```
Formula S already has a proof
  (proof status: trusted, closed, absolute)
Open existing proof (y/n)? n
```

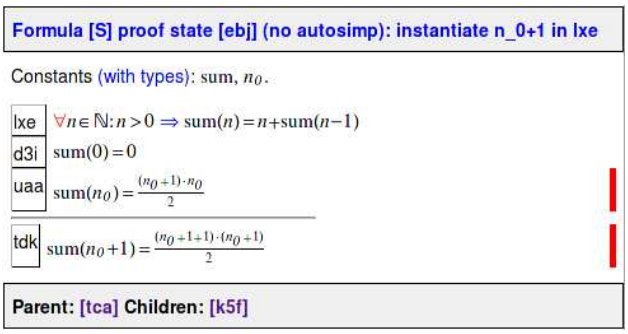
by entering "n". This starts a new proof with the following root state:



The only difference to the root state of the previous proof is the annotation "no autosimp" which indicates that no automatic simplification is performed in the course of this proof. We again execute

```
induction n in byu;
```

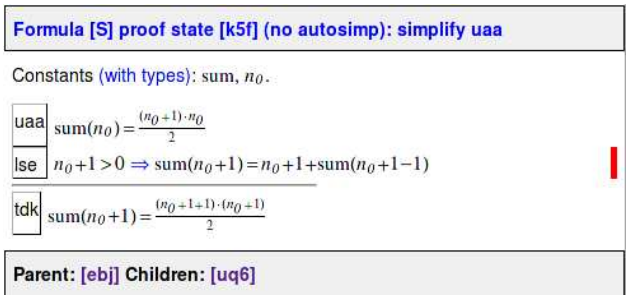
which yields two child states. As before, the first state [dbj] representing the induction base is closed automatically. However, the second state [ebj] representing the induction step now looks as follows:



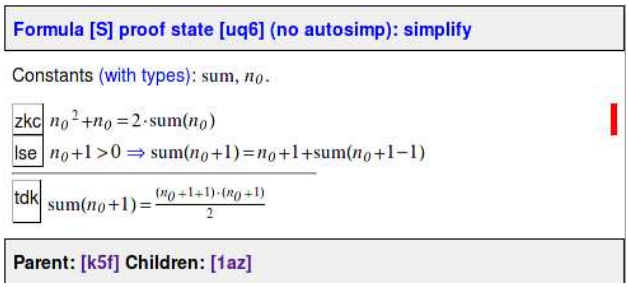
The formula [uaa] representing the induction hypothesis and the formula [tdk] representing the induction goal have not been simplified; they are exact copies of the original formula with the universally quantified variable n replaced by the values n_0 and n_0+1, respectively. If we now execute

```
instantiate n_0+1 in lxe;
```

with get the proof state



where the formula [lse] is the instantiated version of [lxe] without further simplification. We may now select from the label menu of each formula, say [uaa], the entry "Simplify Formula"; this triggers the command `simplify uaa` which yields a new state [uq6] with the simplified formula [zkc]:



Alternatively, we may just press the button (“Simplify State”) which triggers the command `simplify` that simplifies all formulas in the current state (as the default simplification mode always does): indeed, when we press this button, the formulas are simplified in such a way that the external decision procedure recognizes the goal as true and the proof state is closed.

Instead of pressing , we might have also chosen “Automatic Simplification” from the “Options” menu to revert to the default simplification mode (which would have also closed the proof state immediately); by switching the automatic simplification mode selectively on and off, different parts of a (bigger) proof may thus operate in different modes.

Replaying a Proof The software distribution contains in directory `examples` a text file `sum.pn` with the declarations given in this section and a subdirectory `examples/sum` with the proof as shown above. Rather than typing in the declarations manually, one may just go to `examples`, start the system and issue the command

```
read "sum.pn";
```

or just select the file from the GUI's menu entry "File/Read File". The system then reads the file, executes its declarations, and generates the output

```
read "sum.pn";
Value sum:NAT->NAT.
Formula S1.
Formula S2.
Formula S.
Proof read (proof status: trusted, closed, absolute).
File sum.pn read.
```

which indicates that the previously generated proof is now read from file. Actually, only a *proof skeleton* is read consisting of the commands performed to create the proof: while the proof tree may be still displayed by command `proof S`, clicking on the individual nodes of the tree only shows empty proof states. In order to re-generate also the states, the proof has to be *replayed*. This is achieved by simply executing the command `prove S` which will yield the following dialog:

```
prove S;
Formula S already has a (skeleton) proof
  (proof status: trusted, closed, absolute)
Replay skeleton proof (y/n)? y
Proof state [dbj] is closed by decision procedure.
Proof state [k5f] is closed by decision procedure.
Proof replay successful.
Use 'proof S' to see proof.
```

After typing "y" to the question "Replay skeleton proof (y/n)?", the system executes the proof commands while showing in the "Proof Tree" its progress. When the proof has been replayed, also the proof states can be displayed again.

The system keeps automatically track of the status of the declarations on which a proof depends; if some declaration has been changed since the time the proof was generated, the proof status is shown as "untrusted" which indicates that the proof

replay might give errors. Replaying an untrusted proof resets its status to "trusted" (but possibly also from "closed" to "open", because some proof commands may yield errors such that some proof states cannot be closed any more). This automatic tracking of dependencies is a very helpful feature in the production of real-world proof where it is frequently the case that during the proof development some of the declarations change.

This first example should have given a first intuition about the style of interaction with the system. In the next section, we will illustrate more of its features by a more elaborate example.

3.2 A User-Defined Datatype

This section deals with a constructive definition of the datatype "array" and the proof that this definition is adequate with respect to the fundamental properties that we expect of arrays. Such a datatype is useful for verifications of programs operating on arrays; one such verification is shown in the next section. The definition of the datatype also illustrates more features of the specification language of our system.

This specification language already includes a type constructor *ARRAY* such that we can build, given arbitrary types *INDEX* and *ELEM*, the type *ARRAY INDEX OF ELEM*; the elements of this type map every value of type *INDEX* to a value of type *ELEM*. For our purpose, we may define *INDEX* as *NAT* but then encounter the problem that a programming language array has a finite length which needs to be properly represented, too.

The Declarations These considerations lead us to the following *type declarations* (see page 77):

```
INDEX: TYPE = NAT;
ELEM:  TYPE;
ARR:   TYPE = [INDEX, ARRAY INDEX OF ELEM];
```

These declarations introduce a type constant *INDEX* (which is identified with *NAT*), a type constant *ELEM* (which remains undefined and is thus assumed to denote a type different from all other types), and a type constant *ARR*. This type is defined in the declaration as the domain of all binary *tuples* whose first component is of type *INDEX* (representing the length of the array, i.e., the first index that is not in use by the array) and whose second component is of type *ARRAY INDEX*

OF ELEM (representing the actual content of the array, i.e. the mapping of array indices to array elements).

In subsequent *value declarations* (see page 77), we introduce various auxiliary constants whose values remain undefined and that serve as error signals:

```
any:      ARRAY INDEX OF ELEM;
anyelem:  ELEM;
anyarray: ARR;
```

We also define the following auxiliary function constant:

```
content: ARR -> (ARRAY INDEX OF ELEM) =
  LAMBDA(a:ARR): a.1;
```

The declaration of a function constant is just the declaration of a value constant of function type, in our case a constant *content* of type *ARR -> (ARRAY INDEX OF ELEM)*. The value of this function is defined by the *function value expression* (see page 73) *LAMBDA(a:ARR): a.1* which denotes the function that, given an argument *a* of type *ARR*, returns its second component *a.1*, i.e. the content of the array (the notation *a.i* denotes component *i* of tuple *a*; components are numbered starting with 0).

With the help of these auxiliary notions, we define our core functions on arrays:

```
length: ARR -> INDEX =
  LAMBDA(a:ARR): a.0;
new: INDEX -> ARR =
  LAMBDA(n:INDEX): (n, any);
put: (ARR, INDEX, ELEM) -> ARR =
 LAMBDA(a:ARR, i:INDEX, e:ELEM):
   IF i < length(a)
     THEN (length(a), content(a) WITH [i]:=e)
     ELSE anyarray
   ENDIF;
get: (ARR, INDEX) -> ELEM =
  LAMBDA(a:ARR, i:INDEX):
    IF i < length(a)
      THEN content(a)[i]
      ELSE anyelem
    ENDIF;
```

The meaning of these definitions is as follows:

- Function *length* takes an array as its argument and returns the array's first component, i.e., its length.
- Function *new* takes an index (natural number) *n* and returns an array (tuple) of length (first component) *n* and content (second component) *any* (the *tuple expression (a,b)* denotes a tuple of two components *a* and *b*, see page 75).
- Function *put* takes an array *a*, an index *i*, and an element *e*. The value of the function is defined by a *conditional expression* of form *IF E1 THEN E2 ELSE E3 ENDIF*; the expression denotes *E2*, if the boolean value *E1* is true, and *E2*, otherwise (see page 72). Consequently, if *i* is not in the index range of *a*, the function returns the array *anyarray* (signalling an error); otherwise it returns an array that is identical to *a* except that its content at index *i* is *e* (the *array update expression A WITH [I]:=E* takes an *ARRAY* value *A*, an index *I* in the domain of *A*, and an element *E* in the range of *A*; it returns an *ARR* value that is identical to *A* except that it maps *I* to *E*, see page 74).
- Function *get* takes an array *a* and an index *i*. If *i* is not in the index range of *a*, the function returns the element *anyelem* (signalling an error); otherwise it returns the element of the content of *a* at index *i* (the *array value expression A[I]* takes an *ARRAY* value *A* and an index *I* in the domain of *A*; it returns the element of *A* at index *I*, see page 74).

Proving the Array Axioms The adequacy of these definitions is stated by the following *formula declarations* (see page 78):

```
length1: FORMULA
  FORALL(n:INDEX): length(new(n)) = n;

length2: FORMULA
  FORALL(a:ARR, i:INDEX, e:ELEM):
    i < length(a) =>
      length(put(a, i, e)) = length(a);

get1: FORMULA
  FORALL(a:ARR, i:INDEX, e:ELEM):
    i < length(a) => get(put(a, i, e), i) = e;

get2: FORMULA
  FORALL(a:ARR, i, j:INDEX, e:ELEM):
    i < length(a) AND j < length(a) AND i /= j =>
      get(put(a, i, e), j) = get(a, j);
```

These declarations state that the functions defined above obey the laws expected from arrays: *length1* says that the request to allocate an array of length *n* indeed yields an array of this length; *length2* states that putting an element into an array at a valid index does not change the length of the array; *get1* says that consequently looking up this array at that index yields the element put there; *get2* says that looking up this array at any other valid index yields the original element there.

The pretty-printed versions of the declarations are shown below:

```
INDEX∈ type = ℕ
ELEM∈ type
ARR∈ type = [INDEX, array INDEX of ELEM]
any∈ array INDEX of ELEM
anyelem∈ ELEM
anyarray∈ ARR
content∈ ARR→array INDEX of ELEM = λa∈ ARR: a.1
length∈ ARR→INDEX = λa∈ ARR: a.0
new∈ INDEX→ARR = λn∈ INDEX: (n, any)
put∈ (ARR, INDEX, ELEM)→ARR =
   λa∈ ARR, i∈ INDEX, e∈ ELEM:
      if i < length(a) then
         (length(a), content(a) with [i]:=e) else
         anyarray endif
get∈ (ARR, INDEX)→ELEM =
   λa∈ ARR, i∈ INDEX: if i < length(a) then content(a)[i] else anyelem endif
length1 ≡ ∀n∈ INDEX: length(new(n)) = n
length2 ≡
   ∀a∈ ARR, i∈ INDEX, e∈ ELEM: i < length(a) ⇒ length(put(a, i, e)) = length(a)
get1 ≡ ∀a∈ ARR, i∈ INDEX, e∈ ELEM: i < length(a) ⇒ get(put(a, i, e), i) = e
get2 ≡
   ∀a∈ ARR, i∈ INDEX, j∈ INDEX, e∈ ELEM:
      i < length(a) ∧ j < length(a) ∧ i ≠ j ⇒ get(put(a, i, e), j) = get(a, j)
```

The strategy for proving formulas *length1*, *length2*, *get1*, and *get2* is in all cases the same: we expand the constants by their definitions and apply the usual decomposition rules to get rid of the universal quantifier (*FORALL*) and of the logical connectives implication (=>) and conjunction (*AND*). The resulting proof states have only atomic formulas and the system can close these states automatically. We demonstrate this strategy in detail by the proof of the most complex formula *get2* (the other proofs are analogous).

Selecting the command `prove get2` from the menu of formula *get2* yields the initial proof state[5]

Formula [get2] proof state [adu] : expand length, get, put, content

Constants (with types): anyelem, get, length, put, content, anyarray, new, any.

vv6 ∀a∈ ARR, i∈ INDEX, j∈ INDEX, e∈ ELEM:
i < length(a) ∧ j < length(a) ⇒ i = j ∨ get(put(a, i, e), j) = get(a, j)

Children: [c3b]

This state labelled [adu] consists of a single goal [vv6] representing the content of formula *get2*. To expand in this proof state all occurrences of the defined constants to their values, we apply the command (see page 102)

[5]The following screenshots of proof states were taken from the already completed proof; the proof state headers listing the applied proof commands and the links to the generated child states are initially not visible.

```
expand length, get, put, content;
```

The command may be typed in the input area; alternatively, we may press the "Command List" button [ , select from the popup menu the command template `expand []`, instantiate the template parameter and hit the "Enter" key. Even quicker, moving the mouse cursor over the label [vv6] reveals a formula menu from which the more special template `expand [] in vv6` can be selected and instantiated; the resulting command performs instantiations only in this formula (which has in the current state with a single formula the same net effect).

In any case, the expansion yields a single child state [c3b] of the following form:

Formula [get2] proof state [c3b] : scatter

Constants (with types): anyelem, get, length, put, content, anyarray, new, any.

d5q $\forall a \in \mathrm{ARR},\ i \in \mathrm{INDEX},\ j \in \mathrm{INDEX},\ e \in \mathrm{ELEM}:$
$i < a.0 \wedge j < a.0$
$\Rightarrow$
$i = j$
$\vee$
$\mathrm{if}\ j < \mathrm{if}\ i < a.0\ \mathrm{then}\ (a.0,\ a.1\ \mathrm{with}\ [i]{:=}e)\ \mathrm{else}\ \mathrm{anyarray}\ \mathrm{endif}\ .0\ \mathrm{then}\ \mathrm{if}$
$i < a.0\ \mathrm{then}\ (a.0,\ a.1\ \mathrm{with}\ [i]{:=}e)\ \mathrm{else}\ \mathrm{anyarray}\ \mathrm{endif}\ .1[j]\ \mathrm{else}$
$\mathrm{anyelem}\ \mathrm{endif}\ =\ \mathrm{if}\ j < a.0\ \mathrm{then}\ a.1[j]\ \mathrm{else}\ \mathrm{anyelem}\ \mathrm{endif}$

Parent: [adu] Children: [qid]

The state has a single goal [d5q] which is the result of the expansion of all constants in the parent's goal [vv6]. The formula is now a bit clumsy; we become unsure whether we have applied the right strategy and press the "Undo" button to undo the effect (see page 88) of the expansion and return to the parent state [adu] after discarding the generated state [c3b]. On the other hand, we were perhaps too anxious and should unperturbedly proceed our path; pressing the "Redo" button undoes the "Undo" (see page 88) and restores state [c3b]. In real-life proofs, is perhaps the most often pressed button to investigate different proving strategies; the existence of reassures us that the inadvertent use of this button cannot cause any harm.

Our next goal is to simplify the proof state by getting rid of the universal quantifier and of the logical connectives in the goal. The simplest way to achieve this is pressing the "Scatter State" button which applies various proving rules in order to scatter the current state to a number of simpler ones (see page 92). A less aggressive strategy is pursued when pressing the "Decompose State" button which applies fewer rules in order to decompose the formulas in the current state yielding a single child state only (see page 93). These two buttons are frequently applied in the initial stages of a proof; using has the advantage of quickly

reducing a proof to interesting proof situations at the price of giving up explicit control of the proving strategy; it is safer to apply first  to get a simplified version of the current state that can be investigated before scattering.

For our example the choice does not make any difference; pressing any of these buttons yields the dialog

```
Proof state [qid] is closed by decision procedure.
Formula get2 is proved. QED.
Save this proof and overwrite the previous one
  (y/n)? y
Proof saved (browse file get2_index.xhtml).
Quit proof of formula get2
  (use 'proof get2' to see proof).
```

This indicates that the remainder of the proof has been automatically completed and that the system has returned to declaration mode.

By selecting the command `proof get2` from the menu of formula *get2*, we see the structure of the generated proof:

```
[adu]: expand length, get, put, content
  [c3b]: scatter
    [qid]: proved (CVCL)
```

From state [c3b] a single child state was generated that was automatically proved by the external decision procedure CVCL. Clicking on the corresponding node in the tree displays the state as follows:

Formula [get2] proof state [qid] : proved (CVCL)

Constants (with types): a_0, anyelem, get, length, put, content, j_0, new, anyarray, e_0, any, i_0.

kxh $i_0 < a_0.0$
bfe $j_0 < a_0.0$
df4 $j_0 \neq i_0$

2ya if $j_0 < a_0.0$ then $a_0.1[j_0]$ else anyelem endif = if $j_0 <$ if $i_0 < a_0.0$ then ($a_0.0$, $a_0.1$ with $[i_0]:=e_0$) else anyarray endif .0 then if $i_0 < a_0.0$ then ($a_0.0$, $a_0.1$ with $[i_0]:=e_0$) else anyarray endif .1$[j_0]$ else anyelem endif

Parent: [c3b]

The state has four new constants a_0, i_0, j_0, and e_0 that replace the bound variables of the universally quantified goal [d5q] in the parent state (see page 109). The resulting goal without quantifier was decomposed into three atomic formulas as assumptions and a single atomic formula (the equality of two conditional

expressions) as a goal. This proof state was automatically closed by the decision procedure; for a human, the corresponding reasoning steps are (although not really difficult) tedious and error-prone.

This small example already illustrates a general strategy of how to work with the system: to decompose a proof and get rid of quantifiers until sufficiently much low-level knowledge in the form of atomic predicates is available such that a decision procedure can automatically close the proof state. The task of the human (and the difficulty in real-world proofs) is to expose this low-level knowledge by guiding the overall proof construction; the task of the system is to make this process as painless as possible and to take over (via an external decision procedure) low-level reasoning on builtin datatypes (such as `NAT` or `ARRAY`).

Proving the Extensionality Principle We now turn our attention to another interesting property that we expect of arrays and that requires some more work from the user: we would like to prove the "extensionality principle" that two arrays are identical if and only if they have the same length and hold the same elements. For this purpose, we have first to introduce two axioms:

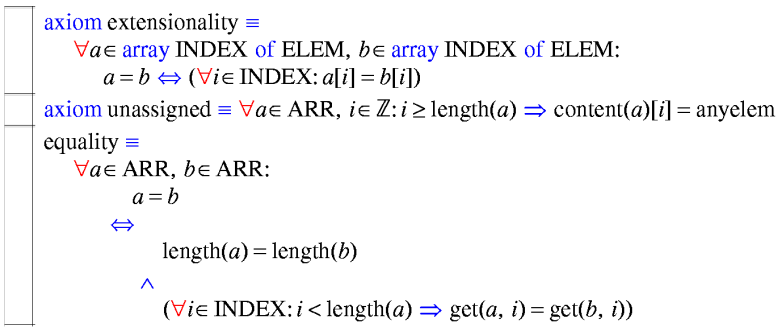
```
extensionality: AXIOM
  FORALL(a, b:ARRAY INDEX OF ELEM):
    a=b <=> (FORALL(i:INDEX):a[i]=b[i]);

unassigned: AXIOM
  FORALL(a:ARR, i:INT):
    (i >= length(a)) => content(a)[i] = anyelem;
```

The first axiom states the extensionality principle on the type constructor `ARRAY` (this principle is not builtin into the system). The second axiom states that we may assume that the content of an array outside the valid index range is the definite but unspecified value *anyelem* such that content of two arrays of same length outside their index range is the same.

With the help of these axioms we are going now to prove the following formula:

```
equality: FORMULA
  FORALL(a:ARR, b:ARR):
    a = b <=>
      length(a) = length(b) AND
      (FORALL(i:INDEX): i < length(a) =>
        get(a,i) = get(b,i));
```

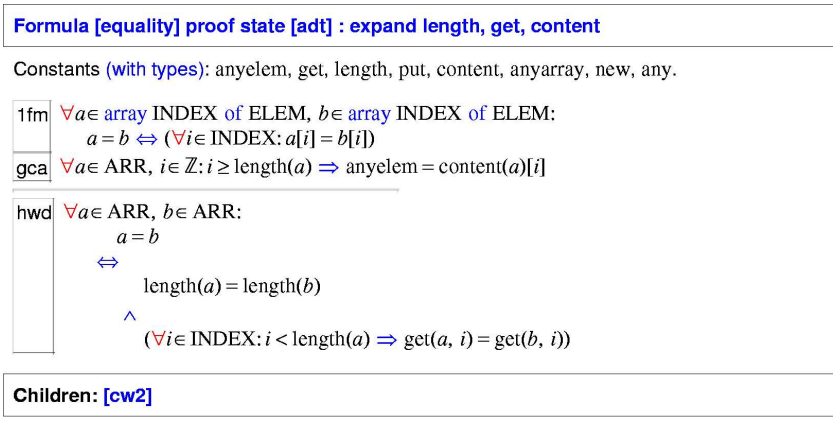
The pretty-printed versions of the declarations are shown below:



To support the understanding of the following presentation, we already show the structure of the proof that we are going to generate:

```
[adt]: expand length, get, content
  [cw2]: scatter
    [qey]: proved (CVCL)
    [rey]: assume b_0.1 = a_0.1
      [zpt]: proved (CVCL)
      [1pt]: instantiate a_0.1, b_0.1 in 1fm
        [y51]: scatter
          [ku2]: auto
            [iub]: proved (CVCL)
```

By selecting “Prove equality” from the menu of formula *equality*, we encounter the initial state [adt] with two assumptions [1fm] and [gca] representing the axioms and the goal [hwd] representing the formula to be proved:



We expand the constant definitions by executing command `expand length, get, content` yielding the state [cw2]:

Formula [equality] proof state [cw2] : scatter

Constants (with types): anyelem, get, length, put, content, anyarray, new, any.

1fm $\forall a \in$ array INDEX of ELEM, $b \in$ array INDEX of ELEM:
$a = b \Leftrightarrow (\forall i \in \text{INDEX}: a[i] = b[i])$

3p3 $\forall a \in \text{ARR}, i \in \mathbb{Z}: i \geq a.0 \Rightarrow \text{anyelem} = a.1[i]$

n14 $\forall a \in \text{ARR}, b \in \text{ARR}:$
$a = b$
$\Leftrightarrow$
$a.0 = b.0$
$\wedge$
$(\forall i \in \text{INDEX}:$
$i < a.0$
$\Rightarrow$
if $i < a.0$ then $a.1[i]$ else anyelem endif $=$ if $i < b.0$ then $b.1[i]$
else anyelem endif)

Parent: [adt] Children: [qey] [rey]

Rather than investigating this state in depth, we aim to quickly push forward to the actual core problem by pressing the "Scatter" button  which yields two child states [qey] and [rey]. While the state [qey] (corresponding to the "left to right" direction $\Rightarrow$ of the proof) is automatically closed, we have to analyze state [rey] (corresponding to the "right to left"direction $\Leftarrow$) in more detail:

Formula [equality] proof state [rey] : assume b_0.1 = a_0.1

Constants (with types): a_0, anyelem, get, length, put, content, new, anyarray, b_0, any.

1fm $\forall a \in$ array INDEX of ELEM, $b \in$ array INDEX of ELEM:
$a = b \Leftrightarrow (\forall i \in \text{INDEX}: a[i] = b[i])$

3p3 $\forall a \in \text{ARR}, i \in \mathbb{Z}: i \geq a.0 \Rightarrow \text{anyelem} = a.1[i]$

ruq $\forall i \in \text{INDEX}:$
$i < a_0.0$
$\Rightarrow$
if $i < a_0.0$ then $a_0.1[i]$ else anyelem endif $=$ if $i < b_0.0$ then $b_0.1[i]$ else anyelem
endif

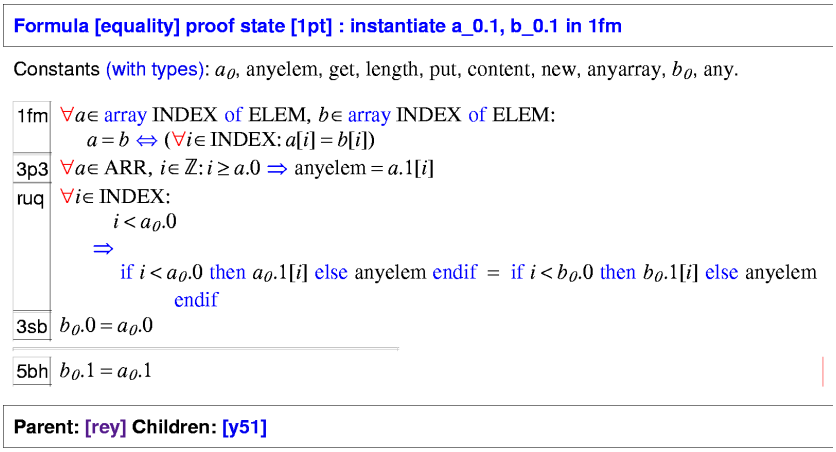
3sb $b_0.0 = a_0.0$

o4i $b_0 = a_0$

Parent: [cw2] Children: [zpt] [1pt]

The state has a single goal [o4i] stating the equality of two arrays a_0 and b_0 (two new constants that were introduced for the universally bound variables in the original goal). The assumption [3sb] states that their first components (the array lengths) are equal. So what is missing to close the state is apparently the knowledge that also their second components (the array contents) are equal. By

executing the command `assume b_0.1 = a_0.1` (see page 100) we can introduce this knowledge as an assumption yielding a new state [zpt] (which is correspondingly automatically closed) and another state [1pt] with a goal [5bh] that represents the obligation is to prove this assumption:



The proof of this goal apparently depends on the knowledge contained in the universally quantified assumptions [1fm], [3p3], and [ruq] such that we may be attempted to ask for an automatic instantiation of these formulas. Actually, the "Auto" button (respectively the command `auto`, see page 98) implements this feature. However, when we press this button, the system executes for a while (if we get impatient, we may abort the execution by pressing the "Abort" button) and finally terminates leaving the current state unchanged, which means that the system could not find the right instantiation to close the proof state.

Thus we have to to rely on our own wit and investigate the assumptions further. We decide that the knowledge expressed in assumption [1fm] for two *ARRAY INDEX OF ELEM* variables a and b actually applies to the two values $a_0.1$ and $b_0.1$ in our goal. We thus select from the menu of [1fm] the template `instantiate [] in 1fm` which we complete to the command `instantiate a_0.1, b_0.1 in 1fm` whose execution yields the following state:

Formula [equality] proof state [y51] : scatter

Constants (with types): a_0, anyelem, get, length, put, content, new, anyarray, b_0, any.

1fm $\forall a \in$ array INDEX of ELEM, $b \in$ array INDEX of ELEM:
$a = b \Leftrightarrow (\forall i \in \text{INDEX}: a[i] = b[i])$

3p3 $\forall a \in \text{ARR}, i \in \mathbb{Z}: i \geq a.0 \Rightarrow \text{anyelem} = a.1[i]$

ruq $\forall i \in$ INDEX:
$i < a_0.0$
$\Rightarrow$
if $i < a_0.0$ then $a_0.1[i]$ else anyelem endif = if $i < b_0.0$ then $b_0.1[i]$ else anyelem endif

3sb $b_0.0 = a_0.0$

2sq $\exists i \in \text{INDEX}: a_0.1[i] \neq b_0.1[i]$

5bh $b_0.1 = a_0.1$

Parent: [1pt] Children: [ku2]

This state contains an existentially quantified assumption [2sq]; by pressing the "Scatter" button  (or just the more predictable "Decompose" ) we get the following state with assumption [lhm] that represents the version of the previous assumption [2sq] where a new constant i_0 replaces the variable i:

Formula [equality] proof state [ku2] : auto

Constants (with types): a_0, anyelem, get, length, put, content, anyarray, new, b_0, any, i_0.

1fm $\forall a \in$ array INDEX of ELEM, $b \in$ array INDEX of ELEM:
$a = b \Leftrightarrow (\forall i \in \text{INDEX}: a[i] = b[i])$

3p3 $\forall a \in \text{ARR}, i \in \mathbb{Z}: i \geq a.0 \Rightarrow \text{anyelem} = a.1[i]$

ruq $\forall i \in$ INDEX:
$i < a_0.0$
$\Rightarrow$
if $i < a_0.0$ then $a_0.1[i]$ else anyelem endif = if $i < b_0.0$ then $b_0.1[i]$ else anyelem endif

3sb $b_0.0 = a_0.0$

lhn $a_0.1[i_0] \neq b_0.1[i_0]$

5bh $b_0.1 = a_0.1$

Parent: [y51] Children: [iub]

Lazy as can be, we again try automatic instantiation with the "Auto" button  and this time get the now already familiar termination dialog

```
Proof state [iub] is closed by decision procedure.
Formula equality is proved. QED.
Save this proof and overwrite the previous one
  (y/n)? y
Proof saved (browse file equality_index.xhtml).
Quit proof of formula equality
  (use 'proof equality' to see proof).
```

with the system returning to declaration mode. Selecting "Proof equality" from the menu of formula *equality* shows the proof skeleton already depicted above.

As we can see from this proof, the automatic instantiation of universally quantified assumptions (or, dually, existentially quantified goals) is not a "cure for all" strategy. The system applies a very simple strategy to instantiate such formulas by a limited number of suitable terms and then attempts to close the resulting proof state by a decision procedure that takes into account the formulas without quantifiers only; if the right combination of instantiations cannot be found (which gets more and more unlikely, the larger the number of quantified formulas in the proof state is), the user must provide (at least some of) the "right instantiations" on her own, which requires creativity and insight into the proof.

In the next section, we will investigate several proofs that require more such insight from the user.

3.3 A Program Verification

Our next goal is the verification of a small program that represents the core of *linear search*: the program finds the first index r at which a value x occurs in an array a; r is -1, if x does not occur in a:

$$
\begin{array}{l}
\{olda = a \wedge oldx = x \wedge n = \mathit{length}(a) \wedge i = 0 \wedge r = -1\} \\
\textbf{while } i < n \wedge r = -1 \textbf{ do} \\
\quad \textbf{if } a[i] = x \\
\qquad \textbf{then } r := i \\
\qquad \textbf{else } i := i + 1 \\
\{a = olda \wedge x = oldx \wedge \\
\quad ((r = -1 \wedge \forall i : 0 \leq i < \mathit{length}(a) \Rightarrow a[i] \neq x) \vee \\
\quad\ (0 \leq r < \mathit{length}(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x))\}
\end{array}
$$

Above program specification is given in the form of an *Hoare triple* [8] of the form $\{I\}P\{O\}$ which states the *partial correctness* of P: that, if the input condition I holds before the execution of P, then the output condition shall O hold afterwards (provided that P terminates).

The Verification Conditions By the rules of the Hoare calculus [8], we can derive from above triple four *verification conditions* A, B_1, B_2, and C that have to be proved:

$$
\begin{aligned}
&\mathit{Input} :\Leftrightarrow \mathit{olda} = a \wedge \mathit{oldx} = x \wedge n = \mathit{length}(a) \wedge i = 0 \wedge r = -1\\
&\mathit{Output} :\Leftrightarrow a = \mathit{olda} \wedge x = \mathit{oldx} \wedge\\
&\quad ((r = -1 \wedge \forall i : 0 \leq i < \mathit{length}(a) \Rightarrow a[i] \neq x) \vee\\
&\quad\ (0 \leq r < \mathit{length}(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x))\\
&\mathit{Invariant} :\Leftrightarrow \mathit{olda} = a \wedge \mathit{oldx} = x \wedge n = \mathit{length}(a) \wedge\\
&\quad 0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge\\
&\quad (r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))
\end{aligned}
$$

$$
\begin{aligned}
&A :\Leftrightarrow \mathit{Input} \Rightarrow \mathit{Invariant}\\
&B_1 :\Leftrightarrow \mathit{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] = x \Rightarrow \mathit{Invariant}[i/r]\\
&B_2 :\Leftrightarrow \mathit{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] \neq x \Rightarrow \mathit{Invariant}[i+1/i]\\
&C :\Leftrightarrow \mathit{Invariant} \wedge \neg(i < n \wedge r = -1) \Rightarrow \mathit{Output}
\end{aligned}
$$

Condition A states that the input condition establishes the loop invariant (a condition that is true before and after each iteration of the loop), conditions B_1 and B_2 state that the invariant is preserved by both branches of the conditional statement in the loop body, condition C states that the invariant and the negation of the loop condition establish the output condition. The notation $F[a/x]$ denotes a version of formula F where every free occurrence of variable x is replaced by term a (after a suitable renaming of bound variables in F).

We are now, based on the definition of the datatype "array" given in the previous section, describing the formulation of these conditions in our system (the software distribution includes the example presented in this section in directory `examples` with the declarations listed in file `linsearch.pn` and the proofs stored in subdirectory `linsearch`).

First we declare the constants occurring in the verification conditions and the predicates *Input* and *Output* in which these constants freely occur:

```
a: ARR; olda: ARR; x: ELEM; oldx: ELEM;
i: NAT; n: NAT; r: INT;

Input: BOOLEAN =
  olda = a AND oldx = x AND n = length(a)
  AND i = 0 AND r = -1;

Output: BOOLEAN =
  a = olda AND
  ((r = -1 AND
     (FORALL(j:NAT): j < length(a) => get(a,j) /= x))
     OR
```

```
    (0 <= r AND r < length(a) AND get(a,r) = x AND
      (FORALL(j:NAT): j < r => get(a,j) /= x)));
```

Since the verification conditions use different instantiations of *Invariant*, this predicate does not directly refer to above constants but is provided with corresponding parameters instead:

```
Invariant: (ARR, ELEM, NAT, NAT, INT) -> BOOLEAN =
  LAMBDA(a: ARR, x: ELEM, i: NAT, n: NAT, r: INT):
    olda = a AND oldx = x AND n = length(a) AND
    i <= n AND
    (FORALL(j:NAT): j < i => get(a,j) /= x) AND
    (r = -1 OR (r = i AND i < n AND get(a,r) = x));
```

The four verification conditions can now be defined as follows:

```
A: FORMULA
  Input => Invariant(a, x, i, n, r);

B1: FORMULA
  Invariant(a, x, i, n, r) AND i < n AND r = -1
  AND get(a,i) = x => Invariant(a, x, i, n, i);

B2: FORMULA
  Invariant(a, x, i, n, r) AND i < n AND r = -1
  AND get(a,i) /= x => Invariant(a, x, i+1, n, r);

C: FORMULA
  Invariant(a, x, i, n, r) AND
  NOT(i < n AND r = -1) => Output;
```

The parameterized predicate *Invariant* is applied to those arguments that correspond to the instantiation values in the original definition, e.g. *Invariant(a, x, i+1, n, r)* represents $Invariant[a/a, x/x, i+1/i, n/n, r/r]$.

The pretty-printed versions of the declarations are shown below:

$$
\begin{array}{l}
\text{Input} \in \mathrm{B} = \text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge i = 0 \wedge r = -1 \\
\text{Output} \in \mathrm{B} = \\
\quad a = \text{olda} \\
\quad \wedge \\
\quad (\quad r = -1 \wedge (\forall j \in \mathbb{N}: j < \text{length}(a) \Rightarrow \text{get}(a,\ j) \neq x) \\
\quad \vee \\
\quad\quad 0 \leq r \wedge r < \text{length}(a) \wedge \text{get}(a,\ r) = x \\
\quad\quad \wedge \\
\quad\quad (\forall j \in \mathbb{N}: j < r \Rightarrow \text{get}(a,\ j) \neq x)) \\
\text{Invariant} \in (\text{ARR},\ \text{ELEM},\ \mathbb{N},\ \mathbb{N},\ \mathbb{Z}) \rightarrow \mathrm{B} = \\
\quad \lambda a \in \text{ARR},\ x \in \text{ELEM},\ i \in \mathbb{N},\ n \in \mathbb{N},\ r \in \mathbb{Z}: \\
\quad\quad \text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge i \leq n \\
\quad\quad \wedge \\
\quad\quad (\forall j \in \mathbb{N}: j < i \Rightarrow \text{get}(a,\ j) \neq x) \\
\quad\quad \wedge \\
\quad\quad (r = -1 \vee r = i \wedge i < n \wedge \text{get}(a,\ r) = x) \\
A \equiv \text{Input} \Rightarrow \text{Invariant}(a,\ x,\ i,\ n,\ r) \\
\mathrm{B1} \equiv \\
\quad \text{Invariant}(a,\ x,\ i,\ n,\ r) \wedge i < n \wedge r = -1 \wedge \text{get}(a,\ i) = x \\
\quad \Rightarrow \\
\quad \text{Invariant}(a,\ x,\ i,\ n,\ i) \\
\mathrm{B2} \equiv \\
\quad \text{Invariant}(a,\ x,\ i,\ n,\ r) \wedge i < n \wedge r = -1 \wedge \text{get}(a,\ i) \neq x \\
\quad \Rightarrow \\
\quad \text{Invariant}(a,\ x,\ i+1,\ n,\ r) \\
C \equiv \text{Invariant}(a,\ x,\ i,\ n,\ r) \wedge \neg\, (i < n \wedge r = -1) \Rightarrow \text{Output}
\end{array}
$$

We are now going to discuss the proof of each condition in turn, starting for better understanding with a display of the the overall structure of each proof.

Verification Condition *A* The proof of verification condition A is very simple:

```
[bca]: expand Input, Invariant
  [fuo]: scatter
    [bxg]: proved (CVCL)
```

The root state [bca] consists of a single goal formula:

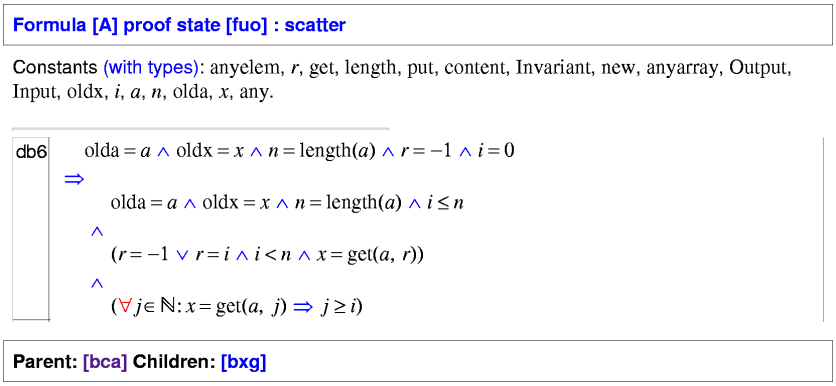
Formula [A] proof state [bca] : expand Input, Invariant

Constants (with types): anyelem, *r*, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, *i*, *a*, *n*, olda, *x*, any.

obk $\text{Input} \Rightarrow \text{Invariant}(a,\ x,\ i,\ n,\ r)$

Children: [fuo]

We execute `expand Input, Invariant` and get the proof state[fuo].

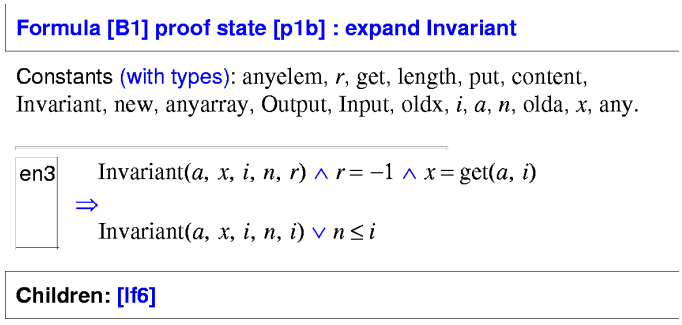


Rather than investigating this state, we simply press the "Scatter" button which generates a single state [bxg] which is automatically closed by the decision procedure.

Verification Condition *B1* The proof of verification condition *B1* is trivial:

```
[p1b]: expand Invariant
  [lf6]: proved (CVCL)
```

The root state [p1b] consists of a single goal [en3] with two occurrences of the predicate *Invariant*:



We execute `expand Invariant` which results in a single child state [lf6] that is closed automatically.

Verification Condition *B2* The proof of verification condition *B1* has the following structure:

[q1b]: expand Invariant in 6kv
  [slx]: scatter
    [a1y]: auto
      [cch]: proved (CVCL)
    [b1y]: proved (CVCL)
    [c1y]: proved (CVCL)
    [d1y]: proved (CVCL)
    [e1y]: proved (CVCL)

Like in the proof of condition *B1*, the root state [q1b] consists of a single goal [6kv] with two occurrences of the predicate *Invariant*:

Formula [B2] proof state [q1b] : expand Invariant in 6kv

Constants (with types): anyelem, *r*, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, *i*, *a*, *n*, olda, *x*, any.

6kv
$$\text{Invariant}(a,\ x,\ i,\ n,\ r) \wedge r = -1$$
$$\Rightarrow$$
$$x = \text{get}(a,\ i) \vee \text{Invariant}(a,\ x,\ 1+i,\ n,\ r) \vee n \leq i$$

Children: [slx]

We execute the command `expand Invariant in 6ev` which results in the following state:

Formula [B2] proof state [slx] : scatter

Constants (with types): anyelem, *r*, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, *i*, *a*, *n*, olda, *x*, any.

kvg
$$\text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge (\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i)$$
$$\wedge$$
$$r = -1$$
$$\Rightarrow$$
$$x = \text{get}(a,\ i) \vee n \leq i \vee n < i$$
$$\vee$$
$$r \neq -1 \wedge (x = \text{get}(a,\ r) \wedge r = i \Rightarrow n \leq i)$$
$$\vee$$
$$\text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge 1+i \leq n$$
$$\wedge$$
$$(r = -1 \vee x = \text{get}(a,\ r) \wedge r = i+1 \wedge 1+i < n)$$
$$\wedge$$
$$(\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq 1+i)$$

Parent: [q1b] Children: [a1y] [b1y] [c1y] [d1y] [e1y]

Rather than investigating this state further, we press the "Scatter" button  which generates five children states of which four are closed automatically. Only the state [a1y] requires our attention:

Formula [B2] proof state [a1y] : auto

Constants (with types): anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, Input, oldx, i, a, n, olda, any, x.

ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
hvv	$n = \text{length}(a)$
564	$\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i$
ptg	$x \neq \text{get}(a,\ i)$
l6h	$i < n$
uvs	$r = -1 \vee r = i \wedge x = \text{get}(a,\ r)$
k4w	$x = \text{get}(a,\ j_0)$
tlk	$1 + i \leq j_0$

Parent: [slx] Children: [cch]

This state contains a universally quantified assumption [564]. We guess that this assumption needs to be appropriately instantiated and press the "Auto" button  which generates a single child state [cch] that is automatically closed; thus the proof is complete.

Verification Condition *C* The proof of condition C has a slightly more complicated structure:

```
[dca]: expand Invariant, Output in zfg
  [tvy]: scatter
    [dcu]: auto
      [t4c]: proved (CVCL)
    [ecu]: split pkg
      [kel]: proved (CVCL)
      [lel]: scatter
        [lvn]: auto
          [lap]: proved (CVCL)
    [fcu]: auto
      [blt]: proved (CVCL)
    [gcu]: proved (CVCL)
```

The root state [dca] has goal [zfg] with occurrences of the predicates *Invariant* and *Output*.

Formula [C] proof state [dca] : expand Invariant, Output in zfg

Constants (with types): anyelem, r, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, i, a, n, olda, x, any.

zfg $\text{Invariant}(a,\ x,\ i,\ n,\ r) \Rightarrow \text{Output} \vee i < n \wedge r = -1$

Children: [tvy]

We use the command `expand Invariant, Output in zfg` to replace these predicates by their definitions, which results in the following state:

Formula [C] proof state [tvy] : scatter

Constants (with types): anyelem, r, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, i, a, n, olda, x, any.

aqc
$$\begin{array}{l} \text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge (\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i) \\ \Rightarrow \\ \quad n < i \vee r \neq -1 \wedge (x = \text{get}(a,\ r) \wedge r = i \Rightarrow n \leq i) \vee i < n \wedge r = -1 \\ \quad \vee \\ \quad\quad \text{olda} = a \\ \quad\quad \wedge \\ \quad\quad (\quad r = -1 \wedge (\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq \text{length}(a)) \\ \quad\quad\quad \vee \\ \quad\quad\quad\quad 0 \leq r \wedge x = \text{get}(a,\ r) \wedge r < \text{length}(a) \\ \quad\quad\quad\quad \wedge \\ \quad\quad\quad\quad (\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq r)) \end{array}$$

Parent: [dca] Children: [dcu] [ecu] [fcu] [gcu]

As usual, we do not bother to investigate the structure of this state further but immediately press the "Scatter" button  which generates four children states of which one is closed automatically. Of the three remaining states [dcu], [ecu], and [fcu], the first one is as follows:

Formula [C] proof state [dcu] : auto

Constants (with types): anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, Input, oldx, i, a, n, olda, any, x.

ed2 $\text{olda} = a$
cmz $\text{oldx} = x$
hvv $n = \text{length}(a)$
564 $\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i$
mys $i \leq n$
x2w $r = -1$
cpb $n \leq i$
k4w $x = \text{get}(a,\ j_0)$
6ha $j_0 < n$

f5e $x = \text{get}(a,\ -1)$

Parent: [tvy] Children: [t4c]

The state has an universally quantified assumption [564]; thus it looks like as if we need to use a proper instantiation of this formula. Before trying the "Auto" button we remember the other two open states and ponder that they may have similar structures. Rather than applying individually on each state, we press the button which applies the `auto` command not only to the current state but also to all its sibling states (see page 97). Our boldness is rewarded by the fact that both [dcu] and [fcu] are automatically closed such that we only need to investigate state [ecu] further:

Formula [C] proof state [ecu] : split pkg

Constants (with types): anyelem, r, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, i, a, n, olda, x, any.

ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
hvv	$n = \text{length}(a)$
564	$\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i$
mys	$i \leq n$
gkr	$r = -1 \vee r = i \wedge x = \text{get}(a,\ r) \wedge i < n$
orv	$r = -1 \Rightarrow n \leq i$
pkg	$r = -1 \Rightarrow (\exists j \in \mathbb{N}: x = \text{get}(a,\ j) \wedge j < \text{length}(a))$
jh5	$0 \leq r$

Parent: [tvy] Children: [kel] [lel]

This state has three assumptions [gkr], [orv] and [pkg] that start with the atomic formula $r = -1$ (which, as we remember, denotes "element not found" in the program). If $r = -1$, we can derive additional knowledge from the implications [orv] and [pkg] (where $r = -1$ appears in the hypothesis of the implication such that we can deduce its conclusion part); if $r \neq -1$, we may derive additional knowledge from the disjunction [gkr] (because then the remaining clauses of the disjunction must be true). Thus our further reasoning depends on the fact, which of the two possibilities $r = -1$ or $r \neq -1$ is true and we have to split our our proof correspondingly into two branches.

One way to proceed is thus to execute the command `case r=-1` which generates two child states, one with the additional assumption $r = -1$, one with the additional assumption $r \neq -1$ (see page 101). However, there also exists another possibility: moving the mouse cursor over the labels of [gkr], [orv], and [pkg] reveal popup menus that list applications of the command `split` to these formulas. This command "splits" the current state into several child states each of which receives as an additional assumption one of the components of the disjunctive formula to which the command is applied (see page 94, also an implication $F \Rightarrow G$ can be seen as a disjunction $\neg F \vee G$).

Being lazy, rather than typing in `case r=-1` on the command line, we select from the menu of [pkg] the command `split pkg`. This yields two child states of which one is automatically closed while the other with label [lel] still requires our attention[6]:

Formula [C] proof state [lel] : scatter

Constants (with types): anyelem, r, get, length, put, content, Invariant, new, anyarray, Output, Input, oldx, i, a, n, olda, x, any.

ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
hvv	$n = \text{length}(a)$
564	$\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i$
mys	$i \leq n$
gkr	$r = -1 \vee r = i \wedge x = \text{get}(a,\ r) \wedge i < n$
orv	$r = -1 \Rightarrow n \leq i$
1bb	$\exists j \in \mathbb{N}: x = \text{get}(a,\ j) \wedge j < \text{length}(a)$
jh5	$0 \leq r$

Parent: [ecu] **Children:** [lvn]

This state has an existential assumption [1bb]. To get rid of the quantifier, we press the "Scatter" button  and get the state [lvn]:

Formula [C] proof state [lvn] : auto

Constants (with types): anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, Input, oldx, i, a, n, olda, any, x.

ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
hvv	$n = \text{length}(a)$
564	$\forall j \in \mathbb{N}: x = \text{get}(a,\ j) \Rightarrow j \geq i$
mys	$i \leq n$
gkr	$r = -1 \vee r = i \wedge x = \text{get}(a,\ r) \wedge i < n$
orv	$r = -1 \Rightarrow n \leq i$
k4w	$x = \text{get}(a,\ j_0)$
6ha	$j_0 < n$
jh5	$0 \leq r$

Parent: [lel] **Children:** [lap]

This state contains a universally quantified assumption [564]; before investigating the state any further, we try whether the automatic instantiation of this formula

[6] The reader may try the alternative path which leads to a slightly different but essentially equivalent proof.

with the "Auto" button  does any good: indeed, a proof state [lap] is generated which is automatically closed such that the proof is completed.

Having proved all verification conditions, the partial correctness of the initially stated program is verified.

As shown in this proof, splitting proof states by case distinctions belongs to those activities (apart from expanding definitions, finding instantiations of quantified formulas, and introducing lemmas, see the next section) where human intervention is required. Since such splits may have drastic consequences on the size of the proof tree (and thus the complexity of the proof), the system does not try on its own to split a proof state by disjunctive assumptions (the "Scatter" button  splits proof states by conjunctive goals only).

3.4 Another Verification

Our final example uses another program verification in order to illustrate some more features of the system and subtle points of its use. The program to be verified represents the core of the *binary search* algorithm for finding an element x in an array a of integer numbers that is sorted in ascending order; the program sets a result value r to an index at which x occurs in a, respectively to -1, if x does not occur in a. The verification task is described by the following Hoare triple:

$$
\begin{array}{l}
\{olda = a \wedge oldx = x \wedge (\forall j : 0 \leq j < length(a) - 1 \Rightarrow a[j] \leq a[j+1]) \wedge \\
\quad r = -1 \wedge low = 0 \wedge high = length(a) - 1\} \\
\textbf{while } r = -1 \wedge low \leq high \textbf{ do} \\
\quad mid := \lfloor (low + high)/2 \rfloor \\
\quad \textbf{if } a[mid] = x \textbf{ then} \\
\qquad r := mid \\
\quad \textbf{else if } a[mid] < x \textbf{ then} \\
\qquad low := mid + 1 \\
\quad \textbf{else} \\
\qquad high := mid - 1 \\
\{a = olda \wedge x = oldx \wedge \\
\quad ((r = -1 \wedge (\forall i : 0 \leq j < length(a) \Rightarrow a[j] \neq x)) \vee \\
\quad\ (0 \leq r < length(a) \wedge a[r] = x))\}
\end{array}
$$

The main difference of to the verification presented in the previous example is the requirement of the array to be sorted and the computation of the index *mid* to be investigated next: this computation makes use of the division operator / (whose

result need not be an integer) and of the "floor" operator $\lfloor . \rfloor$ which maps a real number x to the largest integer number less than or equal x.

By the rules of the Hoare calculus, we derive the following five verification conditions A, B_1, B_2, B_3, and C where B_1, B_2, and B_3 describe the fact that the loop invariant is maintained by each of the three paths that may be taken through the body of the loop:

$$
\begin{array}{l}
\mathit{Input} :\Leftrightarrow \\
\quad \mathit{olda} = a \wedge \mathit{oldx} = x \wedge (\forall j : 0 \leq j < \mathit{length}(a) - 1 \Rightarrow a[j] \leq a[j+1]) \wedge \\
\quad r = -1 \wedge \mathit{low} = 0 \wedge \mathit{high} = \mathit{length}(a) - 1 \\
\mathit{Output} :\Leftrightarrow \\
\quad a = \mathit{olda} \wedge x = \mathit{oldx} \wedge \\
\qquad ((r = -1 \wedge (\forall j : 0 \leq j < \mathit{length}(a) \Rightarrow a[j] \neq x)) \vee \\
\qquad (0 \leq r < \mathit{length}(a) \wedge a[r] = x)) \\
\mathit{Invariant} :\Leftrightarrow \\
\quad \mathit{olda} = a \wedge \mathit{oldx} = x \wedge (\forall j : 0 \leq j < \mathit{length}(a) - 1 \Rightarrow a[j] \leq a[j+1]) \wedge \\
\quad -1 \leq r < \mathit{length}(a) \wedge (r \neq -1 \Rightarrow a[r] = x) \wedge \\
\quad 0 \leq \mathit{low} \leq \mathit{length}(a) \wedge -1 \leq \mathit{high} < \mathit{length}(a) \wedge \mathit{low} \leq \mathit{high} + 1 \wedge \\
\quad (\forall j : 0 \leq j < \mathit{length}(a) \wedge j < \mathit{low} \Rightarrow a[j] < x) \wedge \\
\quad (\forall j : 0 \leq j < \mathit{length}(a) \wedge \mathit{high} < j \Rightarrow x < a[j])
\end{array}
$$

$$
\begin{array}{l}
A :\Leftrightarrow \mathit{Input} \Rightarrow \mathit{Invariant} \\
B_1 :\Leftrightarrow \mathit{Invariant} \wedge r = -1 \wedge \mathit{low} \leq \mathit{high} \wedge a[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor] = x \Rightarrow \\
\quad \mathit{Invariant}[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor / r] \\
B_2 :\Leftrightarrow \mathit{Invariant} \wedge r = -1 \wedge \mathit{low} \leq \mathit{high} \wedge a[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor] \neq x \wedge \\
\quad a[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor] < x \Rightarrow \mathit{Invariant}[(\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor + 1)/\mathit{low}] \\
B_3 :\Leftrightarrow \mathit{Invariant} \wedge r = -1 \wedge \mathit{low} <= \mathit{high} \wedge a[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor] \neq x \wedge \\
\quad a[\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor] \not< x \Rightarrow \mathit{Invariant}[(\lfloor \frac{\mathit{low}+\mathit{high}}{2} \rfloor - 1)/\mathit{high}] \\
C :\Leftrightarrow \mathit{Invariant} \wedge \neg(r = -1 \wedge \mathit{low} \leq \mathit{high}) \Rightarrow \mathit{Output}
\end{array}
$$

A crucial part of the invariant says that "array a is sorted" by the formula $\forall j : 0 \leq j < \mathit{length}(a) - 1 \Rightarrow a[j] \leq a[j+1]$), a property that will become essential in the subsequent proof.

The software distribution contains in directory `examples` the text file `binary-search.pn` with the corresponding declarations in the specification language of our system and a subdirectory `binarysearch` with the corresponding proof. In the following, we will focus on verification condition *B3* whose proof is the most demanding one.

Since the specification language does not have a builtin "floor" function, we introduce a constant *floor* by the following declaration which is accompanied by the declaration of an axiom that characterizes this constant:

```
floor: REAL->INT;
floorAxiom: AXIOM
  FORALL(x:REAL): floor(x) <= x AND
    NOT(EXISTS(y:INT): y <= x AND floor(x) < y);
```

Based on the already previously described definition of datatype "array", (which we specialize in this verification to "array of integers"), we introduce constants for the program variables and mathematical constants occurring in the Hoare triple:

```
a: ARR; olda: ARR; x: INT; oldx: INT;
low: INT; high: INT; mid: INT; r: INT;
```

The verification conditions use various versions of the predicate *Invariant* which is therefore correspondingly parameterized:

```
Invariant: (ARR, INT, INT, INT, INT) -> BOOLEAN =
  LAMBDA(a: ARR, x: INT, low: INT, high: INT, r: INT):
    a = olda AND x = oldx AND
    (FORALL (j:NAT): j < length(a)-1 =>
       get(a, j) <= get(a, j+1)) AND
    -1 <= r AND r < length(a) AND
    (r /= -1 => get(a, r) = x) AND
    0 <= low AND low <= length(a) AND
    -1 <= high AND high < length(a) AND
    low <= high+1 AND
    (FORALL (j:NAT): j < low AND j < length(a) =>
      get(a,j) < x) AND
    (FORALL (j:NAT): high < j AND j < length(a) =>
      get(a,j) > x);
```

The pretty printed form of this invariant is shown below:

$$
\begin{aligned}
&\text{Invariant} \in (\text{ARR}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \to \text{B} = \\
&\quad \lambda a \in \text{ARR},\ x \in \mathbb{Z},\ \text{low} \in \mathbb{Z},\ \text{high} \in \mathbb{Z},\ r \in \mathbb{Z}: \\
&\qquad a = \text{olda} \wedge x = \text{oldx} \\
&\qquad \wedge \\
&\qquad (\forall j \in \mathbb{N}: j < \text{length}(a) - 1 \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1)) \\
&\qquad \wedge \\
&\qquad -1 \leq r \wedge r < \text{length}(a) \wedge (r \neq -1 \Rightarrow \text{get}(a,\ r) = x) \wedge 0 \leq \text{low} \\
&\qquad \wedge \\
&\qquad \text{low} \leq \text{length}(a) \wedge -1 \leq \text{high} \wedge \text{high} < \text{length}(a) \wedge \text{low} \leq \text{high} + 1 \\
&\qquad \wedge \\
&\qquad (\forall j \in \mathbb{N}: j < \text{low} \wedge j < \text{length}(a) \Rightarrow \text{get}(a,\ j) < x) \\
&\qquad \wedge \\
&\qquad (\forall j \in \mathbb{N}: \text{high} < j \wedge j < \text{length}(a) \Rightarrow \text{get}(a,\ j) > x)
\end{aligned}
$$

Proving a Lemma As we will see below, the proof of *B3* (and likewise the proof of *B2*) requires additional knowledge about sorted arrays. In particular, we need to infer that the fact that every pair of neighbor elements preserves the right order (the definition of "sorted") implies the fact that also every pair of non-neighbor elements preserves the order. We express this knowledge in the form of a declaration of a formula (lemma) L:

```
L: FORMULA
  (FORALL (j:NAT): j < length(a)-1 =>
    get(a, j) <= get(a, j+1)) =>
  (FORALL (j, k:NAT): j < length(a) AND
      k < length(a) AND k <= j =>
    get(a, k) <= get(a, j));
```

The pretty printed form of this lemma is shown below:

$$
\begin{aligned}
&L \equiv \\
&\qquad (\forall j \in \mathbb{N}: j < \text{length}(a) - 1 \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1)) \\
&\quad \Rightarrow \\
&\qquad (\forall j \in \mathbb{N},\ k \in \mathbb{N}: \\
&\qquad\quad j < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq j \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ j))
\end{aligned}
$$

The overall structure of the proof of this lemma is as follows:

```
[mca]: flatten
  [c3f]: induction j in z42
    [3da]: scatter
      [s3k]: proved (CVCL)
    [4da]: scatter
      [jdt]: auto
        [4kl]: proved (CVCL)
```

The root state [mca] of this proof is depicted below:

Formula [L] proof state [mca] : flatten

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \leq x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \geq y)$

odj
$$(\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1))$$
$$\Rightarrow$$
$$(\forall j \in \mathbb{N},\ k \in \mathbb{N}: j < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq j \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ j))$$

Children: [c3f]

The goal [odj] is an implication of two universally quantified formulas. We may thus be attempted to perform a predicate logic proof by pressing the "Scatter" button  , but the resulting proof state is a dead end, as is also the result of the more conservative 'Decompose" button  . A closer investigation of state [mca] gives us the key idea to perform an induction proof on the conclusion of the goal, but for this purpose we first need to break up the implication. Consequently, we invoke the command `flatten` described on page 107 which leads to the state [c3f] whose goal [z42] is a universally quantified formula with natural number variables j and k:

Formula [L] proof state [c3f] : induction j in z42

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \leq x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \geq y)$

iuh $\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1)$

z42 $\forall j \in \mathbb{N},\ k \in \mathbb{N}: j < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq j \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ j)$

Parent: [mca] Children: [3da] [4da]

We decide to perform the induction on the first variable of the goal and consequently execute `induction j in z42` resulting in two child states [3da] and [4da].

The proof state [3da] represents the induction base:

Formula [L] proof state [3da] : scatter

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \leq x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \geq y)$

iuh $\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1)$

6nq $\forall k \in \mathbb{N}: 0 < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq 0 \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ 0)$

Parent: [c3f] Children: [s3k]

Its goal [6nq] is still universally quantified; we thus apply the ‘Scatter” button which results in a proof state that is automatically closed.

The proof state [4da] represents the induction step:

Formula [L] proof state [4da] : scatter

Constants (with types): high, anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, floor, Input, low, oldx, a, mid, olda, any, x.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \leq x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \geq y)$

iuh $\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a,\ j) \leq \text{get}(a,\ j+1)$

cgs $\forall k \in \mathbb{N}: j_0 < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq j_0 \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ j_0)$

aw4 $\forall k \in \mathbb{N}:$
$j_0+1 < \text{length}(a) \wedge k < \text{length}(a) \wedge k \leq j_0+1 \Rightarrow \text{get}(a,\ k) \leq \text{get}(a,\ j_0+1)$

Parent: [c3f] Children: [jdt]

Its goal [aw4] is also still universally quantified; we thus apply the ‘Scatter” button resulting in the following state [jdt]:

Formula [L] proof state [jdt] : auto

Constants (with types): k_0, high, anyelem, r, get, length, put, content, Invariant, new, anyarray, j_0, floor, Output, Input, low, oldx, a, mid, olda, x, any.

6hu	$\forall x \in \mathbb{R}: \text{floor}(x) \le x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \ge y)$
iuh	$\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a, j) \le \text{get}(a, j+1)$
cgs	$\forall k \in \mathbb{N}: j_0 < \text{length}(a) \wedge k < \text{length}(a) \wedge k \le j_0 \Rightarrow \text{get}(a, k) \le \text{get}(a, j_0)$
vit	$1+j_0 < \text{length}(a)$
543	$k_0 \le 1+j_0$
ocx	$\text{get}(a, k_0) \le \text{get}(a, 1+j_0)$

Parent: [4da] Children: [4kl]

Closing this proof state apparently depends on the proper instantiation of the universally quantified assumptions [6hu], [iuh], and [cgs]. We thus try to find a suitable automatic instantiation by pressing the "Auto" button  ; indeed the resulting proof state is automatically closed by the decision procedure.

Proving the Verification Condition B3 We will now describe the proof of the following verification condition:

```
B3: FORMULA
  Invariant(a, x, low, high, r) AND r = -1 AND
  low <= high AND get(a, floor((low+high)/2)) /= x AND
  NOT get(a, floor((low+high)/2)) < x
  => Invariant(a, x, low, floor((low+high)/2)-1, r);
```

The pretty printed version of this condition is shown below:

$$
\begin{aligned}
\text{B3} \equiv\ & \\
& \text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1 \wedge \text{low} \le \text{high} \\
& \wedge \\
& \text{get}(a, \text{floor}(\tfrac{\text{low}+\text{high}}{2})) \ne x \\
& \wedge \\
& \neg\, \text{get}(a, \text{floor}(\tfrac{\text{low}+\text{high}}{2})) < x \\
& \Rightarrow \\
& \text{Invariant}(a, x, \text{low}, \text{floor}(\tfrac{\text{low}+\text{high}}{2})-1, r)
\end{aligned}
$$

The overall structure of the proof is as follows:

```
[r1b]: instantiate low/2+high/2 in 6hu
  [zrp]: expand Invariant
    [rui]: scatter
      [zhg]: auto
        [r5c]: proved (CVCL)
      [1hg]: proved (CVCL)
      [2hg]: lemma L
        [63o]: instantiate j_0, floor(low/2+high/2) in z42
          [25c]: proved (CVCL)
          [35c]: auto
            [g6q]: proved (CVCL)
      [3hg]: proved (CVCL)
      [4hg]: auto
        [n21]: proved (CVCL)
      [5hg]: proved (CVCL)
  [1rp]: proved (CVCL)
```

The root state [r1b] of the proof consists of an assumption [6hu] representing the axiomatization of the *floor* constant and the goal [yhd] representing the content of the verification condition *S3*:

Formula [B3] proof state [r1b] : instantiate low/2+high/2 in 6hu

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.

Axioms: $\text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1 \Rightarrow \text{high} < \text{low} \vee 0 \le \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})$.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \le x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \ge y)$

yhd

$$\begin{array}{l} \text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1 \\ \Rightarrow \\ \quad \text{high} < \text{low} \vee x = \text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})) \\ \quad \vee \\ \quad \text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})) < x \\ \quad \vee \\ \quad \text{Invariant}(a, x, \text{low}, -1 + \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}), r) \end{array}$$

Children: [zrp] [1rp]

Apparently, *floor* is only applied to the argument $\frac{high}{2} + \frac{low}{2}$ (three times). Therefore we instantiate already in the very beginning of the proof the assumption which represents the axiomatization of *floor* with this value by executing the command `instantiate high/2+low/2 in 6hu`. This yields the proof state [zrp]:

Formula [B3] proof state [zrp] : expand Invariant

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.

Axioms: $\text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1 \Rightarrow \text{high} < \text{low} \vee 0 \le \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})$.

6hu $\forall x \in \mathbb{R}: \text{floor}(x) \le x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \ge y)$

5cu $2 \cdot \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}) \le \text{high} + \text{low}$

$\wedge$

$(\forall y \in \mathbb{Z}: y > \frac{\text{low}}{2} + \frac{\text{high}}{2} \vee \text{floor}(\frac{\text{low}}{2} + \frac{\text{high}}{2}) \ge y)$

---

yhd $\text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1$

$\Rightarrow$

$\text{high} < \text{low} \vee x = \text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}))$

$\vee$

$\text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})) < x$

$\vee$

$\text{Invariant}(a, x, \text{low}, -1 + \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}), r)$

Parent: [r1b] Children: [rui]

This state has the specialized assumption [5cu]; the general assumption [6hu] will not play a role in the remainder of the proof any more. Now we execute `expand Invariant` to expose the goal formula. The resulting proof state [rui] (not depicted) looks quite daunting such that we immediately press the "Scatter" button  which results in six child states of which three are automatically closed.

The first of the remaining three open proof states is labelled [zhg]:

Formula [B3] proof state [zhg] : auto

Constants (with types): high, anyelem, r, get, length, put, content, Invariant, new, anyarray, floor, Output, Input, low, oldx, a, mid, olda, x, any.
Axioms: $\text{Invariant}(a, x, \text{low}, \text{high}, r) \wedge r = -1 \Rightarrow \text{high} < \text{low} \vee 0 \leq \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})$.

6hu	$\forall x \in \mathbb{R}: \text{floor}(x) \leq x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \geq y)$
unk	$\forall y \in \mathbb{Z}: y > \frac{\text{low}}{2} + \frac{\text{high}}{2} \vee \text{floor}(\frac{\text{low}}{2} + \frac{\text{high}}{2}) \geq y$
ygx	$\forall j \in \mathbb{N}: j < \text{low} \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) < x$
iuh	$\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a, j) \leq \text{get}(a, j+1)$
ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
a4p	$\forall j \in \mathbb{N}: \text{high} < j \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) > x$
x2w	$r = -1$
jtx	$\text{low} \leq \text{high}$
3kp	$x \neq \text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}))$
6he	$x \leq \text{get}(a, \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2}))$
42g	$0 \leq \text{low}$
3dc	$\text{high} < \text{length}(a)$

ntb	$0 \leq \text{floor}(\frac{\text{high}}{2} + \frac{\text{low}}{2})$

Parent: [rui] Children: [r5c]

We guess that this proof state can be closed by a clever instantiation of the universally quantified assumptions. Perhaps this is even the case for the two open sibling states, thus we press the button  which indeed closes this state and another one by automatic instantiation. The only remaining state is the one with label [2hg]:

Formula [B3] proof state [2hg] : lemma L

Constants (with types): high, anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, floor, Input, low, oldx, a, mid, olda, any, x.
Axioms: Invariant(a, x, low, high, r) $\wedge$ $r = -1$ $\Rightarrow$ high < low $\vee$ $0 \le \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2})$.

6hu	$\forall x \in \mathbb{R}: \text{floor}(x) \le x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \ge y)$
3er	$2 \cdot \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}) \le \text{high}+\text{low}$
unk	$\forall y \in \mathbb{Z}: y > \frac{\text{low}}{2}+\frac{\text{high}}{2} \vee \text{floor}(\frac{\text{low}}{2}+\frac{\text{high}}{2}) \ge y$
ygx	$\forall j \in \mathbb{N}: j < \text{low} \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) < x$
iuh	$\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a, j) \le \text{get}(a, j+1)$
ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
a4p	$\forall j \in \mathbb{N}: \text{high} < j \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) > x$
x2w	$r = -1$
jtx	$\text{low} \le \text{high}$
3kp	$x \ne \text{get}(a, \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}))$
6he	$x \le \text{get}(a, \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}))$
42g	$0 \le \text{low}$
3dc	$\text{high} < \text{length}(a)$
noj	$\text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}) < 1+j_0$
6k5	$j_0 < \text{length}(a)$
zki	$x < \text{get}(a, j_0)$

Parent: [rui] Children: [63o]

Apparently this state requires deeper investigation. Its goal [zki] is to prove $x < a[j_0]$ for some index j_0. So what do we know about x and j_0? From assumption [noj], we know $\lfloor \frac{low+high}{2} \rfloor \le j_0$; since a is sorted in ascending order, thus also $a[\lfloor \frac{low+high}{2} \rfloor] \le a[j_0]$ should hold (*). Furthermore, from assumptions [3kp] and [6he] we know $x < a[\lfloor \frac{low+high}{2} \rfloor]$; thus by transitivity of $\le$, the goal should hold.

But do we really know the formula marked (*) above? What we actually know is the property [iuh] which says that $a[j] \le a[j+1]$ for arbitrary pairs of neighbor indices $j, j+1$, while (*) talks about *non-neighbor* indices $\lfloor \frac{low+high}{2} \rfloor$ and j_0. This is the place where we need the knowledge that preserving the order for neighbor indices also implies preserving the order for non-neighboring indices, i.e. we need the formula L that was previously proved. We thus execute the command `lemma L` described on page 106 that imports this formula as an assumption and yields the proof state [63o]:

Formula [B3] proof state [63o] : instantiate j_0, floor(low/2+high/2) in z42

Constants (with types): high, anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, floor, Input, low, oldx, a, mid, olda, any, x.
Axioms: Invariant(a, x, low, high, r) $\wedge$ $r = -1$ $\Rightarrow$ high < low $\vee$ $0 \le \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2})$.

6hu	$\forall x \in \mathbb{R}$: floor(x) $\le x \wedge (\forall y \in \mathbb{Z}$: $y > x \vee$ floor(x) $\ge y$)
3er	$2 \cdot \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}) \le \text{high}+\text{low}$
unk	$\forall y \in \mathbb{Z}$: $y > \frac{\text{low}}{2}+\frac{\text{high}}{2} \vee \text{floor}(\frac{\text{low}}{2}+\frac{\text{high}}{2}) \ge y$
ygx	$\forall j \in \mathbb{N}$: $j <$ low $\wedge$ $j <$ length(a) $\Rightarrow$ get(a, j) $< x$
iuh	$\forall j \in \mathbb{N}$: $j+1 <$ length(a) $\Rightarrow$ get(a, j) $\le$ get(a, $j+1$)
ed2	olda $= a$
cmz	oldx $= x$
a4p	$\forall j \in \mathbb{N}$: high $< j \wedge j <$ length(a) $\Rightarrow$ get(a, j) $> x$
x2w	$r = -1$
jtx	low $\le$ high
3kp	$x \ne$ get(a, floor($\frac{\text{high}}{2}+\frac{\text{low}}{2}$))
6he	$x \le$ get(a, floor($\frac{\text{high}}{2}+\frac{\text{low}}{2}$))
42g	$0 \le$ low
3dc	high < length(a)
noj	floor($\frac{\text{high}}{2}+\frac{\text{low}}{2}$) $< 1+j_0$
6k5	$j_0 <$ length(a)
z42	$\forall j \in \mathbb{N}$, $k \in \mathbb{N}$: $j <$ length(a) $\wedge$ $k <$ length(a) $\wedge$ $k \le j \Rightarrow$ get(a, k) $\le$ get(a, j)
zki	$x <$ get(a, j_0)

Parent: [2hg] Children: [25c] [35c]

This state contains an additional assumption [z42] which is the conclusion of the implication contained in lemma L; its hypothesis is identical to assumption [iuh] and is therefore automatically discharged by the system. Pressing the "Auto" button  does not close the state thus we execute the command `instantiate j_0, floor(low/2+high/2) in z42` in order to explicitly instantiate the variables j and k in the new assumption by j_0 and $\lfloor \frac{low+high}{2} \rfloor$, respectively. This leads to the following state [35c]:

Formula [B3] proof state [35c] : auto

Constants (with types): high, anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, floor, Input, low, oldx, a, mid, olda, any, x.
Axioms: Invariant(a, x, low, high, r) $\wedge$ $r = -1$ $\Rightarrow$ high < low $\vee$ $0 \le \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2})$.

6hu	$\forall x \in \mathbb{R}: \text{floor}(x) \le x \wedge (\forall y \in \mathbb{Z}: y > x \vee \text{floor}(x) \ge y)$
unk	$\forall y \in \mathbb{Z}: y > \frac{\text{low}}{2}+\frac{\text{high}}{2} \vee \text{floor}(\frac{\text{low}}{2}+\frac{\text{high}}{2}) \ge y$
ygx	$\forall j \in \mathbb{N}: j < \text{low} \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) < x$
iuh	$\forall j \in \mathbb{N}: j+1 < \text{length}(a) \Rightarrow \text{get}(a, j) \le \text{get}(a, j+1)$
ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
a4p	$\forall j \in \mathbb{N}: \text{high} < j \wedge j < \text{length}(a) \Rightarrow \text{get}(a, j) > x$
x2w	$r = -1$
jtx	$\text{low} \le \text{high}$
3kp	$x \ne \text{get}(a, \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}))$
6he	$x \le \text{get}(a, \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2}))$
42g	$0 \le \text{low}$
3dc	$\text{high} < \text{length}(a)$
6k5	$j_0 < \text{length}(a)$
z42	$\forall j \in \mathbb{N}, k \in \mathbb{N}: j < \text{length}(a) \wedge k < \text{length}(a) \wedge k \le j \Rightarrow \text{get}(a, k) \le \text{get}(a, j)$
ntb	$0 \le \text{floor}(\frac{\text{high}}{2}+\frac{\text{low}}{2})$

Parent: [63o] Children: [g6q]

We again press  which generates a child state [g6q] which is automatically closed; this completes the proof.

Summary Above proofs demonstrate two things: most notably, they show that even for rather simple verifications "critical" proof states arise that require deeper analysis and perhaps the use of additional knowledge that has to be "imported" into the proof. Either this knowledge is already available in the form of previously declared formulas or the current proof has to be suspended to establish the knowledge and then resume the proof (the command `assume` described on page 100 allows to introduce such knowledge and prove it *within* the current proof). If an imported formula is not (yet) verified by a proof, the current proof is marked as "relative" with respect to unverified formulas.

The success of another important feature is demonstrated by the fact that we did *not* notice it. In proof state [63o] we instantiated a *natural number* variable j by an *integer* value $\lfloor \frac{low+high}{2} \rfloor$. This was legal because the system created an additional proof state [25c] whose goal was to prove that this value is not negative; the system automatically then automatically discharged this goal by applying its

decision procedure. The theoretical basis for this is a system of subtypes (see page 66) which treats natural numbers as a subdomain of integers and integers as a subdomain of reals. This greatly simplifies proofs that have to deal with various arithmetic domains compared to systems that have a less flexible type discipline.

Having studied the example specifications and proofs discussed in this and the previous sections, the user now should have a feeling for the "flavor" of interaction with the RISC ProofNavigator. The system is helpful in quickly decomposing a proof into the "interesting" (i.e. difficult) proof states; it also takes from the user the task of dealing with various kinds of low-level reasoning steps. On the other hand, the system leaves the user with the task of finding the "crucial" steps in a proof like finding non-trivial variable instantiations, imports of lemmas, etc. The system is definitely and deliberately not an automatic theorem prover rather than a proof assistant that attempts to be as helpful as possible but not clever beyond its abilities. Chapter 4 discusses known deficiencies of the system and potential for future improvements.

Chapter 4

Future Work

The RISC ProofNavigator has shown its usefulness in several small and not-so-small system verifications (some of which are contained in the software distribution). However, there are several issues that require further work:

Decision Procedures The system is currently bound to the decision procedure CVCL Version 2.0 and does neither work with more recent versions of CVCL nor with other decision procedures. This is not a fundamental problem but a question of a mapping of the higher-order specification language used by the RISC ProofNavigator to the first-order languages supported by most provers and of corresponding software interfaces. A diploma thesis has recently started corresponding investigations.

Specification Language The specification language of the system is very simple and does not support various features than can be found in other languages. For instance, it lacks the definition of recursive algebraic datatypes with structural induction as a proof principle and also a module system analogous to that of PVS. The introduction of corresponding features will be triggered by the integration of the system into a larger program reasoning framework.

Rewriting Proofs The system is weak when it comes to proofs that essentially depend on applying universally quantified equalities (or equivalences or implications) as rewrite rules. For instance, the software distribution contains an example specification `arrays.pn` with an axiomatic characterization of arrays; based on this specification, the proof of the extensionality principle corresponding to the one presented in Section 3.2 becomes quite cumbersome. The system should be extended to allow to mark formulas of a certain form as "rewrite rules" that are automatically applied in proof state simplifications (such features can be e.g. found in PVS and Isabelle).

Proof Checking The system stores the individual commands invoked in a proof for later replay; it does not generate *proof objects* that describe the individual low-level reasoning steps performed by the commands and that can be used for independent verification of a proof by an external proof checker. Some decision procedures (e.g. CVCL) optionally generate such objects as well as several interactive theorem provers (e.g. CoQ or Isabelle). It would be worthwhile to investigate similar features for the RISC ProofNavigator.

While some of above issues will be most probably tackled in the future, we will mainly concentrate on the development of an program exploration environment that integrates the RISC ProofNavigator as a reasoning component. The demands of this environment will drive the further elaboration of the software.

References

[1] Clark Barrett. CVC Lite Homepage, April 2006. New York University, NY, http://www.cs.nyu.edu/ acsys/cvcl.

[2] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, Berlin, 2004.

[4] Bruno Buchberger, Tudor Jebelean, et al. A Survey of the Theorema project. In Wolfgang Küchlin, editor, *ISSAC'97 International Symposium on Symbolic and Algebraic Computation*, pages 384–391, Maui, Hawaii, July 21–23, 1997. ACM Press, New York.

[5] S. Buswell and others (eds). The OpenMath Standard. Version 2.0, The OpenMath Society, June 2004. http://www.openmath.org.

[6] David Carlisle and others (eds). Mathematical Markup Language (MathML) Version 2.0 (Second Edition). W3C Recommendation, World Wide Web Consortium, October 2003. http://www.w3.org/TR/MathML2.

[7] The Coq Proof Assistant, 2006. The LogiCal Project, INRIA, France, http://coq.inria.fr.

[8] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] Isabelle, 2006. University of Cambridge, UK and Technical University Munich, Germany, http://www.cl.cam.ac.uk/Research/HVG/Isabelle.

[10] Michael Kohlhase. OMDoc — An Open Markup Format for Mathematical Documents (Version 1.2). Technical Specification and Primer, Saarland University, Germany, April 2006. http://www.mathweb.org/omdoc.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, 2005.

[12] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 14–18, 1992. Springer.

[13] Terence Parr et al. ANTLR Reference Manual. Version 2.7.5, University of San Francisco, CA, January 2005. http://www.antlr.org/doc.

[14] PVS Specification and Verification System, 2006. Computer Science Laboratory, Software Research Institute (SRI) International, Menlo Park, CA, http://pvs.csl.sri.com.

[15] Wolfgang Schreiner. Program Verification with the RISC ProofNavigator. In *Teaching Formal Methods: Practice and Experience, BCS-FACS Christmas Meeting*. Electronic Workshops in Computing (eWiC), British Computer Society, 2006.

[16] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 2007. To appear.

[17] SMT-LIB — The Satisfiability Modulo Theories Library, 2006. University of Iowa, Iowa City, IA, http://combination.cs.uiowa.edu/smtlib.

[18] Overview of Theorema, 2006. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at/research/theorema.

[19] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, Berlin, 2006.

Appendix A

Specification Language

In this appendix, we describe the language for constructing *specifications*. The main syntactic elements of specifications are *type expressions*, *value expressions*, and *declaration expressions* denoting at the semantic level *types*, *values*, and *declarations*. We will frequently not distinguish between expressions and their denotations, and call both "types", "values", and "declarations", respectively.

In the following, the variables *T*, *T1*, *T2*, ..., *Tn* denote types, the variables *E*, *E1*, *E2*, ..., *En* denote values, the variables *D*, *D1*, *D2*, ..., *Dn* denote declarations, and the variables *I*, *I1*, *I2*, ..., *In* denote identifiers.

As for the *lexical grammar* (the rules for forming words) of the language, a specification consists of a sequence of tokens separated by white-space (which includes indentations and line breaks that have thus no significance). If a line contains the comment token %, the rest of the line (including the token) is ignored.

As for the *syntactical grammar* (the rules for forming sentences) of the language, a specification consists of a sequence of declarations each of which is terminated by a semicolon:

```
D1;
D2;
...
Dn;
```

The lexical grammar and the syntactical grammar of the language are defined in Appendix F.

As for the semantics of a specification, the following sections describe the language's types, values, and declarations in an informal style. The primary purpose of these descriptions is easy readability, not ultimate preciseness; a formal definition of the language semantics remains subject of another paper.

A.1 Types

In this section, we describe the language's *type expressions* denoting *types* i.e. domains of values. The type system is *higher-order* i.e. it includes functions and does not make a fundamental difference between functions and predicates (which are just functions whose result is a truth value). Types are partially ordered by a *subtype* relationship such that if *T1* is a subtype of *T2*, any value of type *T1* may occur where a value of type *T2* is expected. The specification language is *strongly typed*; the system ensures type discipline for every declaration by a combination of static checking and dynamic proving (see Section A.3).

A.1.1 Atomic Types

Synopsis

```
BOOLEAN
NAT
INT
REAL
```

Description Apart from atomic types introduced by the user via type declarations (see page 77), there are the following builtin atomic types:

`BOOLEAN` This type denotes the domain of truth values with literal constants `TRUE` and `FALSE`.

`NAT` This type denotes the domain of natural numbers with literal constants `0`, `1`, `2`, ... and the addition function `+` and the multiplication function `*` on natural numbers (see page 70).

`INT` This type denotes the domain of integer numbers whose values can be constructed from `NAT` values by application of the negation function `-` (see page 70).

`REAL` This type denotes the domain of real numbers of which some values can be constructed from `INT` values by application of the division function `/` (see page 70).

`NAT` is a subtype of `INT` which is in turn a subtype of `REAL`. Formulas are constructed from values of type `BOOLEAN` (see page 78).

Pragmatics The type `NAT` is essentially equivalent to the subtype (see page 67)

```
SUBTYPE(LAMBDA(x:INT): x >= 0)
```

A.1.2 Range Types

Synopsis

```
[E1..E2]
```

Description This type takes two expressions *E1* and *E2* denoting integer values such that the value of *E1* is less than or equal the value of *E2*; the type denotes the domain of all integers greater than or equal *E1* and less than or equal *E2*.

The type gives rise to a type checking condition $E_1 \leq E_2$.

Pragmatics This type is essentially equivalent to the subtype (see page 67)

```
SUBTYPE(LAMBDA(x:INT): E1 <= x AND x <= E2)
```

A.1.3 Subtypes

Synopsis

```
SUBTYPE(E)
```

Description This type takes a value E of type $T \rightarrow$ `BOOLEAN` for some type T, i.e. a unary predicate on T. The type denotes the domain of all values of T that satisfy this predicate.

The denoted predicate must not be false for all values of T; thus the type gives rise to a type checking condition $\exists x \in T : E_x$.

Pragmatics The value E is typically denoted by a function expression as described on page 73.

A.1.4 Function Types

Synopsis

```
TA -> TR
(T1, T2, ..., Tn) -> TR
```

Description The type `TA -> TR` denotes the domain of unary functions with argument type *TA* and result type *TR*; every such function f takes an argument a from *TA* and returns a result $f(a)$ from *TR*.

The type `(T1, T2, ..., Tn) -> TR` denotes the domain of n-ary functions with argument types *T1*, *T2*, ..., *Tn* and result type *TR*; every such function f takes n arguments a_1 from *T1*, a_2 from *T2*, ..., a_n from *Tn* and returns a result $f(a_1, a_2, \ldots, a_n)$ from *TR*.

Pragmatics Because of a deficiency of the underlying decision procedure, the system does currently not understand that, if two functions are equal, also their function values are equal, i.e. it does not implement the following derivation rule:

$$\frac{\begin{array}{l}[E]\ T_1 : \text{type} \\ [E]\ T_2 : \text{type} \\ [E] \vdash I_1 : T_1 \to T_2 \\ [E] \vdash I_2 : T_1 \to T_2 \\ [E] \vdash V : T_1 \\ [E] \ldots \vdash I_1 = I_2\end{array}}{[E] \ldots \vdash I_1(V) = I_2(V)}$$

To overcome this problem, one may instead use an array type (which implements the corresponding rule for array access, see page 69) or explicitly introduce corresponding axioms (see page 78) on demand.

The type `(T1, T2, ..., Tn) -> TR` is different from the type `[T1, T2, ..., Tn] -> TR`. While the former denotes a domain of n-ary functions, the later denotes a domain of unary functions whose argument type is a tuple type (see page 69); each such a function f takes an n-ary tuple $t = (a_1, a_2, \ldots, a_n)$ and returns a result $f(t) = f((a_1, a_2, \ldots, a_n))$.

Function values can be constructed as described on page 73.

A.1.5 Array Types

Synopsis

```
ARRAY T1 OF T2
```

Description This type denotes the domain of arrays with indices of type *T1* and values of type *T2* (see page 74 for the operations supported on arrays).

Pragmatics From the semantic point of view, this type is essentially equivalent to a unary function type (see page 68). The major difference is that the system understands that, if two arrays are equal, access to these arrays at equal indices yields equal results (the system does not understand the corresponding rule for function applications).

A.1.6 Tuple Types

Synopsis

```
[T1, T2, ..., Tn]
```

Description This type denotes the domain of n-ary tuples $(a_1, a_2, \ldots, a_n)$ with n values a_1 from *T1*, a_2 from *T2*, ..., a_n from *Tn* (see page 75 for the operations supported on tuples).

The tuple type `[T]` is identified with `T`.

A.1.7 Record Types

Synopsis

```
[# I1:T1, I2:T2, ..., In:Tn #]
```

Description This type denotes the domain of n-ary records (# $I_1 := a_1, I_2 := a_2, \ldots, I_n := a_n$ #) with n values a_1 from *T1*, a_2 from *T2*, ..., a_n from *Tn* assigned to record field identifiers I_1, I_2, ..., I_n (see page 75 for the operations supported on records).

A.2 Values

In this section, we describe the language's *value expressions* denoting *values*, i.e., our objects of discourse. Each value expression has a specific type such that any value it may denote must be an element of the domain denoted by the type. Value expressions of type `BOOLEAN` are sometimes called *formulas*; all other value expressions are also called *terms*. However, since the specification language is higher-order, there is no fundamental difference between formulas and terms.

A.2.1 Arithmetic Terms

Synopsis

```
Digits
E1+E2
E1-E2
-E1
E1*E2
E1/E2
E1^E2
```

Descriptions These expressions represent number literals (composed of the decimal digits 0...9) and the arithmetic operations addition, subtraction, negation, division, and exponentiation. Apart from exponentiation, the types of the values *E1* and *E2* may be arbitrary (potentially different) arithmetic types (`NAT`, `INT`, `REAL`, range types, and subtypes of arithmetic types). In the case of exponentiation, the type of *E1* may be an arbitrary arithmetic type but the type of *E2* must be an integral type (`NAT`, `INT`, a range type, or a subtype of an integral type).

For determining the result type of each operation, the usual subtype ordering is considered, i.e., `NAT` and range types are subtypes of `INT`, and `INT` is a subtype of `REAL`. The result type of each function is then the smallest result type that can be deduced from the argument types and the the operation without actually considering the argument values, e.g. adding two `NAT` values yields a `NAT` value but adding a `NAT` value and an `INT` value yields an `INT` value. Dividing two values always yields a `REAL` value. Exponentiation with an exponent *E2* of (a subtype of) type `NAT` yields a result with the type of the base value *E1*; otherwise the type of the result is `REAL`.

Pragmatics For convenience, also the term `+`*E* is accepted as a synonym of *E*.

A.2.2 Atomic Formulas

Synopsis

```
E1 = E2
E1 /= E2
E1 < E2
E1 <= E2
E1 > E2
E1 >= E2
```

Description These expressions represent the comparison operations equality, inequality, less-than, less-than-or-equal, greater-than, and greater-than-or-equal. The expressions are composed from two values *E1* and *E2*. In the case of equality and inequality, the types of both values may be arbitrary but must be equal. In the case of the other operations, the types of the values may be (potentially different) numeric types (`NAT`, `INT`, `REAL`, a range type, or a subtype of a numeric type). The result type of the expression is `BOOLEAN`.

A.2.3 Propositional Formulas

Synopsis

```
TRUE
FALSE
NOT E
E1 AND E2
E1 OR E2
E1 => E2
E1 <=> E2
E1 XOR E2
```

Description These expression represent the logical constants “true” and “false” and the logical operations negation, conjunction, disjunction, implication, equivalence, and non-equivalence. They are composed from values *E*, *E1*, and *E2* of type `BOOLEAN` and yield a value of type `BOOLEAN`.

A.2.4 Quantified Formulas

Synopsis

```
FORALL(I1:T1, I2:T2, ..., In:Tn): E
EXISTS(I1:T1, I2:T2, ..., In:Tn): E
```

Description These expressions represent universal and existential quantification, respectively. They are composed of identifiers *I1* of type *T1*, *I2* of type *T2*, ..., *In* of type *Tn* and a value *E* of type `BOOLEAN` (in which `I1`, `I2`, ..., `In` may freely occur) and have themselves type `BOOLEAN`.

Pragmatics If quantified expressions occur in the context of another expression, they must be put into parentheses, as e.g. the `EXISTS` formula in

```
FORALL(x:NAT): x > 0 => (EXISTS(y:NAT): y+1=x)
```

For multiple parameters with the same type, the parameter type needs to be declared only once, thus it is for example legal to write

```
FORALL(x,y:NAT, z:INT): x+y >= 0*z
```

A.2.5 Conditional Expressions

Synopsis

```
IF E THEN E1 ELSE E2 ENDIF
IF E THEN E1
   ELSIF E' THEN E1' ... ELSE E2 ENDIF
```

Description The expression `IF` *E* `THEN` *E1* `ELSE` *E2* `ENDIF` consists of a value *E* of type `BOOLEAN` and of two values *E1* and *E2* of same type which is also the type of the conditional expression. The value of the expression is *E1*, if *E* is `TRUE`, and *E2*, otherwise.

The more general form

```
IF E THEN E1
ELSIF E' THEN E1'
ELSIF E'' THEN E1''
...
ELSE E2
ENDIF
```

is a syntactic short cut for

```
IF E THEN E1
ElSE IF E' THEN E1'
ELSE IF E'' THEN E1''
...
ELSE E2
ENDIF ... ENDIF ENDIF
```

A.2.6 Let Expressions

Synopsis

```
LET I1:T1 = E1,... IN E
LET I1 = E1,... IN E
```

Description This expression consists of a sequence of value definitions which set up an environment in which an expression E is evaluated. The type and the value of the let expression is that of E.

Each definition consists of an identifier I_i which is bound to the value of an expression E_i of type T_i (the type is optional in the definition); the identifier I_i is visible in the expressions $E_{i+1}, E_{i+2}, \ldots$ used in the subsequent definitions (but not in the expression E_i defining I_i itself).

Matching the types of *E1* to *T1*, ... may give rise to a type checking condition.

A.2.7 Function Values and Applications

Synopsis

```
LAMBDA(I1:T1, I2:T2, ..., In:Tn): E
E(E1, ..., En)
```

Description The *function value* expression

```
LAMBDA(I1:T1, I2:T2, ..., In:Tn): E
```

denotes the function that, given arguments *I1* of type *T1*, *I2* of type *T2*, ..., *In* of type *Tn*, returns the value of expression *E* (in which *I1*, *I2*, ..., *In* may freely occur). Correspondingly the type of the expression is the function type `(T1, T2, ..., Tn) -> T` where *T* is the type of *E* (see page 68).

The *function application* expression `E(E1, ..., En)` consists of an expression *E* denoting a value of type `(T1, T2, ..., Tn) -> T` for some types *T1*, *T2*, ..., *Tn*, *T* and of expressions *E1* of type *T1*, *E2* of type *T2*, ..., *En* of type *Tn*. The value of the application is the result of *E* when applied to the values of *E1*, *E2*, ..., *En*; the type of the application is *T*.

Pragmatics In a function value expression, for multiple parameters with the same type, the parameter type needs to be declared only once, thus it is for example legal to write

```
LAMBDA(x,y:NAT, z:REAL): x*y+z
```

A.2.8 Array Values, Updates, and Selections

Synopsis

```
ARRAY(I:T): E
E WITH [E1]:=E2
E[E1]
```

Description The *array value* expression `ARRAY(I:T): E` denotes the array where every index *I* of type *T* is mapped to the element denoted by the expression *E* (in which *I* may freely occur). Correspondingly the type of the expression is the array type `ARRAY T OF TE` where *TE* is the type of *E* (see page 69).

The *array update* expression `E WITH [E1]:=E2` is composed of a value *E* of type `ARRAY T1 OF T2` for some types *T1* and *T2*, a value *E1* of type *T1* and a value *E2* of type *T2*. It denotes the array that is identical to *E* except that index *E1* is mapped to element *E2*. Correspondingly the type of the expression is also `ARRAY T1 OF T2`.

The *array selection* expression `E[E1]` is composed of a value *E* of type `ARRAY T1 OF T2`, for some types *T1* and *T2*, and a value *E1* of type *T1*. It denotes the element of *E* at index *E1*; correspondingly its type is `T2`.

A.2.9 Tuple Values, Updates, and Selections

Synopsis

```
(E1, E2, ..., En)
E WITH [0]:=E1
E WITH [1]:=E2
...
E WITH [n-1]:=En
E.0
E.1
...
E.n-1
```

Description The *tuple value* expression `(E1, E2, ..., En)` denotes the *n*-ary tuple composed of the values *E1*, *E2*, ..., *En*. Correspondingly the type of the expression is the tuple type `(T1, T2, ..., Tn)` where *T1* is the type of *E1*, *T2* is the type of *E2*, ..., *Tn* is the type of *En* (see page 69). The tuple `(E)` is identified with *E*.

The *n tuple update* expressions `E WITH [0]:=E1`, `E WITH [1]:=E2`, ..., `E WITH [n-1]:=En` each take a value *E* of tuple type `(T1,T2,...,Tn)` for some types *T1*, *T2*, ..., *Tn* and one of the values *E1* of type *T1*, *E2* of type *T2*, ..., *En* of type *Tn*. They denote the tuple that is identical to *E* except that component 0, 1, ..., $n-1$ is updated by value *E1*, *E2*, ..., *En*. Correspondingly the type of each expression is also `(T1,T2,...,Tn)`.

The *n tuple selection* expressions `E.0`, `E.1`, ..., `E.n-1` each take a value *E* of tuple type `(T1,T2,...,Tn)` for some types *T1*, *T2*, ..., *Tn*. The expressions denote the first, second, ..., *n*-th component of this tuple; their types are *T1*, *T2*, ..., *Tn*, correspondingly.

A.2.10 Record Values, Updates, and Selections

Synopsis

```
(# I1:=E1, I2:=E2, ..., In:=En #)
E WITH [I1]:=E1
E WITH [I2]:=E2
...
E WITH [In]:=En
```

```
E.I1
E.I2
...
E.In
```

Description The *record value* expression

```
(# I1:=E1, I2:=E2, ..., In:=En #)
```

denotes the *n*-ary record composed of the values of *E1*, *E2*, ..., *En* by assigning these values to the record field identifiers *I1*, *I2*, ..., *In*. Correspondingly the type of the expression is the record type `(# I1:T1, I2:T2, ..., In:Tn #)` where *T1* is the type of *E1*, *T2* is the type of *E2*, ..., *Tn* is the type of *En* (see page 69).

The *n record update* expressions `E WITH [I1]:=E1`, `E WITH [I2]:=E2`, ..., `E WITH [In]:=En` each take a value expression *E* of record type `(# I1:T1, I2:T2, ..., In:Tn #)` for some identifiers *I1*, *I2*, ..., *In* and types *T1*, *T2*, ..., *Tn* and one of the values *E1* of type *T1*, *E2* of type *T2*, ..., *En* of type *Tn*. They denote the record that is identical to *E* except that record field *I1*, *I2*, ..., *In* is updated by value *E1*, *E2*, ..., *En*. Correspondingly the type of each expression is also `(# I1:T1, I2:T2, ..., In:Tn #)`.

The *n record selection* expressions `E.I1`, `E.I2`, ..., `E.In` take a value *E* of record type `(# I1:T1, I2:T2, ..., In:Tn #)` for some identifiers *I1*, *I2*, ..., *In* and types *T1*, *T2*, ..., *Tn*. The expressions denote the first, second, ..., *n*-th component of this record; their types are *T1*, *T2*, ..., *Tn*, correspondingly.

A.3 Declarations

In this section, we describe the language's *declaration expressions* denoting *declarations* each of which extends the environment by a mapping of an identifier to a type, value, or formula. When type checking a declaration, a *type checking condition* may be generated which is forwarded to a decision procedure for automatic proving. If the condition cannot be proved, a warning is issued, but the type checking condition is nevertheless assumed true. Thus the user must establish by a separate proof that the type checking condition is valid (any further proof might be unsound, otherwise).

A.3.1 Type Declarations

Synopsis

```
I: TYPE
I: TYPE = T
```

Description This declaration introduces a type constant *I*. The declaration form `I: TYPE` associates to *I* a non-empty atomic type which is different from any other atomic type declared so far. The declaration form `I: TYPE = T` equates *I* with *T*.

If the type expression *T* contains a range type (see page 67) or subtype (see page 67), the type declaration gives rise to a type checking condition which states that the declared type is not empty.

A.3.2 Value Declarations

Synopsis

```
I: T
I: T = E
```

Description This declaration introduces a value constant denoted by the identifier *I*; the type of the constant is *T*. The command form `I: T` does not define the value of *I*. The command form `I: T = E` defines the value of *I* by the value *E* whose type must match *T*.

As well determining the type of *E* as matching the type of *E* to *T* may give rise to a type checking condition.

A value declaration may also appear in proving mode (see page 100).

Pragmatics The declaration may also introduce function or predicate constants which are just value constants with function type (see page 68). Matching the value type `U1 -> U2` to the declaration type `T1 -> T2` only succeeds if both `T1` equals `U1` and `T2` equals `U2`. This is stricter than theoretically necessary (it would suffice to check that `T1` is a subtype of `U1` and `U2` is a subtype of `T2`) because the decision procedure applied by the system may not accept a looser type discipline.

A.3.3 Formula Declarations

Synopsis

```
I: FORMULA E
I: AXIOM E
```

Description This declaration introduces a formula constant *I* with value *E* of type `BOOLEAN`; the formula denoted by *I* is consequently assumed to be true.

If the declaration is tagged with the keyword `FORMULA`, the formula needs proof; it can be also used as an additional assumption in the proof of another formula by use of the `lemma` command described on page 106.

If the declaration is tagged with the keyword `AXIOM`, the formula does not need proof; it is automatically imported as an additional assumption into the proof of every subsequent formula.

Type checking *E* may give rise to a type checking condition.

Appendix B

System Commands

In this appendix, we describe all commands that the user can apply to control the system. Some commands are only applicable in declaration mode (when the system is processing declarations as described in Appendix A), others are only applicable in proving mode (when the system is constructing the proof of a formula under the guidance of the user); some commands are applicable in both modes.

Every command has a textual representation that can be typed in on the command line[1]. If the system is executed with its GUI and is running in proving mode, all applicable commands can be also selected from a menu that pops up when pressing the button  . Additionally various commands can be applied by pressing a button in the prover's GUI or by selecting an item in the menu associated to every formula in a proof state (which specializes the command for the application to this formula). A few commands can be selected from the GUI's window menu "File".

Derivation Rules and Sequents The effect of some commands of the proof states is explained by derivation rules that sketch the logical foundation of these commands. For a command that generates from the current state with label `[S]` various child states `[S1]`, `[S2]`, ... which is displayed in the proof tree as

```
[S] c
  [S1]
  [S2]
  ...
```

[1]The only exception is the "Abort" button  described on page 91 whose effect cannot be achieved by any textual command.

the corresponding rule has the form

$$\frac{\begin{array}{l}[E]\,A_1,\ldots \vdash B_1,\ldots \\ [E]\,A_2,\ldots \vdash B_2,\ldots \\ \ldots\end{array}}{[E]\,A,\ldots \vdash B,\ldots}$$

where a *sequent* $[E]\,A,\ldots \vdash B,\ldots$ represents a proof state with declaration environment E, assumptions $A,\ldots$ and goals $B,\ldots$. The sequent below the line represents the proof state labeled `[S]`, the sequents above the line represent the child states `[S1]`, `[S2]`, ... generated by the application of the command. Consequently, the rule can be read "from bottom to top" as

> In order to prove $[E]\,A,\ldots \vdash B,\ldots$, we have to prove $[E]\,A_1,\ldots \vdash B_1,\ldots$, and $[E]\,A_2,\ldots \vdash B_2,\ldots$, and ....

Furthermore, a sequent of the special form $[E] \vdash e : T$ means "in environment E, expression e has type T"; a sequent $[E] \vdash I : T = V$ means "in environment E, constant I has type T and value V" The environment $[E, a : T]$ is derived from environment $[E]$ by adding the declaration of a constant a of type T (shadowing any previous declaration of a constant with the same name).

Type Checking Conditions If the prover command contains a value expression (see page 70), it may give rise to a type checking condition. The prover handles this condition by generating a child state (in addition to the child states generated anyway by the command) with the type checking condition as a single goal (which may be automatically discharged by a decision procedure as in declaration mode, see page 76, or by an user-controlled proof).

Proof Status Every proof of a formula has a couple of status attributes which are displayed in the menu of the formula. These attributes are:

Trust Status: "trusted" or "untrusted" When a proof is read from file, it is given status "trusted", if none of the declarations and proofs on which the proof depends has changed since the time that the proof was generated. Replaying the proof thus cannot trigger an error. Otherwise the proof is given status "untrusted"; replaying the proof may trigger errors. Replaying an untrusted proof promotes its status to "trusted" (after having removed from the proof those commands that yielded errors in the replay).

Completion Status: "closed" or "open" A proof is closed, if every leaf of the proof tree denotes a proof situation that is automatically closed by the prover (via the application of an internal or external decision procedure). Otherwise the proof is "open": there are still proof states that require investigation by the user.

Dependence Status: "absolute" or "relative" A complete proof is "absolute", if it does not depend on assumptions introduced by application of command `lemma` (see page 106) or only on assumptions which have themselves absolute proofs. Otherwise the proof is "relative": its correctness depends on formulas that still need proof.

Proof Replay When a previously stored proof is replayed but the formula to be proved has changed since the generation of the proof, the proof is marked as "untrusted" and a warning is issued that errors may occur in the replay. There are two kinds of errors possible:

- A proof command triggers an error, i.e. the command applicable in a state of the original proof is not applicable any more in the corresponding state of the new proof. In this case, the system gives the user the possibility to provide a substitution command which is executed instead of the original command (continuing the replay of the proof subtrees on the child states). If the user abandons the offer, the corresponding proof state remains open (i.e. the proof subtrees corresponding to the child states are lost).
- A proof command generates in the new proof not the same number of child states than in the original proof. In this case, the system asks the user to map the new child states to the old child states such that the proof trees recorded for the child states are not completely lost.

In the first kind of error, the possibility to substitute a proof command is mainly useful if the command has failed because it refers to the label of a formula that has changed since the proof was created. In this case, the user may investigate the proof state, find the corresponding formula, and then execute the command again with a reference to the new label; the remainder of the proof is thus preserved.

As an example for the second kind of error, let us assume that the command `scatter` described on page 92 has in the original proof of a formula f been applied to a goal $a \wedge b$ generating two child states, one with goal a and one with goal b. If afterwards the declaration of f has been changed such that in the corresponding state of the new proof the goal has form $a \wedge b \wedge c$, the proof replay gives the following dialog:

```
Formula f already has a (skeleton) proof
  (proof status: untrusted)
Replay skeleton proof (y/n)? y
Warning: proof is untrusted, errors may occur.
Warning: problem in state [gda].
In the old proof, command "scatter" generated
  2 states:
0: expand a
1: expand b
In the new proof, the command generates 3 states
  with the following goals:
0: a
1: b
2: c
You can now map ranges [a..b] of new states to
  equally sized ranges [c..] of old states.
Enter start state a (0..2, -1 to abort): 0
Enter end state b (0..1, -1 to abort): 1
Enter start state c (0..0, -1 to abort): 0
Enter start state a (0..2, -1 to abort): -1
You have constructed the following mapping:
New states [0..1] are mapped to old states [0..1].
Do you want to use this mapping (y/n)? y
Proof state [wtn] is closed by decision procedure.
Proof state [nxu] is closed by decision procedure.
Proof replay successful.
```

If no mapping is provided by the user (answer -1 in the dialog), the proof trees generated for the child states in the old proof are lost.

B.1 Declaration Commands

This section lists those commands that are related to constant declarations and that are mainly applicable in the "declaration mode" of the system.

B.1.1 `read`: Read Declaration File

Synopsis

```
read "path"
```

Alternative Applications

- **Window Menu "File", Item "Read File"**: `read "`*path*`"`

Applicable In declaration mode.

Description This commands reads the commands in the text file denoted by *path* (which may be an elementary file name or a compound file path) and processes them as if they were typed in interactively. The file may include only commands that are legal in declaration mode and it may include another applications of `read` which read other files. If by such nested applications of `read` an attempt is made to recursively read a file, an error is reported.

Pragmatics A file that starts with an application of the command `newcontext` described on page 83 can be used as a simple substitute for a "module" facility (which the system lacks otherwise).

B.1.2 `newcontext`: Start New Declaration Context

Synopsis

```
newcontext
newcontext "path"
```

Applicable In declaration mode.

Description The command creates a new and empty declaration environment erasing from the system memory all previous constant declarations. The command associates this environment to a directory in the file system (creating the directory, if it does not yet exist) for persistently storing representations of the constants declared in the new environment and of proofs of formulas performed. If `newcommand` is invoked without argument, this directory is named `Proof-Navigator` and is located in the current working directory of the system (unless the system has been started with the option `--context`, see Appendix D). If the command is invoked with an argument "*path*", the directory with the name and location denoted by *path* is (if necessary created and) used rather than the default directory.

B.1.3 `tcc`: Print Type Checking Condition

Synopsis

```
tcc
```

Applicable In declaration mode.

Description For type checking a constant declaration, it may be necessary to check a condition which cannot be derived by static type checking but requires actual proving. When in proving mode, such a type checking condition gives rise to an additional proof state with the type checking condition as the goal. When in declaration mode, the system attempts to resolve a type checking condition silently by the application of an automated decision procedure; only if a condition cannot be automatically proved (which may or may not indicate an error), it is displayed to the user. By application of the command `tcc`, the last generated type checking condition (even if successfully proved) is displayed.

Pragmatics It is questionable whether the information provided by this command is really of interest to the general user. The command was mainly introduced for debugging purposes and will perhaps go away in a future version of the system.

B.1.4 `type`, `value`, `formula`: Print Constant Declaration

Synopsis

```
type T
value V
formula F
```

Alternative Applications

- **Menu of Type Constant *T***: `type T`
- **Menu of Value Constant *V***: `value V`
- **Menu of Formula Constant *F***: `formula F`

Applicable In declaration mode and in proving mode.

Description These commands print the declaration of a type constant T, value constant V, or formula constant F visible in the current environment.

B.1.5 `environment`: Print Environment

Synopsis

```
environment
```

Applicable In declaration mode and in proving mode.

Description This command prints the names and kinds of all constants (types, values, formulas) declared in the current environment.

B.1.6 `option`: Set System Option

Synopsis

```
option I
option I = "value"
```

Alternative Applications

- **Window Menu "Options", Item "No Automatic Simplification"**:
  `option autosimp="false"`
- **Window Menu "Options", Item "Automatic Simplification"**:
  `option autosimp="true"`

Applicable In declaration mode and in proving mode.

Description This command switches on the option named *I* respectively sets the *value* of this option. Currently the following option is supported:

`autosimp` This option may have the value `true` (default) or `false`. Setting the option affects the "simplification mode" of the system: If the option is `true`, all formulas in all proof states are automatically simplified with the help of an external transformation procedure. If the option is `false`, no automatic simplification is applied; in this case, the user may invoke the command `simplify` described on page 99 to explicitly trigger the simplification of selected proof states or formulas.

When a new proof is started, the proof is executed in the currently active simplification mode. This mode is recorded in the proof such that the proof may be stored and later continued respectively replayed independently of the simplification mode active at that time. If the command is executed during a proof, a single child state is generated whose simplification mode is set according to the value of the option. Thus different parts of a proof may operate in different simplification modes.

B.2 Control Commands

The commands listed in this section do not alter the proof; they rather enable the user to enter or leave a proof, to navigate within a proof, and to get additional information on proof states.

B.2.1 `prove`: Enter Proving Mode

Synopsis

`prove` *F*

Alternative Invocations

- **Menu of Formula Constant *F***: `prove` *F*

Applicable In declaration mode.

Description The command takes the name *F* of a formula. If the formula already has an associated proof, the user is asked whether to enter this proof or to start a new proof. The system consequently switches to proving mode setting the current proof state to the first open state in the proof.

Pragmatics Before leaving the proving mode by the command `quit` described on page 87, the user may decide whether to retain the proof originally associated to the formula or to overwrite it with the the newly constructed proof.

B.2.2 `quit`: Leave Current Mode

Synopsis

```
quit
```

Alternative Invocations

- **Button**  : `quit`
- **Window Menu "File", Item "Quit":** `quit`
- **Window Button "Close":** `quit`

Applicable In declaration mode and in proving mode.

Description If in declaration mode, the command lets the system terminate. If in proving mode, the command asks the user whether to retain any original proof or to overwrite it with the newly constructed proof and then switches from proving mode back to declaration mode.

Pragmatics A proof is only saved to file when the command `quit` is executed (in proving mode).

The item "Quit" in window menu "File" and the window button "close" terminate the system, even when in proving mode, after confirmation by the user.

B.2.3 `prev`, `next`: Cycle Through Open Proof States

Synopsis

```
prev
next
```

Alternative Invocations

- **Button**  : `prev`
- **Button**  : `next`

Applicable In proving mode.

Description All open proof states are internally organized in a list in an order as they are encountered by a depth-first left-to-right traversal of the proof tree. The command `next` switches the current state to its successor in the list (to the first state in the list, if the current state is the last one); the command `prev` switches the current state to its predecessor in the list (to the last state in the list, if the current state is the first one). Thus repeated applications of `next` respectively `prev` cycle through all open proof states.

Pragmatics To quickly switch to an open state distant from the current one, use the command `goto` described on page 89 respectively double-click on the corresponding item in the visual representation of the proof tree.

B.2.4 `undo`, `redo`: Undo/Redo Proof Commands

Synopsis

```
undo
undo S
redo
```

Alternative Invocations

- **Button**  : `undo`
- **Button**  : `redo`

Applicable In proving mode.

Description The command `undo` cancels the effect of the proof command that was executed in the parent of the current state; it re-opens the parent state (discarding all its child states) and makes it the new current state.

If the optional state label *S* is provided, it must denote some ancestor of the current state, i.e., some state on the path from the parent of the current state to the root of the proof tree. Then the command cancels the effects of all proof commands along this path such that the ancestor is re-opened and becomes the new current state. Thus the effect of this form of the command is the same as if just `undo` is executed multiple times until the ancestor is reached.

If a state is re-opened by applying `undo`, it internally still retains its discarded child states unless another application of a proof command generates new child states. Until this is the case, an application of `redo` is able to restore the subtree rooted in the current proof state to its shape before the application of `undo` that re-opened the current state (however, the new current state after a `redo` need not be the the same as the current state before the `undo` but may be just a sibling of it). A repeated application of `redo` along a path of proof states in which `undo` was applied thus restores the situation before these applications.

Pragmatics A user may want to discard an unsuccessful proof branch by multiple applications of `undo` but accidentally performs one `undo` too much, such that the parent state of the unsuccessful branch is re-opened and also all other (potentially successful) sibling branches are discarded. Then the user only needs to apply `redo` once to restore the accidentally discarded branches. If the user also wishes to re-store the unsuccessful branch, she just has to go to the root state of that branch and continue there with the application of `redo`.

B.2.5 `goto`: Go to Another Open Proof State

Synopsis

```
goto S
```

Alternative Invocations

- **Double Click on Proof Tree Item [*S*]**: `goto` *S*

Applicable In proving mode.

Description This command switches the current state to the open proof state denoted by label *S*. If *S* is not the label of an open state in the current proof, the command has no effect.

Pragmatics A single click on proof tree item *S* only displays the denoted proof state but does not change the current state (see the command `state` on page 91).

B.2.6 `counterexample`: Generate Counterexample

Synopsis

```
counterexample
```

Alternative Invocations

- **Button** : `counterexample`

Applicable In proving mode.

Description A decision procedure may generate for the current proof state

$$[E]\, A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$$

a *counterexample*, i.e. an interpretation for the constants declared in the environment E which it believes to satisfy the formula

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \neg B_1 \wedge \neg B_2 \wedge \ldots \wedge \neg B_m$$

The counterexample is given as a conjunction of equations $c = t$ respectively $c(\ldots) = t$ for non-Boolean constants/functions c and of atomic propositions p respectively $p(\ldots)$ for Boolean constants/functions (i.e. predicates) p.

Pragmatics The presented counterexample is often verbose and not very helpful. This counterexample is actually only a part of the counterexample generated by the decision procedure where (seemingly) uninteresting conditions have been removed. The generation of the counterexample may take some time; this process may be interrupted by pressing the abort button  .

B.2.7 Abort Prover Activity

Synopsis

- **Button** 

Applicable In proving mode, during prover activity.

Description Pressing the abort button interrupts several (not all) prover activities, in particular the application of a decision procedure attempting to close a state or producing a counterexample.

Pragmatics Not in all situations in which the button is enabled, pressing the button has an effect. There is no textual command that has the same effect as the abort button.

B.2.8 `state`: Display Another Proof State

Synopsis

```
state S
```

Applicable In proving mode.

Alternative Invocations

- **Single Click on Proof Tree Item [*S*]**: `state S`

Description This command displays the proof state denoted by label *S*. If *S* is not the label of a state in the current proof, the command has no effect.

Pragmatics A double click on proof tree item *S* also switches the current state to the denoted state (see the command `goto` on page 89).

B.2.9 `open`: List Open Proof States

Synopsis

```
open
```

Applicable In proving mode.

Description This command prints the list of open proof states in the order as they are encountered in a depth-first left-to-right search of the proof tree.

B.3 Primary Commands

This section lists the commands typically applied in the initial phase of decomposing a proof into simple proof states (`scatter`, `decompose`, `split`, possibly `induction`) and in the final phases of closing proof states (`autostar`, `auto`). In the middle proof phases, typically the commands in the next section are applied.

B.3.1 `scatter`: Scatter Proof State

Synopsis

```
scatter
```

Alternative Invocations

- **Button**  : `scatter`

Applicable In proving mode.

Description This command generates one or more child states by applying the commands `decompose` described on page 93 and `split` described on page 94. The application of `scatter` is recursively repeated for each child state until no more state is generated. If the initial applications of `decompose` and `split` do not generate new states, then `scatter` does not generate a child state.

Pragmatics For the recursive application of `split` being effective, `scatter` should be applied to a state with a single goal. The command then replaces the current state by a (potentially large) number of child states each of which has a number of (hopefully simple) assumptions and a single (hopefully simple) goal. Some of these states may have been already closed by a decision procedure; some others may be subsequently closed by the application of the `autostar` command described on page 97.

It is advisable to apply `decompose` manually before `scatter` in order to get insight into the current proof state before attempting to scatter it.

The effect of this command may be simulated by repeated applications of the commands `decompose` described on page 93 and `split` described on page 94.

B.3.2 `decompose`: Decompose Formulas

Synopsis

```
decompose
```

Applicable In proving mode.

Alternative Invocations

- **Button**  : `decompose`

Description The command generates a single child state by repeatedly applying the command `flatten` described on page 107, the command `skolemize` described on page 93, and a simplification procedure, until the resulting proof state does not change any more. If the resulting state equals the current state, no child state is generated.

Pragmatics The command generates at most *one* child state. This state can be considered as a simplified version of the current state and thus may be inspected to get further insight, e.g. to determine why a proof does not work or to find another proving strategy. An application of the command can be considered as safe in the sense that it does (should?) never complicate the proof.

This command is used by the command `scatter` described on page 92 (which may generate *multiple* child states).

The effect of this command may be simulated by repeated applications of the commands `flatten` described on page 107 and `skolemize` described on page 109.

B.3.3 `split`: Split Proof State

Synopsis

```
split
split F
```

Alternative Invocations

- **Button**  : `split`
- **Formula Menu [*F*]**: `split F`

Applicable In proving mode.

Description This command generates two or more child states by splitting some selected formula according to some of the rules listed below (there is at most one rule applicable). The form `split` may only applied in a proof state with a single goal which is thus implicitly selected for splitting. The form `split F` explicitly selects the formula (assumption or goal) with label S for splitting. The rules are applied recursively to the generated child states by splitting the formulas resulting from the previous split until no more splitting is possible. If no rule is applicable for splitting the selected formula, no child state is generated.

The applied rules are:

$$\frac{\begin{array}{l}[E]\ \ldots, A_1, \ldots \vdash \ldots \\ [E]\ \ldots, A_2, \ldots \vdash \ldots \\ \ldots \\ [E]\ \ldots, A_n, \ldots \vdash \ldots\end{array}}{[E]\ \ldots, A_1 \vee A_2 \vee \ldots \vee A_n, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, A_1, \ldots \quad [E]\ \ldots \vdash \ldots, A_2, \ldots \quad \ldots \quad [E]\ \ldots \vdash \ldots, A_n, \ldots}{[E]\ \ldots \vdash \ldots, A_1 \wedge A_2 \wedge \ldots \wedge A_n, \ldots}$$

$$\frac{[E]\ \ldots, \neg A_1, \ldots \vdash \ldots \quad [E]\ \ldots, A_2, \ldots \vdash \ldots}{[E]\ \ldots, A_1 \Rightarrow A_2, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, A_1 \Rightarrow A_2, \ldots \quad [E]\ \ldots \vdash \ldots, A_2 \Rightarrow A_1, \ldots}{[E]\ \ldots \vdash \ldots, A_1 \Leftrightarrow A_2, \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, A_1 \Rightarrow \neg A_2, \ldots \quad [E]\ \ldots \vdash \ldots, A_2 \Rightarrow \neg A_1, \ldots}{[E]\ \ldots \vdash \ldots, A_1 \text{ xor } A_2, \ldots}$$

Pragmatics By the recursive application of the rules, e.g. a single application of `split` to the assumption of the proof state

$$[E]\ A \vee (B \Rightarrow C \vee D) \vdash \ldots$$

generates four child states:

$$[E]\ A \vdash \ldots$$
$$[E]\ \neg B \vdash \ldots$$
$$[E]\ C \vdash \ldots$$
$$[E]\ D \vdash \ldots$$

The command is used by the command `scatter` described on page 92.

B.3.4 `induction`: Perform Mathematical Induction

Synopsis

```
induction
induction I
induction I in F
```

Alternative Invocations

- **Formula Menu [*F*]**: `induction ... in F`

Applicable In proving mode.

Description The command applies the principle of mathematical induction to a universally quantified goal with a natural number variable, i.e., a variable with type `NAT` (but see below for the potential application of the command to an integer variable). If the proof state has a single goal with a single universally quantified natural number variable, the command form `induction` selects this goal and its variable for performing the induction. If the proof state has a single goal with multiple universally quantified variables, the command form `induction` *`I`* selects the variable with name *I*. If the proof state has multiple goals, the command form `induction` *`I`* `in` *`F`* selects the goal with label *F* and the variable *I* in this goal. If *I* does not denote a natural number variable or *F* does not denote a universally quantified goal with such a variable, the command has no effect.

Otherwise, the command creates two child states which are identical to the current state with the following exceptions:

1. The first child state (the *induction base*) instantiates the variable in the goal by the number 0.
2. The second child state (the *induction step*) has as an additional assumption the original goal with the variable instantiated by a constant a not occurring in the environment of the current proof state and instantiates the variable in the goal by the term $a+1$.

If the quantified goal only has a single variable, the quantifier is dropped from the instantiated versions. The command thus implements the following rule:

$$\frac{\begin{array}{l}[E]\ \ldots \vdash\ \ldots, \forall \ldots : G[0/x], \ldots \\ [E, a:\mathbb{N}]\ \ldots, \forall \ldots : G[a/x] \vdash\ \ldots, \forall \ldots : G[a+1/x], \ldots \\ \neg \exists T : [E] \vdash\ a : T\end{array}}{[E]\ \ldots \vdash\ \ldots, \forall x \in \mathbb{N}, \ldots : G, \ldots}$$

The command may also be applied to an integer variable, i.e., a variable with type `INT`. In this case, a third goal is created which ensures the validity of the application of the induction principle by proving that the original goal can be proved by only considering non-negative integer values for the variable according to the following rule:

$$\frac{\begin{array}{l}[E]\ \ldots \vdash\ \ldots, \forall \ldots : G[0/x], \ldots \\ [E, a:\mathbb{N}]\ \ldots, \forall \ldots : G[a/x] \vdash\ \ldots, \forall \ldots : G[a+1/x], \ldots \\ [E]\ \ldots, \forall x \in \mathbb{Z}, \ldots : x \geq 0 \Rightarrow G \vdash\ \forall x \in \mathbb{Z}, \ldots : G \\ \neg \exists T : [E] \vdash\ a : T\end{array}}{[E]\ \ldots \vdash\ \ldots, \forall x \in \mathbb{Z}, \ldots : G, \ldots}$$

Pragmatics For better readability, the name of the induction constant a is derived from the name of the bound variable x by appending some natural number subscript ($x_0, x_1, \ldots$) respectively by replacing an already existing subscript.

B.3.5 `autostar`: Apply `auto` also to Sibling States

Synopsis

```
autostar
```

Alternative Invocations

- **Button**  : `autostar`

Applicable In proving mode.

Description This command applies the command `auto` described on page 98 to the current state and to all of its (not yet closed) subsequent siblings, i.e., to all (not yet closed) states that have the same parent as the current state and appear in the sequence of the parent's children after the current state. Some of these states may be closed; all others remain unchanged.

The proof does not record the application of `autostar` but the successful applications of `auto`; when the proof is replayed, only these applications are replayed.

Pragmatics This command is especially useful after an application of the command `scatter` described on page 92 which potentially generates a large number of child states with simple goals.

During the execution of `autostar`, the proof states already processed by `auto` are displayed and those that are successfully closed are recorded. If during the execution of `autostar` the abort button  is pressed, only the current invocation of `auto` is aborted while the execution of `autostar` itself continues with the next state in sequence. Thus `autostar` can be used to interactively probe for states that can be closed by `auto` without fear to get stuck in some long-running invocation or to lose successful invocations by aborting.

B.3.6 `auto`: Close State by Automatic Formula Instantiation

Synopsis

```
auto
auto F1, F2, ...
```

Alternative Invocations

- **Button**  : `auto`
- **Formula Menu [*F*]**: `auto F`

Applicable In proving mode.

Description This command generates a single child state by automatically instantiating selected quantified formulas by suitable instantiation terms according to the rules listed below. The instantiations terms are taken from those ground terms (terms without free variables) that appear in some formula (goal or assumption) of the state and whose type matches the type of the bound variable to be instantiated. The type of the instantiation expression e needs not equal the type of the bound variable; if necessary, a side condition $p(e)$ is added to the instantiated formula to make the instantiation legal.

If the command is applied as `auto`, all formulas (assumptions and goals) may be instantiated; if the application is `auto F1, F2, ...`, only the formulas with labels $F1, F2, \ldots$ may be instantiated.

The generated child state is only added to the current state, if it can be immediately closed by a decision procedure; otherwise it is discarded and the current state remains without child.

The applied rules are:

$$\frac{\begin{array}{l}[E]\ \ldots, \forall x \in T : A, \ldots, p(e) \Rightarrow A[e/x] \vdash \ldots \\ [E, p(e) \vdash\ e : T\end{array}}{[E]\ \ldots, \forall x \in T : A, \ldots \vdash \ldots}$$

$$\frac{\begin{array}{l}[E]\ \ldots \vdash \ldots, \exists x \in T : A, \ldots, p(e) \wedge A[e/x] \\ [E, p(e)] \vdash\ e : T\end{array}}{[E]\ \ldots \vdash \ldots, \exists x \in T : A, \ldots}$$

Pragmatics For performance reasons, only a limited number of instantiations is performed; thus `auto` may miss instantiation terms that are able to close the state, even if they appear in the proof state. In this case, one may try to use the form `auto F1, F2, ...` to limit the instantiations to the "interesting" quantified formulas of the current state.

Since no proof search is implemented, the automatic instantiation of formulas is only useful if it allows a decision procedure to close the state; therefore the command discards the generated state, if it cannot be immediately closed. Use the command `instantiate` described on page 104 to manually generate instantiations for further use in child states.

B.3.7 `simplify`: Simplify Formulas

Synopsis

```
simplify
simplify F
```

Alternative Invocations

- **Button**  : `simplify`
- **Formula Menu [*F*]**: `simplify F`

Applicable In proving mode with automatic simplification switched off.

Description If automatic simplification has been switched off (by use of the command `option autosimp="false"` described on page 85), this command may be used to trigger simplification explicitly: `simplify` simplifies all formulas in the current proof state; `simplify F` simplifies the formula denoted by label *F*.

Pragmatics The command applies an external transformation procedure: the resulting formula is logically equivalent to the input formula but not necessarily in a form that the user may consider "simpler".

B.4 Secondary Commands

This section lists those commands that are typically applied in the middle phase of constructing a proof; they require creativity from the user to determine the right proof strategy and to guide the system towards a successful proof completion.

B.4.1 *I* : *T*=*E*: Declare Value

Synopsis

```
I:T
I:T=E
```

Description A value declaration may not only be issued in declaration mode but also in proving mode. It extends the current proof state by a single child state whose environment contains the declaration implementing the rule

$$\frac{[E, I : T = E]\, A, \ldots \vdash B, \ldots}{[E]\, A, \ldots \vdash B, \ldots}$$

An additional proof state may be generated for a type checking condition corresponding to the value A and the matching of its type to T.

B.4.2 `assume`: Use and Prove Assumption

Synopsis

```
assume A
```

Applicable In proving mode.

Description This command takes a formula A and extends the current proof states by two child states. Both states are duplicates of the current state except that in the first state A is added as an assumption while in the second state A becomes the only goal and the negation(s) of the original goal(s) are added as assumptions. In other words, the command implements the rule

$$\frac{[E]\ \ldots_1, A \vdash \ldots_2 \qquad [E]\ \ldots_1, \neg\ldots_2 \vdash A}{[E]\ \ldots_1 \vdash \ldots_2}$$

An additional proof state may be generated for a type checking condition corresponding to the value *A*.

Pragmatics The two new proof states are logically equivalent to the proof states resulting from the corresponding application of the command `case` described on page 101. However, the application of `assume` puts in the second state *A* into the goal position rather rather than the original goal(s).

B.4.3 `case`: Perform Case Distinction

Synopsis

```
case A
case A1, A2, ..., An
```

Applicable In proving mode.

Description The command `case A` takes a formula *A* and extends the current proof states by two child states. Both states are duplicates of the current state except that in the first state *A* is added as an assumption while in the second state $\neg A$ is added as an assumption. In other words, the command implements the rule

$$\frac{[E]\ \ldots, A \vdash \ldots \qquad [E]\ \ldots, \neg A \vdash \ldots}{[E]\ \ldots \vdash \ldots}$$

An additional proof state may be generated for a type checking condition corresponding to the value *A*.

The more general form `case A1, A2, ..., An` implements the rule

$$\frac{\begin{array}{l}[E]\ \ldots, A_1 \vdash \ldots \\ [E]\ \ldots, \neg A_1, A_2 \vdash \ldots \\ \ldots \\ [E]\ \ldots, \neg A_1, \neg A_2, \ldots, \neg A_n \vdash \ldots\end{array}}{[E]\ \ldots \vdash \ldots}$$

Additional proof states may be generated for the type checking conditions corresponding to the values *A1*, *A2*, ...

Pragmatics The two proof states resulting from the application of `case` *A* are logically equivalent to the proof states resulting from the corresponding application of the command `assume` described on page 100. However, the application of `case` leaves in the second state the goal unchanged.

The net effect of the execution of `case` *A1*`,` *A2*`, ...` is the same as that of a sequence of commands `case` *A1*, `case` *A2*, ..., executed (each command, apart from the first one, executed in the second state resulting from the application of the previous command).

B.4.4 `expand`: Expand Definitions

Synopsis

```
expand I1, I2, ...
expand I1, I2, ... in F1, F2, ...
```

Alternative Invocations

- **Formula Menu [*F*]**: `expand ... in` *F*

Applicable In proving mode.

Description The command takes a list of identifiers *I1*,*I2*,... denoting names of constants (functions, predicates) whose values have been explicitly defined in their declarations. It generates a single child state by expanding all occurrences of these constants in the current proof state to their values. If the expansion yields

new occurrences of value constants whose names occur among these identifiers, these occurrences are also expanded, such that the resulting state is free of value constants with these names. The command may be also provided with a list of labels $F1, F2, \ldots$ of formulas of the current proof state; in this case, only the occurrences of the constants in these formulas are expanded. If some identifier does not denote the name of a value constant with an explicit definition or some label does not denote a formula in the current state, no child state is generated.

B.4.5 `flip`: Flip Formula

Synopsis

```
flip F
```

Alternative Invocations

- **Formula Menu [*F*]**: `flip F`

Applicable In proving mode.

Description The command takes the label F of a formula (assumption or goal) in the current proof state. It generates a single child state where the formula has been removed and the negated version of the formula, if an assumption, is added as a goal respectively, if a goal, is added as an assumption. The command thus implements the rules

$$\frac{[E]\ \ldots, \ldots \vdash \ldots, \neg A}{[E]\ \ldots, A, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots, \neg A, \vdash \ldots, \ldots}{[E]\ \ldots \vdash \ldots, A, \ldots}$$

If F is not the label of a formula in the current state, no child state is generated.

Pragmatics By application of this command, one may generate a proof state without assumption (corresponding to a single assumption `true`) or a proof state without goal (corresponding to a single goal `false`); the later case corresponds to the proving technique “proof by contradiction”.

B.4.6 `goal`: Make Formula Goal

Synopsis

```
goal F
```

Alternative Invocations

- **Assumption Menu [*F*]**: `goal F`

Applicable In proving mode.

Description The command takes the label F of an assumption in the current proof state. It generates a single child state where the formula has been removed from the assumptions and the negated version of the formula becomes the single goal; the negated versions of the original goals become additional assumptions. The command thus implements the rule

$$\frac{[E]\ \ldots_1, \ldots_2, \neg \ldots_3 \vdash \neg A}{[E]\ \ldots_1, A, \ldots_2 \vdash \ldots_3}$$

Pragmatics Some commands, especially the commands `scatter` described on page 92 and `split` described on page 94, treat the goal of a proof states specially in that they attempt to split goal formulas (but not assumptions). By application of the command `goal`, one may make the right formula the target of splitting in a subsequent application of `scatter` or `split`.

B.4.7 `instantiate`: Instantiate Variables in Formula

Synopsis

```
instantiate V1, V2, ... in F
```

Alternative Invocations

- **Formula Menu [*F*]**: `instantiate ... in F`

Applicable In proving mode.

Description This command takes the label F of a formula of the current state such that this formula is either a universally quantified assumption or an existentially quantified goal with n bound variables. It also takes n instantiation values $V_1, V_2, \ldots, V_n$ such that each V_i is either the reserved identifier '_' or a term with a type that matches the type T_i of the corresponding variable. The command then creates a child state that is identical to the current state except that an instantiated version of the universal assumption is added as an additional assumption respectively an instantiated version of the existential goal is added as an additional goal (see the rules depicted below). In the instantiated version of the formula, for every term V_i different from '_', every free occurrence of the corresponding variable in the body of the quantified formula is replaced by V_i (potentially renaming some variables bound within the body) and the variable is removed from the list of bound variables. If no value V_i is '_', all variables are replaced and the quantifier is dropped from the instantiation result. The type of term V_i needs not equal T_i; if necessary, a side condition $p(V_i)$ is generated and an additional proof state with goal $p(V_i)$ is created.

If F does not denote a universal assumption or existential goal of the current proof state, if the number of instantiation values $V1, V2, \ldots, V_n$ does not equal the number of bound variables of the formula, if some T_i cannot be type-checked or has a type that does not match the type of the corresponding variable, or if all instantiation terms are '_', no child state is generated.

The following rules describe the case of the instantiation of a quantified formula with two variables whose first is instantiated (corresponding to an application of the command `instantiate V1, _ in ...`):

$$\frac{\begin{array}{l}[E]\ \ldots, \forall x_1 \in T_1, x_2 \in T_2 : G, \ldots, \forall x_2 \in T_2 : G[V_1/x_1] \vdash \ldots \\ [E]\ \ldots, \forall x_1 \in T_1, x_2 \in T_2 : G, \ldots \vdash p(V_1) \\ [E, p(V_1)] \vdash V_1 : T_1\end{array}}{[E]\ \ldots, \forall x_1 \in T_1, x_2 \in T_2 : G, \ldots \vdash \ldots}$$

$$\frac{\begin{array}{l}[E]\ \ldots \vdash \ldots, \exists x_1 \in T_1, x_2 \in T_2 : G, \ldots, \exists x_2 \in T_2 : G[V_1/x_1] \\ [E]\ \ldots \vdash p(V_1) \\ [E, p(V_1)] \vdash V_1 : T_1\end{array}}{[E]\ \ldots \vdash \ldots, \exists x_1 \in T_1, x_2 \in T_2 : G, \ldots}$$

An additional proof state may be generated for a type checking condition corresponding to the values *V1*, *V2*, ...

B.4.8 `lemma`: Import Lemmas

Synopsis

```
lemma L1, L2, ...
```

Applicable In proving mode.

Description This command adds the formulas with labels `L1, L2, ...` as assumptions to the current proof state. These formulas must be defined in declaration mode before the formula that is currently proved. This implements the following rule:

$$\frac{\begin{array}{l}[E] \vdash L_1 : \text{formula} = F_1 \\ [E] \vdash L_2 : \text{formula} = F_2 \\ \ldots \\ [E]\ \ldots, F_1, F_2 | - \ldots\end{array}}{[E]\ \ldots \vdash \ldots}$$

By the application of this command, the status of the current proof also depends on the proof status of these formulas; in particular, the current proof is only considered complete, if also these formulas have complete proofs.

B.4.9 `typeaxiom`: Instantiate Type Axiom

Synopsis

```
typeaxiom V1, V2, ... in V
```

Applicable In proving mode.

Description For every constant declaration $c : T$ in the environment of the current proof state, the system implicitly adds an invisible "type axiom" $p_T(c)$ to the state which may be used as additional knowledge by a decision procedure unaware about the properties of type T (e.g., an assumption $x \geq 0$ for a constant $x : \mathbb{N}$). This may not be sufficient in case of a function $f : S \to T$ where for some *application* $f(a)$ a type axiom $p_T(f(a))$ should be added to the prove state (e.g.,

an assumption $f(a) \geq 0$ for a function $f : S \to \mathbb{N}$ and constant $a : S$). Likewise, this may not be sufficient for an array a : *ARRAY S OF T* where for some *array access* $a[i]$ a type axiom $p_T(a[i])$ should be added.

The command `typeaxiom` takes a value V which denotes a function with n parameters (respectively an array); it also takes n argument terms $V_1, V_2, \ldots, V_n$ whose types match the parameter types (respectively one argument term V_1 whose type matches the index type of the array). The command generates a child state which is identical to the current state but has the "type axiom" for the application of the function to the terms added as an explicit assumption. The types of the argument terms need not equal the types of the parameter terms; if necessary, a second child state with a side condition $p(V_1, V_2, \ldots, V_n)$ as a goal is generated. If no type axiom can be derived from the function/array or the types of the argument terms do not match the types of the parameters, no child state is generated.

An additional proof state may be generated for a type checking condition corresponding to the values *V1*, *V2*, ..., and the matching of their types to the types of the instantiation parameters.

Pragmatics The command makes knowledge implicit in the domains of function types explicit as assumptions in those (rather rare) cases where a proof state can be only closed by a decision procedure with the help of this knowledge. It is useful for functions whose domain involves subtypes of some base type, e.g. for functions whose domains involve the domain of natural numbers $\mathbb{N}$ or involve a type generated by the constructor `SUBTYPE`.

B.5 Basic Commands

This section lists those commands that are rarely applied directly by the user; their main purpose is to serve as "building blocks" for more powerful prover commands (in particular the commands `scatter` described on page 92 and `decompose` described on page 93).

B.5.1 `flatten`: Flatten Propositional Formulas

Synopsis

```
flatten
```

Applicable In proving mode.

Description The command generates a single child state by flattening every suitable propositional formula (assumption or goal) of the current proof state according to the rules listed below. For every formula of the current state, the command looks for an applicable rule (there can be at most one) and, if such a rule exists, applies it to the state; i.e. a rule is applied at most *once* for every formula (for a *repeated* application of the rules, see the command `decompose` on page 93). If no rule is applicable at all, no child state is generated.

The applied rules are:

$$\frac{[E]\ \ldots, \mathrm{neg}(A), \ldots \vdash \ldots}{[E]\ \ldots, \neg A, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, \mathrm{neg}(A), \ldots}{[E]\ \ldots \vdash \ldots, \neg A, \ldots}$$

$$\frac{[E]\ \ldots, A, B, \ldots \vdash \ldots}{[E]\ \ldots, A \wedge B, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, A, B, \ldots}{[E]\ \ldots \vdash \ldots, A \vee B, \ldots}$$

$$\frac{[E]\ \ldots, A \vdash \ldots, B, \ldots}{[E]\ \ldots \vdash \ldots, A \Rightarrow B, \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, (A \Rightarrow B) \wedge (B \Rightarrow A), \ldots}{[E]\ \ldots \vdash \ldots, A \Leftrightarrow B, \ldots}$$

$$\frac{[E]\ \ldots, (A \Rightarrow B) \wedge (B \Rightarrow A), \ldots \vdash \ldots}{[E]\ \ldots, A \Leftrightarrow B, \ldots \vdash \ldots}$$

$$\frac{[E]\ \ldots \vdash \ldots, (A \Rightarrow \neg B) \wedge (B \Rightarrow \neg A), \ldots}{[E]\ \ldots \vdash \ldots, A \text{ xor } B, \ldots}$$

$$\frac{[E]\ \ldots, (A \Rightarrow \neg B) \wedge (B \Rightarrow \neg A), \ldots \vdash \ldots}{[E]\ \ldots, A \text{ xor } B, \ldots \vdash \ldots}$$

An application $\mathrm{neg}(A)$ yields a formula which is logically equivalent to A but where negation only occurs at the level of atomic formulas. The operator neg is recursively defined on the syntactic structure of its argument as follows:

$\text{neg}(\neg A) := A$
$\text{neg}(A \wedge B) := \text{neg}(A) \vee \text{neg}(B)$
$\text{neg}(A \vee B) := \text{neg}(A) \wedge \text{neg}(B)$
$\text{neg}(A \Rightarrow B) := A \wedge \text{neg}(B)$
$\text{neg}(A \Leftrightarrow B) := A \text{ xor } B$
$\text{neg}(A \text{ xor } B) := A \Leftrightarrow B$
$\text{neg}(\text{let } \ldots \text{ in } A) := \text{ let } \ldots \text{ in } \text{neg}(A)$
$\text{neg}(\forall x \in T : A) := \exists x \in T : \text{neg}(A)$
$\text{neg}(\exists x \in T : A) := \forall x \in T : \text{neg}(A)$
$\text{neg}(a = b) := a \neq b$
$\text{neg}(a \neq b) := a = b$
$\text{neg}(A) := \neg A, \text{ for every other } A$

Pragmatics The effect of this command is subsumed by the effect of the command `decompose` described on page 93. The only reason to apply `flatten` directly is to observe step-by-step how `decompose` produces its result.

B.5.2 `skolemize`: Skolemize Quantified Formulas

Synopsis

```
skolemize
```

Applicable In proving mode.

Description The command generates a single child state by skolemizing every suitable quantified formula (assumption or goal) of the current proof state according to the rules listed below. For every formula of the current state, the command looks for an applicable rule (there can be at most one) and, if such a rule exists, applies it to the state; i.e. a rule is applied at most *once* for every formula (for a *repeated* application of the rules, see the command `decompose` on page 93). If no rule is applicable at all, no child state is generated.

The applied rules are:

$$\frac{\begin{array}{l}[E, a : T] \ \ldots \vdash \ \ldots, A[a/x], \ldots \\ \neg\exists T : [E] \vdash a : T\end{array}}{[E] \ \ldots \vdash \ \ldots, \forall x \in T : A, \ldots}$$

$$\frac{[E, a:T]\ \ldots, A[a/x], \ldots \vdash\ \ldots \quad \neg\exists T:[E] \vdash\ a:T}{[E]\ \ldots, \exists x \in T : A, \ldots \vdash\ \ldots}$$

In both rules, a new *skolem constant a* is generated that does not occur in the current proof state and which in the child state replaces the variable x bound by the quantifier in the current state.

Pragmatics For better readability, the name of the skolem constant a is derived from the name of the bound variable x by appending some natural number subscript ($x_0, x_1, \ldots$) respectively by replacing an already existing subscript.

The effect of this command is actually subsumed by the effect of the command `decompose` described on page 93. The only reason to apply `flatten` directly is to observe step-by-step how `decompose` produces its result.

Appendix C

System Installation

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

C.1 README

```
--------------------------------------------------------------------------------
README
Information on the RISC ProofNavigator.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
Copyright (C) 2005-, Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

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

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
--------------------------------------------------------------------------------

RISC ProofNavigator
-------------------
http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator

This is the RISC ProofNavigator, an interactive proof assistant for supporting
formal reasoning about computer programs and computing systems. This software
is freely available under the terms of the GNU General Public License, see
file COPYING.
```

```
The RISC ProofNavigator runs on computers with x86-compatible processors
running under the GNU/Linux operating system. For installation instructions,
see file INSTALL. For learning how to use the software, see the file
"main.pdf" in directory "manual"; several proof examples can be found in
directory "examples".

Please send bug reports to the author of this software:

  Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
  http://www.risc.uni-linz.ac.at/people/schreine
  Research Institute for Symbolic Computation (RISC)
  Johannes Kepler University
  A-4040 Linz, Austria

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

CVC Lite 2.0
http://www.cs.nyu.edu/acsys/cvcl
--------------------------------
This is a C++ library/program for validity checking in various theories.

The ProofNavigator currently only works with CVCL 2.0, not any of the later
CVCL versions available from the CVCL web site. To download the CVCL 2.0
source, go to the RISC ProofNavigator web site (URL see above), Section "Third
Party Software", and click on the link "CVCL 2.0 local copy".

RIACA OpenMath Library 2.0
http://www.riaca.win.tue.nl/products/openmath/lib
-------------------------------------------------
This is a library for converting mathematical objects to/from
the OpenMath representation.

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

General Purpose Hash Function Algorithms Library
http://www.partow.net/programming/hashfunctions
-----------------------------------------------
A library of hash functions implemented in various languages.

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

ANTLR 2.7.6b2
http://www.antlr.org
--------------------
This is a framework for constructing parsers and lexical analyzers.

On a Debian 3.1 GNU/Linux distribution, just install the package "antlr"
by executing (as superuser) the command

  apt-get install antlr

The Eclipse Standard Widget Toolkit 3.3
```

```
http://www.eclipse.org/swt
--------------------------------------
This is a widget set for developing GUIs in Java.

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

Mozilla Firefox 2.0.X or SeaMonkey  1.1.X (or higher)
http://www.mozilla.org
-----------------------------------------------------
See the question "What do I need to run the SWT browser in a standalone
application on Linux GTK or Linux Motif?" in the FAQ at
http://www.eclipse.org/swt/faq.php.

Chances are that the SWT browser will work with the Firefox included in your
Linux distribution (but it will *not* work with the Firefox downloaded from
the Mozilla site). For instance, on a Debian 4.0 GNU/Linux distribution, just
install Firefox by executing (as superuser) the command

  apt-get install iceweasel

If the SWT browser does not work with the Firefox included in your GNU/Linux
distribution, go to the page http://www.mozilla.org/projects/seamonkey to
download and install the SeaMonkey 1.1.4 browser instead. You might have to
set the environment variable MOZILLA_FIVE_HOME in the "ProofNavigator" script
to "/usr/lib/mozilla".

The GIMP Toolkit GTK+ 2.X (or higher)
http://www.gtk.org
-------------------------------------
This library is required by "Eclipse Linux (x86/GTK2)" and by
"Mozilla 1.7.8 GTK2".

On a Debian 3.1 GNU/Linux distribution, the package is automatically
installed, if you install the "mozilla-browser" package (see above).

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

Java Development Kit 5.0 (or higher)
http://java.sun.com/j2se/1.5.0
-----------------------------------
Go to the "Downloads" section to download the Sun JDK 5.0. The ProofNavigator
does currently not use any of the Java 5 language features and can therefore
be also compiled with JDK 1.4.2 (but this may change in the future).

Tango Icon Library 0.6.1
http://tango-project.org/
-------------------------
Go to the "Base Icon Library" section, subsection "Download", to download
the icons used in the ProofNavigator.

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

C.2 INSTALL

```
------------------------------------------------------------------------------
INSTALL
Installation notes for the RISC ProofNavigator.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
Copyright (C) 2005-, Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

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

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
------------------------------------------------------------------------------

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

To use the RISC ProofNavigator, you have three options:

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

The procedures for the three options are described below.

A Note for Microsoft Windows Users
----------------------------------
The RISC ProofNavigator does currently not run natively under MS Windows.

However, you can find on the RISC ProofNavigator web page a link to a virtual
machine with a Debian 4.0 GNU/Linux distribution that includes an installation
of the RISC ProofNavigator. The virtual machine is in the format of the free
virtualization environment VirtualBox (http://www.virtualbox.org). Install on
your MS Windows PC the VirtualBox software and the virtual machine (see the
web page for further instructions) and you can enjoy the RISC ProofNavigator
also under MS Windows.

A) Using the Distribution
-------------------------
We provide a distribution for computers with ix86-compatible processors
running under the GNU/Linux operating system (the software has been developed
on the Debian 3.1 "sarge" distribution, but any other distribution will work
as well). If you have such a computer, you need to make sure that you also
have
```

```
1) A Java 5 or higher runtime environment.

   You can download the Sun JRE 5.0 from
   http://java.sun.com/j2se/1.5.0/download.jsp

2) The Mozilla Firefox or SeaMonkey browser.

   On a Debian 4.0 GNU/Linux system, just install the package
   "iceweasel" by executing (as superuser) the command

     apt-get install iceweasel

   On other Linux distributions, first look up the FAQ on

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

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

   See the README file for further information.

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

   On a Debian 3.1 GNU/Linux system, GTK+ is automatically installed,
   if you install the Mozilla browser as described in the previous paragraph.

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

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

  ProofNavigator-VERSION.tgz

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

  tar zxf ProofNavigator-VERSION.tgz

This will create the following files

  README           ... the readme file
  INSTALL          ... the installation notes (this file)
  CHANGES          ... the change history
  COPYING          ... the GNU Public License
  bin/
    ProofNavigator ... the main script to start the program
    cvcl           ... CVC Lite, a validity checker used by the software.
  doc/
    index.html     ... API documentation
  examples/
    README         ... short explanation of examples
    *.pn           ... some example specifications and proofs
  lib/
    *.jar          ... Java archives with the program classes
    swt32/         ... SWT for GNU/Linux computers with 32 bit processors
      swt.jar
      *.so
    swt64/         ... SWT for GNU/Linux computers with 64 bit processors
      swt.jar
      *.so
  manual/
```

```
    main.pdf       ... the PDF file for the manual
    index.html     ... the root of the HTML version of the manual
  src/
    fmrisc/        ... the root directory of the Java package "fmrisc"
      ProofNavigator/
        Main.java  ... the main class for the ProofNavigator
```

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

Put the "bin" directory into your PATH

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

You should now be able to execute

```
  ProofNavigator
```

If you happen to get a message that ends with

```
  ...
  ERROR: You may try "ProofNavigator --nogui".
```

your Mozilla installation could not be detected. In this case, please set the value of the variable MOZILLA_FIVE_HOME in the "ProofNavigator" script to the location of the Mozilla libraries (e.g. /usr/lib/mozilla). In any case, you should be able to start

```
  ProofNavigator --nogui
```

to get a text-only interface.

B) Compiling the Source Code

To compile the Java source, first make sure that you have the Java 5 development environment installed. You can download the Sun JDK 5.0 from

```
   http://java.sun.com/j2se/1.5.0/download.jsp
```

Furthermore, on a GNU/Linux system you need also the Mozilla browser 1.7.X GTK2 and the GIMP toolkit GTK+ 2.X installed (see Section A).

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

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

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

```
  javac -cp ".:../lib/antlr.jar:../lib/om-lib.jar:../lib/swt.jar"
    fmrisc/ProofNavigator/Main.java
```

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

```
Then execute

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

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

C) Downloading the Source Code from the Subversion Repository
-------------------------------------------------------------
You can now download the source code of any version of the ProofNavigator
directly from the ProofNavigator Subversion repository.

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

To download the source code, you need a Subversion client, see
http://subversion.tigris.org/project_links.html for a list of available
clients. On a computer with the Debian 3.1 distribution of GNU/Linux, it
suffices to install the "svn" package by executing (as superuser) the command

  apt-get install svn

which will provide the "svn" command line client.

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

  svn://svn.risc.uni-linz.ac.at/schreine/FM-RISC/tags/VERSION

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

  svn export
    svn://svn.risc.uni-linz.ac.at/schreine/FM-RISC/tags/VERSION SOURCEDIR

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

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

------------------------------------------------------------------------------
End of INSTALL.
------------------------------------------------------------------------------
```

Appendix D

System Invocation

Invoking the system by the the command `ProofNavigator -h` gives output which lists the available startup options:

```
RISC ProofNavigator Version 1.1 (October 24, 2007)
http://www.risc.uni-linz.ac.at/research/formal/software/...
(C) 2005-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "ProofNavigator -help" to see the options available.
------------------------------------------------------------------
Usage: ProofNavigator [OPTION]... [FILE]
FILE:   name of file to be read on startup.
OPTION: one of the following options:
  -n, --nogui:  use command line interface
  -c, --context NAME: use subdirectory NAME to store context.
  --cvcl PATH: PATH refers to executable "cvcl".
  -s, --silent: omit startup message.
  -h, --help:   print this message.
```

The command optionally receives the name (in general path) of a declaration file which is read and processed as described for the command `read` on page 82. The command also accepts various startup options:

-n, –nogui Start the system without graphical user interface relying on the textual command line interface only.

Currently, this option is mainly useful in case of problems with the installation of the software, if the necessary GUI library cannot be found (see Appendix C). In the future, the option may become useful for non-interactive applications of the system.

-c, –context *NAME* When the system starts up, it creates in the current working directory a subdirectory `ProofNavigator` for storing the session context (see also Appendix E and the command `newcontext` described on page 83. With this option, the subdirectory is given a different name and/or different location as specified by the parameter *NAME* (which must denote a directory path).

–cvcl *PATH* The system currently uses CVCL (CVC Lite) [2, 1] Version 2.0 as an external decision procedure and assumes that the CVCL executable `cvcl` can be found in the current `PATH`. With the option `--cvcl` an alternative location and/or name of the executable can be specified by the parameter *PATH* (which must denote a file path).

This option is only used within the `ProofNavigator` script for customization of the installation (see Appendix C); passing it to the script has no effect.

-s, –silent With this option, the startup message is suppressed.

-h, –help With this option, the description shown above is printed and the system is terminated.

Appendix E

Context Directory

The system generates the following files in the context directory:

`cvcl.log` A log file describing the interaction between the system and the CVCL instance used for automatically verifying the type checking conditions generated in declaration mode. This file can be erased after a session without consequence.

`cvcl0.log` A log file describing the interaction between the system and the CVCL instances used to simplify formulas in a proof state respectively close a proof state. This file can be erased after a session without consequence (and perhaps should be erased, since it can become very large).

`*.xhtml` A collection of files presenting the declarations and proof states in XHTML+MathML format. The files use the following DOCTYPE declaration specified by W3C (see Section "A.2.3 MathML as a DTD Module" of [6]):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.1 plus MathML 2.0//EN"
  "http://www.w3.org/TR/MathML2/dtd/xhtml-math11-f.dtd">
```

The files are primarily intended for internal use by the graphical user interface of the system. However, after a session they can be also copied to another location for the persistent presentation of declarations and proofs. The files can be viewed by any Web browser that understands above document type, e.g. the browsers of the Mozilla suite (as well the former Mozilla browser as the current Firefox and SeaMonkey browsers). Most notably, the MicroSoft Internet Explorer Version 6 does *not* understand these declarations and thus cannot be used for viewing the files.

The entry files to the presentations are:

decl-list.xhtml The list of declarations issued in declaration mode.

***F*.xhtml** The proof of formula *F*.

kind_name_*. For every declaration of a constant *name* of a particular *kind* (*type*, *value*, *formula*), four files are generated that are mainly used to maintain declaration and proof dependencies across different sessions of the system and thus allow to assess the status of a proof generated in a previous session with respect to the declarations of the current session.

***kind_name_*decl.xgz** This is an XML file compressed with the compression tool `gzip`; its plain XML content can be printed with the tool `zcat`.

The file contains an OMDoc [10] representation of the declaration with mathematical objects represented according to the OpenMath (OM) standard [5] using as far as possible symbols from standard OM content dictionaries; if for some concept of the specification language no appropriate standard symbol exists, an ad-hoc symbol with the content dictionary prefix `fmrisc` is used instead.

These files are actually not (yet) used by the system; in a future version they may provide a bridge to third-party tools understanding OMDoc.

***kind_name_*hash.txt** This is a text file that contains a single decimal number representing a hash code for the declaration. If the hash code of the declaration in the current session coincides with the value stored in this file in a previous session, it is assumed that this declaration has not changed since then.

***kind_name_*time.txt** This is a text file that contains a single decimal number representing the time stamp when the corresponding constant declaration has been generated respectively the definition of the constant has most recently changed.

***kind_name_*refs.xml** This is an XML file that lists all entities referenced by the corresponding constant together with the values of their time stamps. If the definition of a referenced entity changes in a subsequent session, the value in its time stamp file is updated and does not coincide any more with the value listed in this file. This discrepancy is consequently detected and reflected in the status of a proof (see page 80) depending on this constant.

proof_*name*_*.* For every proof of formula *name*, two files are generated that allow to assess the status of a proof in a later session (with respect to the declarations of that session) and to replay the proof.

proof_*name*_decl.xgz This is an XML file compressed with the compressed with the compression tool `gzip`; its plain XML content can be printed with the tool `zcat`.

The file describes the tree structure of the proof including the proof commands issued at every node of the tree in an ad-hoc XML format. Specification language entities used in these commands are described according to to the OpenMath (OM) standard [5] using as far as possible symbols from standard OM content dictionaries; if for some concept of the specification language no appropriate standard symbol exists, an ad-hoc symbol with the content dictionary prefix `fmrisc` is used instead.

proof_*name*_refs.xgz This is an XML file that lists all entities referenced by the proof together with the values of their time stamps (as already discussed above).

For illustration of the content of the `*_decl.xgz` files, let us consider the example of Section 3.1 with declarations

```
sum: NAT->NAT;
S1: AXIOM sum(0)=0;
S2: AXIOM FORALL(n:NAT): n>0 => sum(n)=n+sum(n-1);
S: FORMULA FORALL(n:NAT): sum(n) = (n+1)*n/2;
```

and proof

```
[tca]: induction n byu
  [dbj]: proved (CVCL)
  [ebj]: auto
    [k5f]: proved (CVCL)
```

The contents of the correspondingly generated declaration and proof files are (after uncompression) as follows:

value_sum_decl.xgz

```
<?xml version="1.0" encoding="UTF-8"?>
<omdoc:omgroup
  xmlns:omdoc="http://www.mathweb.org/omdoc"
  xmlns:fmrisc="http://www.risc.uni-linz.ac.at/FM-RISC"
  xmlns:om="http://www.openmath.org/OpenMath">
  <omdoc:symbol kind="object" name="sum">
    <omdoc:type system="simply_typed" xml:id="sum_type">
```

```
        <om:OMA>
          <om:OMS cd="sts" name="mapsto"/>
          <om:OMA>
            <om:OMS cd="set1" name="cartesian_product"/>
            <om:OMS cd="setname1" name="N"/>
          </om:OMA>
          <om:OMS cd="setname1" name="N"/>
        </om:OMA>
      </omdoc:type>
    </omdoc:symbol>
  </omdoc:omgroup>
```

formula_S1_decl.xgz

```
<?xml version="1.0" encoding="UTF-8"?>
<omdoc:omgroup
  xmlns:omdoc="http://www.mathweb.org/omdoc"
  xmlns:fmrisc="http://www.risc.uni-linz.ac.at/FM-RISC"
  xmlns:om="http://www.openmath.org/OpenMath">
  <omdoc:assertion type="axiom" xml:id="S1">
    <om:FMP>
      <om:OMA>
        <om:OMS cd="relation1" name="eq"/>
        <om:OMA>
          <om:OMV name="sum"/>
          <om:OMI>0</om:OMI>
        </om:OMA>
         <om:OMI>0</om:OMI>
      </om:OMA>
     </om:FMP>
   </omdoc:assertion>
</omdoc:omgroup>
```

formula_S2_decl.xgz

```
<?xml version="1.0" encoding="UTF-8"?>
<omdoc:omgroup
  xmlns:omdoc="http://www.mathweb.org/omdoc"
  xmlns:fmrisc="http://www.risc.uni-linz.ac.at/FM-RISC"
  xmlns:om="http://www.openmath.org/OpenMath">
  <omdoc:assertion type="axiom" xml:id="S2">
    <om:FMP>
      <om:OMBIND>
        <om:OMS cd="quant1" name="forall"/>
        <om:OMBVAR>
          <om:OMATTR>
            <om:OMATP>
              <om:OMS cd="sts" name="type"/>
```

```
            <om:OMS cd="setname1" name="N"/>
          </om:OMATP>
          <om:OMV name="n"/>
        </om:OMATTR>
      </om:OMBVAR>
      <om:OMA>
        <om:OMS cd="logic1" name="implies"/>
        <om:OMA>
          <om:OMS cd="relation1" name="gt"/>
          <om:OMV name="n"/>
          <om:OMI>0</om:OMI>
        </om:OMA>
        <om:OMA>
          <om:OMS cd="relation1" name="eq"/>
          <om:OMA>
            <om:OMV name="sum"/>
            <om:OMV name="n"/>
          </om:OMA>
          <om:OMA>
            <om:OMS cd="arith1" name="plus"/>
            <om:OMV name="n"/>
            <om:OMA>
              <om:OMV name="sum"/>
              <om:OMA>
                <om:OMS cd="arith1" name="minus"/>
                <om:OMV name="n"/>
                <om:OMI>1</om:OMI>
              </om:OMA>
            </om:OMA>
          </om:OMA>
        </om:OMA>
      </om:OMA>
    </om:OMBIND>
  </om:FMP>
</omdoc:assertion>
</omdoc:omgroup>
```

formula_S_decl.xgz

```
<?xml version="1.0" encoding="UTF-8"?>
<omdoc:omgroup
  xmlns:omdoc="http://www.mathweb.org/omdoc"
  xmlns:fmrisc="http://www.risc.uni-linz.ac.at/FM-RISC"
  xmlns:om="http://www.openmath.org/OpenMath">
  <omdoc:assertion type="formula" xml:id="S">
    <om:FMP>
      <om:OMBIND>
        <om:OMS cd="quant1" name="forall"/>
        <om:OMBVAR>
```

```
            <om:OMATTR>
              <om:OMATP>
                <om:OMS cd="sts" name="type"/>
                <om:OMS cd="setname1" name="N"/>
              </om:OMATP>
              <om:OMV name="n"/>
            </om:OMATTR>
          </om:OMBVAR>
          <om:OMA>
            <om:OMS cd="relation1" name="eq"/>
            <om:OMA>
              <om:OMV name="sum"/>
              <om:OMV name="n"/>
            </om:OMA>
            <om:OMA>
              <om:OMS cd="arith1" name="divide"/>
              <om:OMA>
                <om:OMS cd="arith2" name="times"/>
                <om:OMA>
                  <om:OMS cd="arith1" name="plus"/>
                  <om:OMV name="n"/>
                  <om:OMI>1</om:OMI>
                </om:OMA>
                <om:OMV name="n"/>
              </om:OMA>
              <om:OMI>2</om:OMI>
            </om:OMA>
          </om:OMA>
        </om:OMBIND>
      </om:FMP>
    </omdoc:assertion>
  </omdoc:omgroup>
```

proof_S_decl.xgz

```
  <?xml version="1.0" encoding="UTF-8"?>
  <fmrisc:proof
    xmlns:fmrisc="http://www.risc.uni-linz.ac.at/FM-RISC"
    xmlns:om="http://www.openmath.org/OpenMath"
    closed="true" autosimplify="true" name="S">
    <fmrisc:state>
      <fmrisc:command name="induction">
        <om:OMV name="n"/>
        <fmrisc:label>byu</fmrisc:label>
      </fmrisc:command>
      <fmrisc:state>
        <fmrisc:command name="proved">CVCL</fmrisc:command>
      </fmrisc:state>
      <fmrisc:state>
```

```
        <fmrisc:command name="auto"/>
        <fmrisc:state>
          <fmrisc:command name="proved">CVCL</fmrisc:command>
        </fmrisc:state>
      </fmrisc:state>
    </fmrisc:state>
  </fmrisc:proof>
```

Appendix F

Grammar

In this appendix, we give the lexical and the syntactical grammar of the specification language and the prover commands. The syntax is given in the notation of the ANTLR parser generator [13] used for the implementation of the system. Non-determinism in the syntax is resolved by extra means provided by ANTLR (like semantic predicates) which are not shown in this presentation.

F.1 Lexical Grammar

```
// identifiers, labels, numbers, strings
IDENT: REALLETTER (LETTER | DIGIT)* ;
LABEL: (LETTER | DIGIT) (LETTER | DIGIT)* ;
NUMBER: DIGIT (DIGIT)* ;
REALLETTER: ('a'..'z' | 'A'..'Z' );
LETTER: ('a'..'z' | 'A'..'Z' | '_');
DIGIT: ('0'..'9');
STRING : '"' (~('"' | '\n' | '\r' | '\f' | '\uffff'))*
         '"' ;
UNDERSCORE: '_';

// language tokens
LPAR: "(";
RPAR: ")";
LBRACK: "[";
RBRACK: "]";
LPARGRID: "(#";
RPARGRID: "#)";
LBRACKGRID: "[#";
RBRACKGRID: "#]";
PERIOD: ".";
```

```
COLON: ":";
SEMICOLON: ";";
COMMA: ",";
ASSIGNMENT: ":=";
EQUALITY: "=";
NONEQUALITY: "/=";
LESS: "<";
LESSEQ: "<=";
GREATER: ">";
GREATEREQ: ">=";
PLUS: "+";
MINUS: "-";
TIMES: "*";
DIVIDES: "/";
POWER: "^";
IMPLIES: "=>";
EQUIV: "<=>";
ARROW: "->";
DOTDOT: "..";

// whitespace (filtered by lexer)
WS: (' ' | '\t' | EOL | COMMENT)+ ;
EOL: ('\n' | '\r' | '\f') ;
COMMENT : '%' (~('\n' | '\r' | '\f' | '\uffff'))*
              (EOL | '\uffff') ;
```

F.2 Syntactical Grammar

```
// parsing of a declaration or prover command
main:
  declaration SEMICOLON
| "tcc" SEMICOLON
| "prove" IDENT SEMICOLON
| "flip" LABEL SEMICOLON
| "goal" LABEL SEMICOLON
| "skolemize" SEMICOLON
| "flatten" SEMICOLON
| "split" ( LABEL )? SEMICOLON
| "decompose" SEMICOLON
| "simplify" ( LABEL )? SEMICOLON
| "scatter" SEMICOLON
| "auto" ( LABEL ( ","  LABEL )* )? SEMICOLON
| "autostar" SEMICOLON
| "counterexample" SEMICOLON
| "assume" valueExp SEMICOLON
| "case" valueExp ( ","  valueExp )* SEMICOLON
| "lemma" IDENT ( "," IDENT )* SEMICOLON
```

```
| "instantiate" valueExp ( ","  valueExp  )*
    "in" LABEL SEMICOLON
| "typeaxiom" valueExp ( ","  valueExp  )*
    "in" IDENT SEMICOLON
| "expand" IDENT ( "," IDENT )*
    ( "in" LABEL ( "," LABEL )* )? SEMICOLON
| "induction" ( IDENT ( "in" LABEL )? )? SEMICOLON
| "open" SEMICOLON
| "next" SEMICOLON
| "prev" SEMICOLON
| "goto" LABEL SEMICOLON
| "undo" ( LABEL )? SEMICOLON
| "redo" ( LABEL )? SEMICOLON
| "environment" SEMICOLON
| "proof" ( IDENT )? SEMICOLON
| "state" ( LABEL ( IDENT )? )? SEMICOLON
| "type" IDENT SEMICOLON
| "value" IDENT SEMICOLON
| "formula" IDENT SEMICOLON
| "quit" SEMICOLON
| "read" STRING SEMICOLON
| "option" IDENT ( "=" STRING )? SEMICOLON
| "newcontext" ( STRING )? SEMICOLON
| SEMICOLON
| EOF
;

declaration:
  IDENT ":"
  ( "TYPE" ( "=" typeExp )?
  | typeExp ( "=" valueExp )?
  | "FORMULA" valueExp
  | "AXIOM" valueExp
  )
;

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

typeExpBase:
  IDENT
| "BOOLEAN"
| "INT"
| "NAT"
| "REAL"
```

```
| "[" typeExp ( ","  typeExp )+  "]"
| "[" typeExp "]"
| "[#" IDENT ":" typeExp ( "," IDENT ":" typeExp )* "#]"
| "SUBTYPE" "(" valueExp ")"
| "[" valueExp ".." valueExp "]"
| "(" typeExp ")"
;

valueExp:
  "LAMBDA" paramList ":" valueExp
| "ARRAY" paramList ":" valueExp
| "FORALL" paramList ":" valueExp
| "EXISTS" paramList ":" valueExp
| valueExp90
;

vdeclaration:
  IDENT ( ":" typeExp )? "=" valueExp
;

valueExp90:
  "LET" vdeclaration ( "," vdeclaration )* "IN" valueExp90
| valueExp70
;

valueExp70:
  valueExp60 "=>" valueExp70
| valueExp60 ( "<=>" valueExp70 | "XOR" valueExp70 )*
;

valueExp60:
  valueExp50 ( "OR" valueExp60 )*
;

valueExp50:
  valueExp45 ( "AND" valueExp50 )*
;

valueExp45:
  "NOT" valueExp45
| valueExp43
;

valueExp43:
  valueExp40 ( "=" valueExp43 | "/=" valueExp43 )?
;

valueExp40:
  valueExp30
```

```
  ( "<" valueExp40 | "<=" valueExp40 |
    ">" valueExp40 | ">=" valueExp40 )?
;

valueExp30:
  valueExp10 ( "+" valueExp30 | "-" valueExp30 )*
;

valueExp10:
  valueExp9 ( "*" valueExp9 | "/" valueExp9 )*
;

valueExp9:
  valueExp8 ( "^" valueExp8 )*
;

valueExp8:
  "+" valueExp6
| "-" valueExp6
| valueExp6
;

valueExp6:
  valueExp5
  ( "WITH" ( "." ( NUMBER | IDENT ) | "[" valueExp "]" )+
  ":=" valueExp )*

valueExp5:
  valueExp3
  (
    "." ( NUMBER | IDENT )
  | "[" valueExp "]"
  )*
;

valueExp3:
  valueExp0 ( "(" valueExp ( "," valueExp  )* ")" )*
;

valueExp0:
  IDENT
| UNDERSCORE
| NUMBER
| "TRUE"
| "FALSE"
| "(" valueExp ( ","  valueExp )* ")"
| "(#" IDENT ":=" valueExp ( "," IDENT ":=" valueExp )* "#)"
| "IF" valueExp "THEN" valueExp
  ( "ELSIF" valueExp "THEN" valueExp )*
```

```
  "ELSE" valueExp "ENDIF"
;

paramList:
  "(" param ( "," param )* ")"
;

param:
  IDENT ( "," IDENT )* ":" typeExp
;
```