



RODIN Deliverable 3.6 (D30)

Public Versions of Basic Tools and Platform

Editor: *Laurent Voisin (ETH Zurich)*

Public Document

October 29, 2007

<http://rodin.cs.ncl.ac.uk>

Contributors:

Jean-Raymond Abrial (ETH Zurich)

Stefan Hallerstede (ETH Zurich)

Thai Son Hoang (ETH Zurich)

Gabriel Katz (ETH Zurich)

Farhad Mehta (ETH Zurich)

Christophe Métayer (ClearSy)

François Terrier (ETH Zurich)

Laurent Voisin (ETH Zurich)

Foreword

This deliverable groups together four documents providing support for the users of the RODIN platform:

1. User Manual of the RODIN Platform
2. The Event-B Modelling Notation
3. The Event-B Mathematical Language
4. Tutorials for RODIN

All these documents, as well as the platform itself, are available from the SourceForge web site of the Rodin platform

`http://rodin-b-sharp.sourceforge.net` .

User Manual of the RODIN Platform

October 2007

Version 2.3

User Manual of the RODIN Platform

Contents

1	Project	1
1.1	Project Constituents and Relationships	1
1.2	The Project Explorer	2
1.3	Creating a Project	3
1.4	Removing a Project	4
1.5	Exporting a Project	4
1.6	Importing a Project	5
1.7	Changing the Name of a Project	5
1.8	Creating a Component	5
1.9	Removing a Component	6
2	Anatomy of a Context	6
2.1	Carrier Sets	8
2.1.1	Carrier Sets Creation Wizard.	8
2.1.2	Direct Editing of Carrier Sets.	8
2.2	Enumerated Sets	9
2.3	Constants	10
2.3.1	Constants Creation Wizard.	10
2.3.2	Direct Editing of Constants.	11
2.4	Axioms	12
2.4.1	Axioms Creation Wizard.	12
2.4.2	Direct Editing of Axioms.	12
2.5	Theorems	13
2.5.1	Theorems Creation Wizard.	13
2.5.2	Direct Editing of Theorems.	14
2.6	Adding Comments	15
2.7	Dependencies	15
2.8	Pretty Print	16

3	Anatomy of a Machine	17
3.1	Dependencies	17
3.2	Variables	18
3.2.1	Variables Creation Wizard.	18
3.2.2	Direct Editing of Variables.	18
3.3	Invariants	19
3.3.1	Invariants Creation Wizard.	19
3.3.2	Direct Editing of Invariants.	20
3.4	Theorems	20
3.4.1	Theorems Creation Wizard.	20
3.4.2	Direct Editing of Theorems.	21
3.5	Events	21
3.5.1	Events Creation Wizard.	21
3.5.2	Direct Editing of Events.	22
3.6	Adding Comments	24
3.7	Pretty Print	24
3.8	Dependencies: Refining a Machine	25
3.9	Adding more Dependencies	25
3.9.1	Abstract Event	25
3.9.2	Splitting an Event	26
3.9.3	Merging Events	26
3.9.4	Witnesses	26
3.9.5	Variant	27
4	Saving a Context or a Machine	28
4.1	Automatic Tool Invocations	28
4.2	Errors, The Problems Window	29
4.3	Preferences for the Auto-prover	30
5	The Proof Obligation Explorer	31
6	The Proving Perspective	34
6.1	Loading a Proof	34
6.2	The Proof Tree	34
6.2.1	Description of the Proof Tree	34

6.2.2	Decoration	35
6.2.3	Navigation within the Proof Tree	35
6.2.4	Hiding	36
6.2.5	Pruning	36
6.2.6	Copy/Paste	36
6.3	Goal and Selected Hypotheses	37
6.4	The Proof Control Window	38
6.5	The Smiley	40
6.6	The Operator "Buttons"	40
6.7	The Search Hypotheses Window	40
6.8	The Automatic Post-tactic	41
6.8.1	Rewrite rules	41
6.8.2	Automatic inference rules	47
6.8.3	Preferences for the Post-tactic	50
6.9	Interactive Tactics	50
6.9.1	Interactive Rewrite Rules	50
6.9.2	Interactive Inference rules	56
A	The Mathematical Language	60
B	ASCII Representations of the Mathematical Symbols	60
B.1	Atomic Symbols	61
B.2	Unary Operators	61
B.3	Assignment Operators	61
B.4	Binary Operators	62
B.5	Quantifiers	63
B.6	Bracketing	63

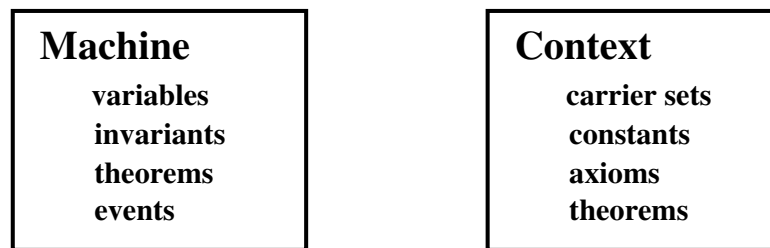
Foreword

The reader of this Manual is supposed to have a basic acquaintance with Eclipse. Basic help about using the Eclipse platform is available on-line from the RODIN platform. To view it, select **Help > Help Contents** from the menubar. Then, select **Workbench User Guide** in the Web browser window that pops up.

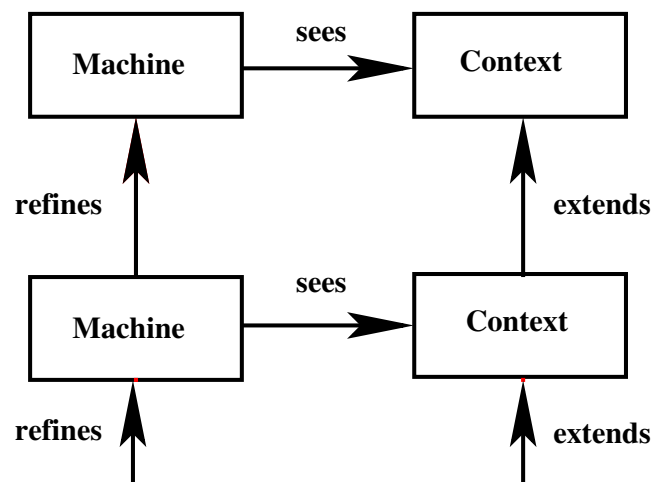
1 Project

1.1 Project Constituents and Relationships

The primary concept in doing formal developments with the RODIN Platform is that of a *project*. A project contains the complete mathematical development of a *Discrete Transition System*. It is made of several components of two kinds: *machines* and *contexts*. Machines contain the variables, invariants, theorems, and events of a project, whereas contexts contain the carrier sets, constants, axioms, and theorems of a project:

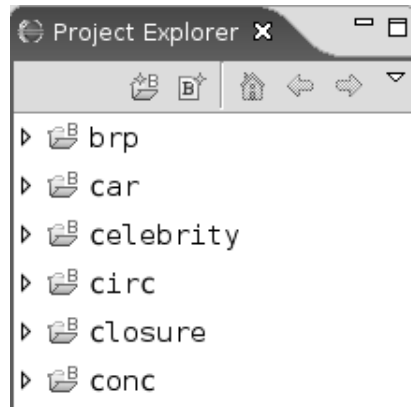


We remind the reader of the various relationships existing between machines and contexts. This is illustrated in the following figure. A machine can be "refined" by another one, and a context can be "extended" by another one (no cycles are allowed in either of these relationships). Moreover, a machine can "see" one or several contexts. A typical example of machine and context relationship is shown below:

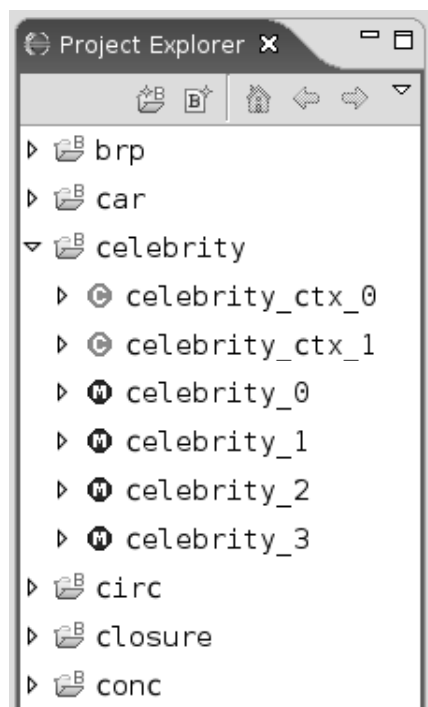


1.2 The Project Explorer

Projects are reachable in the RODIN platform by means of a window called the "Project Explorer". This window is usually situated on the left hand side of the screen (but, in Eclipse, the place of such windows can be changed very easily). Next is a screen shot showing a "Project Explorer" window:



As can be seen in this screen shot, the Project Explorer window contains the list of current project names. Next to each project name is a little triangle. By pressing it, one can expand a project and see its components as shown below.

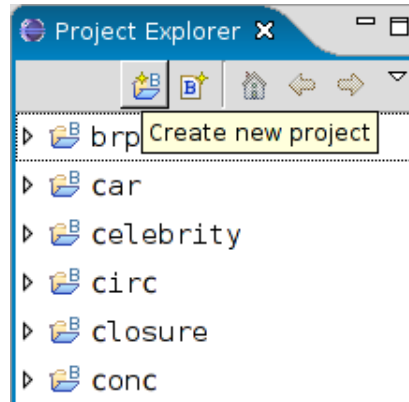


We expanded the project named "celebrity". This project contains 2 contexts named "celebrity_ctx_0" and "celebrity_ctx_1". It also contains 4 machines named "celebrity_0" to "celebrity_3". The icons ((c) or (m)) situated next to the components help recognizing their kind (context or machine respectively)

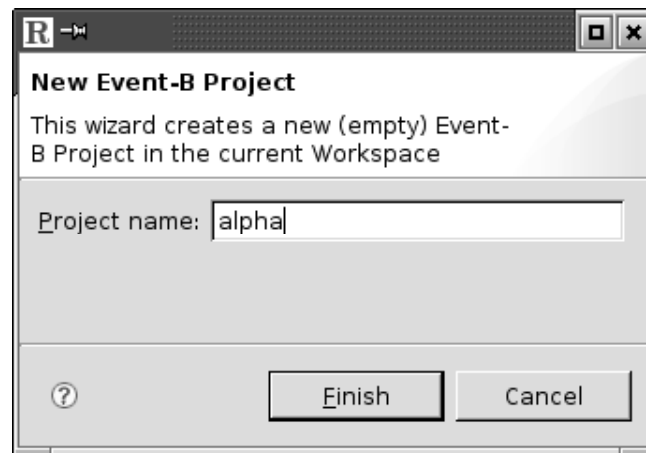
In the remaining parts of this section we are going to see how to create (section 1.3), remove (section 1.4), export (section 1.5), import (section 1.6), change the name (section 1.7) of a project, create a component (section 1.8), and remove a component (section 1.9). In the next two sections (2 and 3) we shall see how to manage contexts and machines.

1.3 Creating a Project

In order to create a new project, simply press the button as indicated below in the "Project Explorer" window:



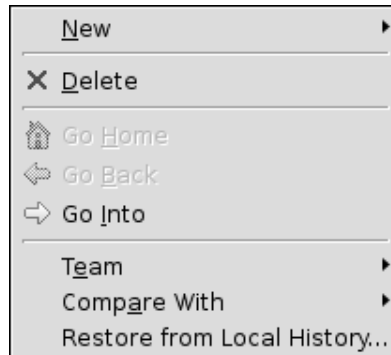
The following wizard will then appear, within which you type the name of the new project (here we type "alpha"):



After pressing the "Finish" button, the new project is created and appears in the "Project Explorer" window.

1.4 Removing a Project

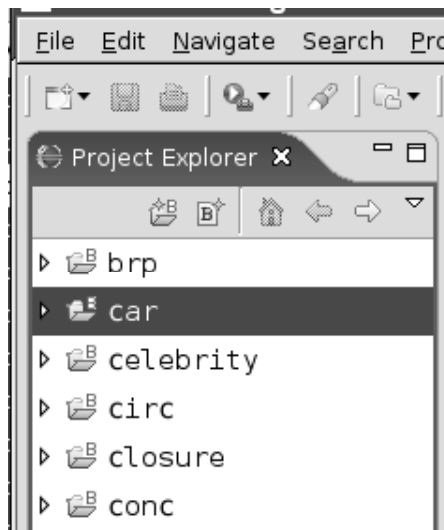
In order to remove a project, you first select it on the "Project Explorer" window and then right click with the mouse. The following contextual menu will appear on the screen:



You simply click on "Delete" and your project will be deleted (confirmation is asked). It is then removed from the Project Explorer window.

1.5 Exporting a Project

Exporting a project is the operation by which you can construct automatically a "zip" file containing the entire project. Such a file is ready to be sent by mail. Once received, an exported project can be *imported* (next section), it then becomes a project like the other ones which were created locally. In order to export a project, first select it, and then press the "File" button in the menubar as indicated below:



A menu will appear. You then press "Export" on this menu. The Export wizard will pop up. In this window, select "Archive File" and click "Next >". Specify the path and name of the archive file into which you want to export your project and finally click "Finish". This menu sequence belongs (as well as the various options) to Eclipse. For more information, have a look at the Eclipse documentation.

1.6 Importing a Project

A ".zip" file corresponding to a project which has been exported elsewhere can be imported locally. In order to do that, click the "File" button in the menubar (as done in the previous section). You then click "Import" in the coming menu. In the import wizard, select "Existing Projects into Workspace" and click "Next >". Then, enter the file name of the imported project and finally click "Finish". Like for exporting, the menu sequence and layout are part of Eclipse.

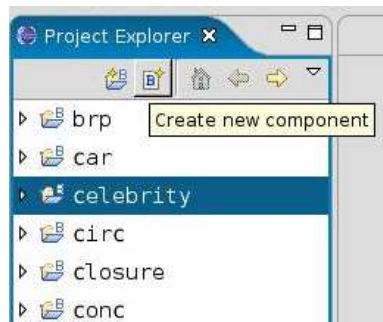
The importation is refused if the name of the imported project (not the name of the file), is the same as the name of an existing local project. The moral of the story is that when exporting a project to a partner you better modify its name in case your partner has already got a project with that same name (maybe a previous version of the exported project). Changing the name of a project is explained in the next section.

1.7 Changing the Name of a Project

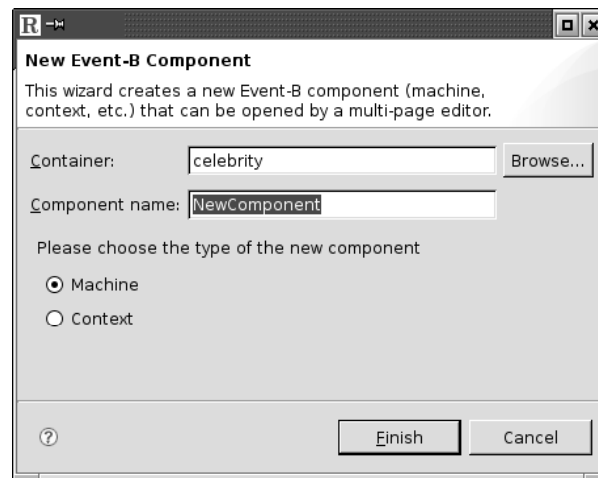
The procedure for changing the name of a project is a bit heavy at the present time. Here is what you have to do. Select the project whose name you want to modify. Enter the Eclipse "Resource" perspective. A "Navigator" window will appear in which the project names are listed (and your project still selected). Right click with the mouse and, in the coming menu, click "Rename". Modify the name and press enter. Return then to the original perspective. The name of your project has been modified accordingly.

1.8 Creating a Component

In this section, we learn how to create a component (context or machine). In order to create a component in a project, you have to first select the project and then click the corresponding button as shown below:



You may now choose the type of the component (machine or context) and give it a name as indicated:



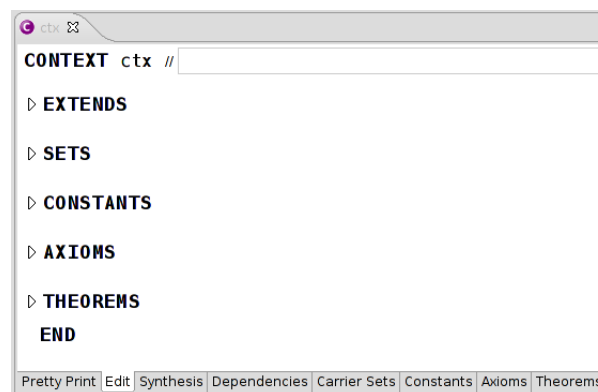
Click "Finish" to eventually create the component. The new component will appear in the Project Explorer window.

1.9 Removing a Component

In order to remove a component, press the right mouse button. In the coming menu, click "Delete". This component is removed from the Project Explorer window.

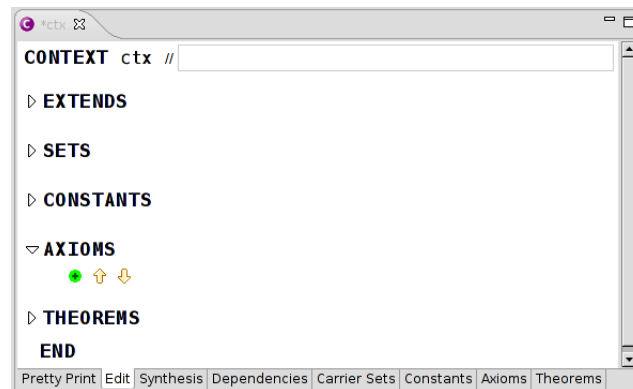
2 Anatomy of a Context

Once a context is created, a window such as the following appears in the editing area (usually next to the center of the screen):

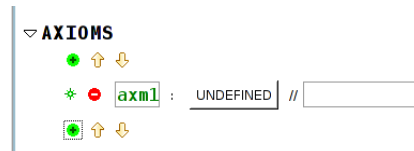


You are in the "Edit" area allowing you to edit pieces of the context, namely dependencies (keyword "EXTENDS"), carrier sets (keyword "SETS"), constants (keyword "CONSTANTS"), axioms (keyword

"AXIOMS"), or theorems (keyword "THEOREMS"). By pressing the triangle next to each keyword, you can add, remove, or move corresponding modelling elements. As an example, here is what you obtain after pressing the triangle next to the keyword "AXIOMS":



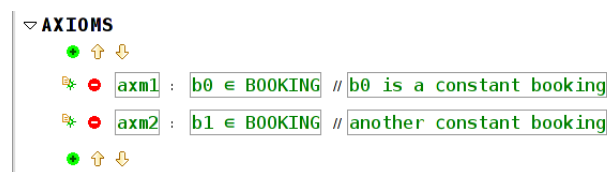
By pressing the (+) button, you obtain the following:



You can now enter a new axiom and a comment in the corresponding boxes as indicated below:



You can add another axiom in a similar fashion:



For removing an axiom, press the (-) button. You can also move an axiom up or down by selecting it (press mouse left key when situated on the axiom logo) and then pressing one of the two arrows: (↑) or (↓). Other modelling elements are created in a similar fashion.

It is also possible to do so in a different way as explained in sections 2.1 to 2.7. The creation of these elements, except dependencies (studied in section 2.7), can be made by two distinct methods, either by using wizards or by editing them directly. In each section, we shall review both methods.

NOTE: The hurried reader can skip these sections and go directly to section 2.8

2.1 Carrier Sets

2.1.1 Carrier Sets Creation Wizard.

In order to activate the carrier set creation wizard, you have to press the corresponding button in the toolbar as indicated below:



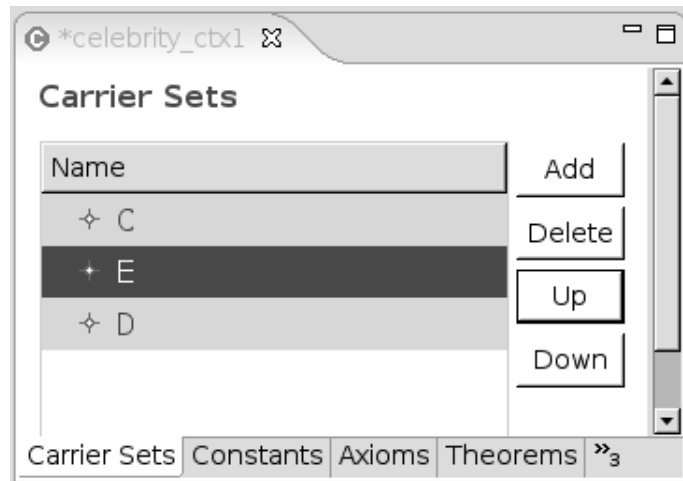
After pressing that button, the following wizard pops up:



You can enter as many carrier sets as you want by pressing the "More" button. When you're finished, press the "OK" button.

2.1.2 Direct Editing of Carrier Sets.

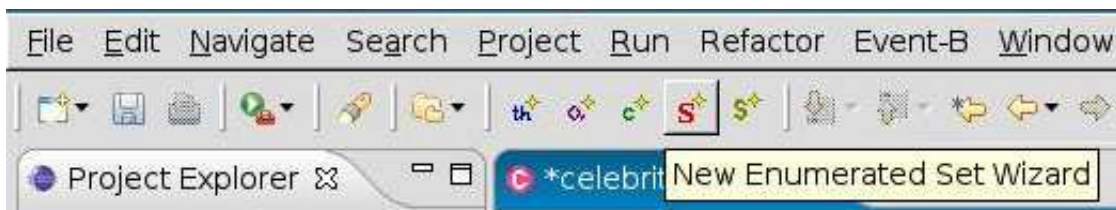
It is also possible to create (button "Add") or remove (button "Delete") carrier sets by using the central editing window (see window below). For this, you have first to select the "Carrier Sets" tab of the editor. Notice that you can change the order of carrier sets: first select the carrier set and then press button "Up" or "Down".



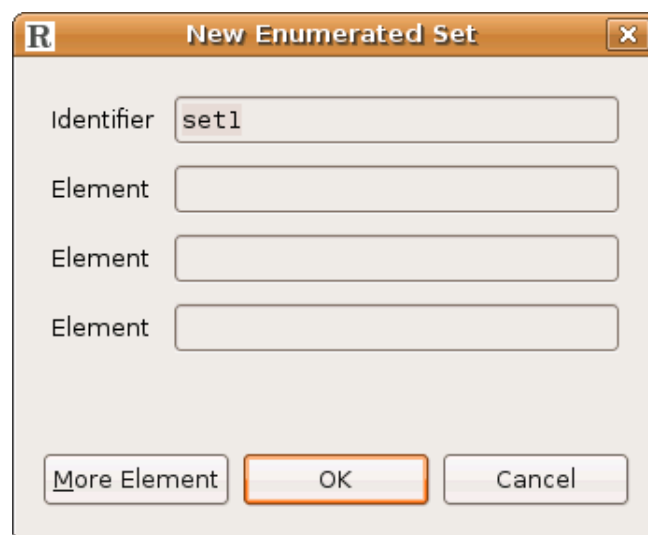
As can be seen, we have created three carrier sets C , E , and D .

2.2 Enumerated Sets

In order to activate the enumerated set creation wizard, you have to press the corresponding button in the toolbar as indicated below:



After pressing that button, the following wizard pops up:



You can enter the name of the new enumerated set as well as the name of its elements. By pressing the button "More Elements", you can enter additional elements. When you're finished, press the "OK" button.

The effect of entering "COLOR", "red", "green", and "orange" in this wizard is to add the new carrier set *COLOR* (section 2.1) and the three constants (section 2.3) *red*, *green*, and *orange*. Finally, it adds the following axioms (section 2.4):

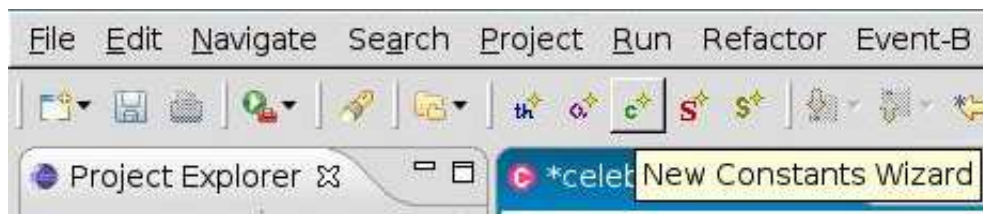
$$\begin{aligned} &COLOR = \{red, green, orange\} \\ &red \neq green \\ &red \neq orange \\ &green \neq orange \end{aligned}$$

If you enter several time the same enumerated set element, you get an error message.

2.3 Constants

2.3.1 Constants Creation Wizard.

In order to activate the constants creation wizard, you have to press the corresponding button in the toolbar as indicated below:



After pressing that button, the following wizard pops up:

You can then enter the names of the constants, and an axiom which can be used to define its type. Here is an example:

The 'New Constant' dialog box has a title bar with an 'R' icon and a close button. It contains two input fields: 'Identifier' with the text 'bit' and 'Axiom' with a small box containing 'axm6' and a larger box containing 'bit ∈ {0, 1}'. At the bottom are four buttons: 'Add', 'More Axm.', 'OK', and 'Cancel'.

By pressing the "More Axm." button you can enter additional axioms. For adding more constants, press "Add" button. When you're finished, press button "OK".

2.3.2 Direct Editing of Constants.

It is also possible to create (button "Add") or remove (button "Delete") constants by using the central editing window. For this, you have first to select the "Constants" tab of the editor. You can also change the relative place of a constant: first select it and then press button "Up" or "Down".

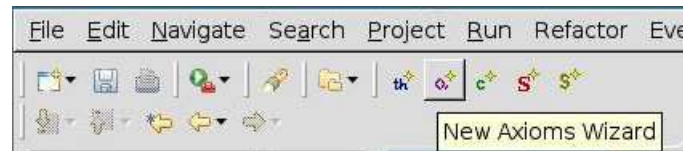
The central editing window has a title bar with a file icon, the text '*celebrity_ctx1', and window control buttons. The main area is titled 'Constants' and contains a list box with the following items: 'red', 'green', 'orange', 'bit', 'bit1' (which is selected and highlighted), and 'bit2'. To the right of the list box are four buttons: 'Add', 'Delete', 'Up', and 'Down'. At the bottom is a tab bar with five tabs: 'Carrier Sets', 'Constants' (which is active), 'Axioms', 'Theorems', and a button with the number '3'.

As can be seen, two more constants, *bit1* and *bit2* have been added. Note that this time the axioms concerning these constants have to be added directly (see next section 2.4).

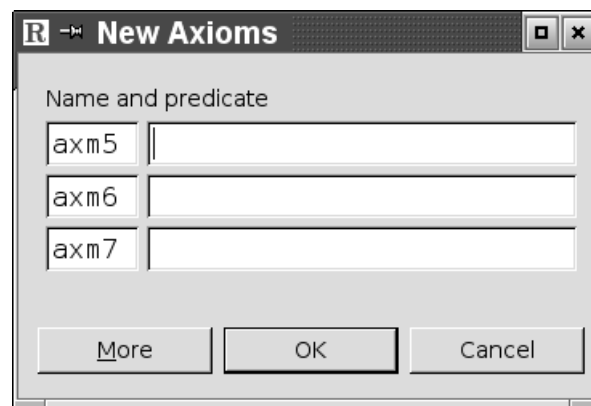
2.4 Axioms

2.4.1 Axioms Creation Wizard.

In order to activate the axioms creation wizard, you have to press the corresponding button in the toolbar as indicated below:



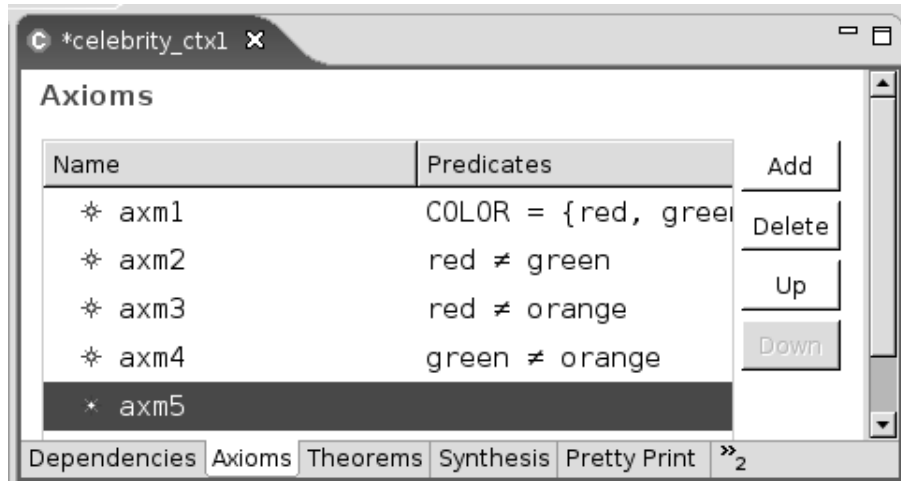
After pressing that button, the following wizard appears:



You can then enter the axioms you want. If more axioms are needed then press "More". When you are finished, press "OK".

2.4.2 Direct Editing of Axioms.

It is also possible to create (button "Add") or remove (button "Delete") axioms by using the central editing window. For this, you have first to select the "Axioms" tab of the editor. You can also change the relative place of an axiom: first select it and then press button "Up" or "Down".



Note that the "Up" and "Down" buttons for changing the order of axioms are important for well-definedness. For example the following axioms in that order

$$\begin{aligned} \text{axm1} : y/x = 3 \\ \text{axm2} : x \neq 0 \end{aligned}$$

does not allow to prove the well-definedness of $y/x = 3$. The order must be changed to the following:

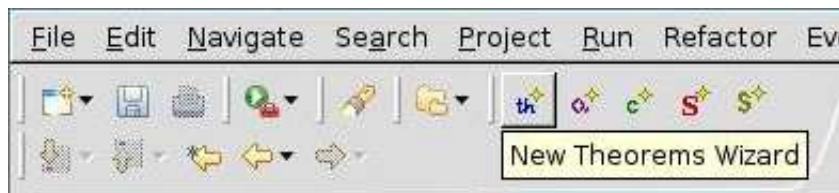
$$\begin{aligned} \text{axm2} : x \neq 0 \\ \text{axm1} : y/x = 3 \end{aligned}$$

The same remark applies to theorems (section 2.5 and 3.4), invariants (section 3.3) and event guards (section 3.5.2)

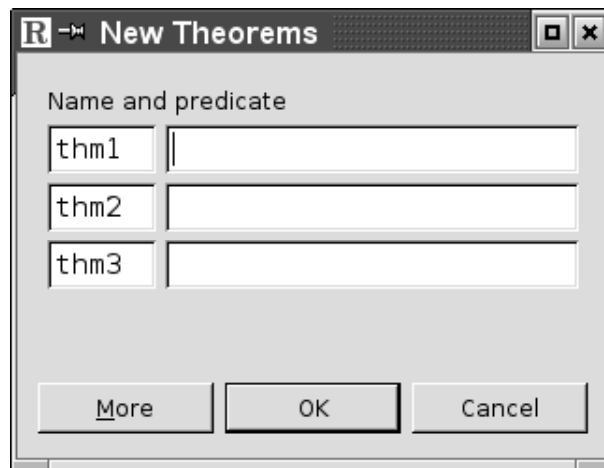
2.5 Theorems

2.5.1 Theorems Creation Wizard.

In order to activate the theorems creation wizard, you have to press the corresponding button in the toolbar as indicated below:



After pressing that button, the following wizard pops up:



A dialog box titled "New Theorems" with a standard window title bar. It contains a section labeled "Name and predicate" with three rows of input fields. The first row has "thm1" in a small box and an empty text field. The second row has "thm2" in a small box and an empty text field. The third row has "thm3" in a small box and an empty text field. At the bottom, there are three buttons: "More", "OK", and "Cancel".

You can then enter the theorems you want. If more theorems are needed then press "More". When you are finished, press "OK".

2.5.2 Direct Editing of Theorems.

It is also possible to create (button "Add") or remove (button "Delete") theorems by using the central editing window. For this, you have first to select the "Theorems" tab of the editor. You can also change the relative place of a theorem: first select it and then press button "Up" or "Down".

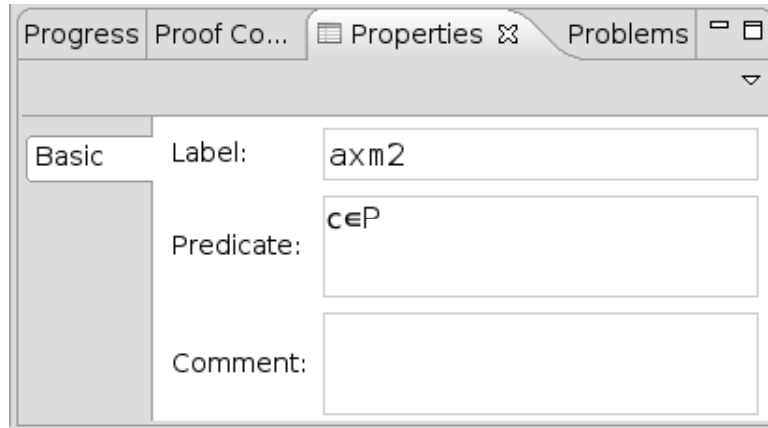


A window titled "*celebrity_ctx_1" with a tab labeled "Theorems". It contains a table with two columns: "Label" and "Predicate". The "Label" column has a single entry "thm1" which is selected. To the right of the table are four buttons: "Add", "Delete", "Up", and "Down".

Label	Predicate
thm1	

2.6 Adding Comments

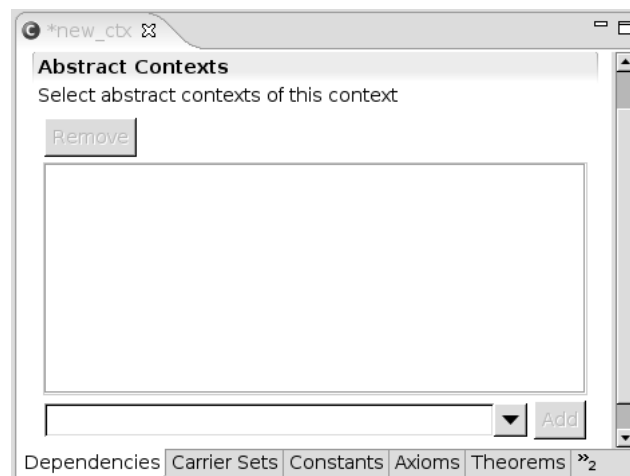
It is possible to add comments to carrier sets, constants, axioms and theorems. For doing so, select the corresponding modeling element and enter the "Properties" window as indicated below where it is shown how to add comments on a certain axiom:



Multiline comments can be added in the editing area labeled "Comments".

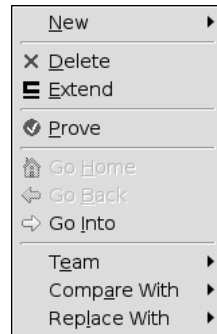
2.7 Dependencies

By selecting the "Dependencies" tab of the editor, you obtain the following window:

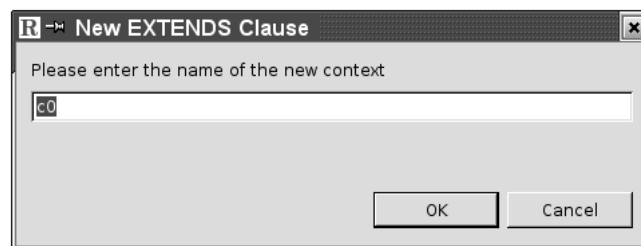


This allows you to express that the current context is *extending* other contexts of the current project. In order to add the name of the context you want to extend, use the combobox which appears at the bottom of the window and then select the corresponding context name.

There is another way to directly create a new context extending an existing context C . Select the context C in the project window, then press the right mouse key, you will get the following menu:



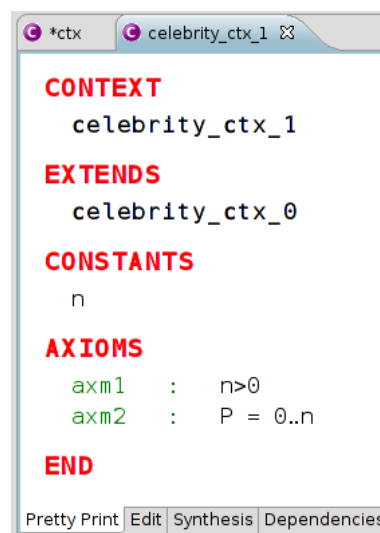
After pressing the "Extend" button, the following wizard pops up:



You can then enter the name of the new context which will be made automatically an extension of C .

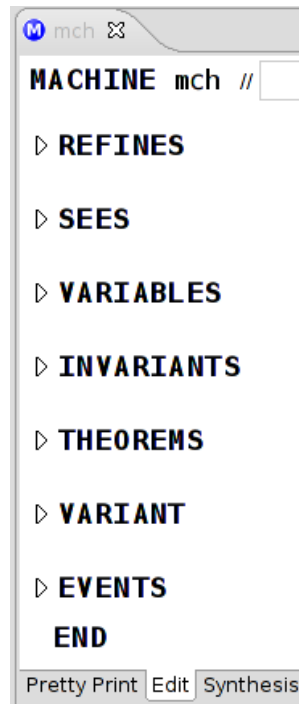
2.8 Pretty Print

By selecting the "Pretty Print" tab, you may have a global view of your context as if it would have been entered through an input text file.



3 Anatomy of a Machine

Once a machine is created, a window such as the following appears in the editing area:



You are in the "Edit" page allowing you to edit elements of the machine, namely dependencies (keywords "REFINES" and "SEES"), variables (keyword "VARIABLES"), invariants (keyword "INVARIANTS"), theorems (keyword "THEOREMS"), variants (keyword "VARIANTS"), or events (Keywords "EVENTS"). These modelling elements can be edited in a way that is similar to what has been explained for contexts in section 2, it is not repeated here.

It is also possible to do so in a different way as explained in sections 3.1 to 3.6. The creation of these elements, except dependencies can be made by two distinct methods, either by using wizards or by editing them directly. In each section, we shall review both methods.

NOTE: The hurried reader can skip these sections and go directly to section 3.7

3.1 Dependencies

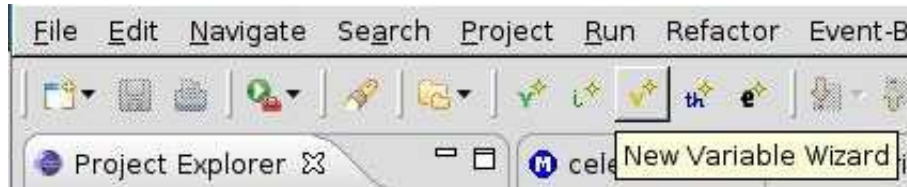
The "Dependencies" window is shown automatically after creating a machine (you can also get it by selecting the "Dependencies" tab). This was shown in the previous section so that we do not copy the screen shot again. As can be seen in this window, two kinds of dependencies can be established: machine dependency in the upper part and context dependencies in the lower part.

In this section, we only speak of context dependencies (machine dependency will be covered in section 3.8). It corresponds to the "sees" relationship alluded in section 1. In the lower editing area, you can select some contexts "seen" by the current machine.

3.2 Variables

3.2.1 Variables Creation Wizard.

In order to activate the variables creation wizard, you have to press the corresponding button in the toolbar as indicated below:



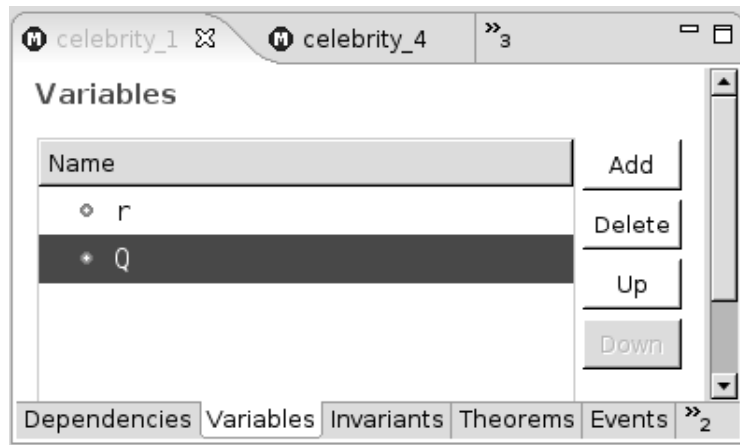
After pressing that button, the following wizard pops up:

A screenshot of a dialog box titled 'New Variable'. It contains three rows of input fields: 'Identifier' with 'var1', 'Initialisation' with 'act2' and 'var1 =', and 'Invariant' with 'inv2' and 'var1 ∈'. At the bottom, there are four buttons: 'Add', 'More Inv.', 'OK', and 'Cancel'. The 'OK' button is highlighted with an orange border.

You can then enter the names of the variables, its initialization, and an invariant which can be used to define its type. By pressing button "More Inv." you can enter additional invariants. For adding more variables, press the "Add" button. When you're finished, press the "OK" button.

3.2.2 Direct Editing of Variables.

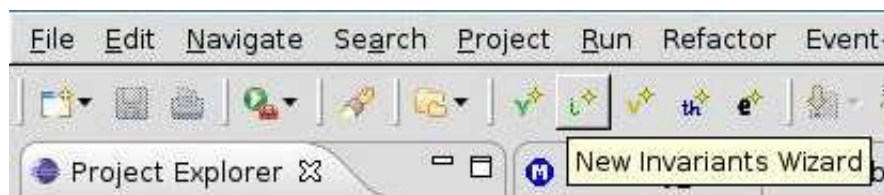
It is also possible to create (button "Add") or remove (button "Delete") variables by using the central editing window. For this, you have first to select the "Variables" tab of the editor. You can also change the relative place of a variable: first select it and then press button "Up" or "Down".



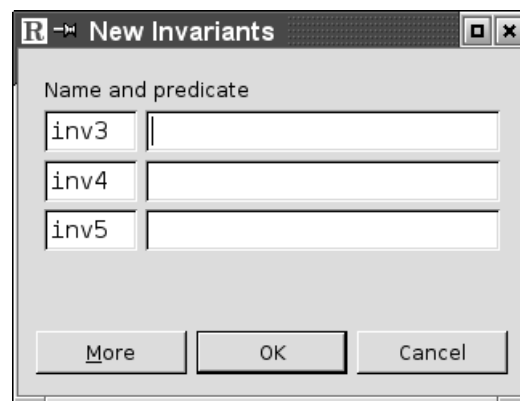
3.3 Invariants

3.3.1 Invariants Creation Wizard.

In order to activate the invariants creation wizard, you have to press the corresponding button:



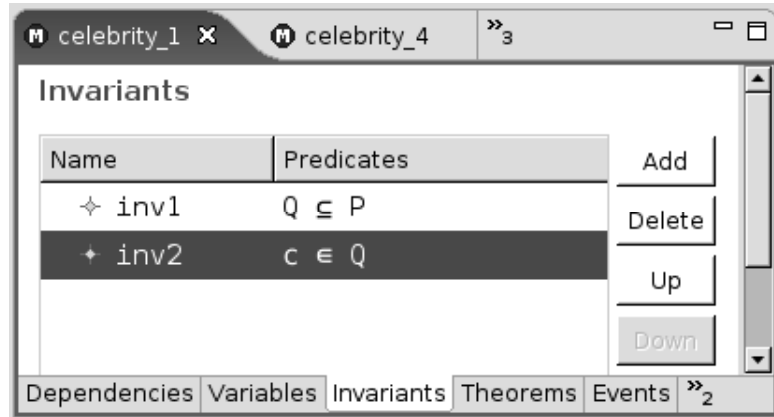
After pressing that button, the following wizard pops up:



You can then enter the invariants you want. If more invariants are needed then press "More".

3.3.2 Direct Editing of Invariants.

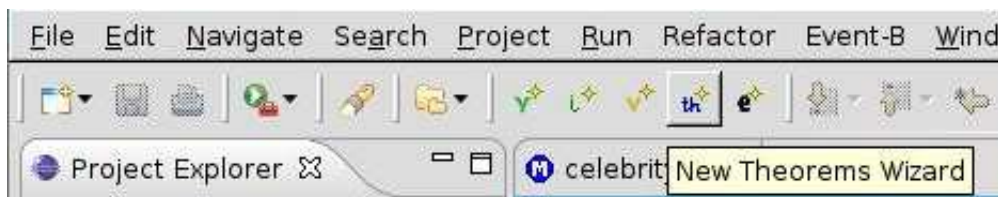
It is also possible to create (button "Add") or remove (button "Delete") invariants by using the central editing window. For this, you have first to select the "Invariants" tab of the editor. You can also change the relative place of an invariant: first select it and then press button "Up" or "Down".



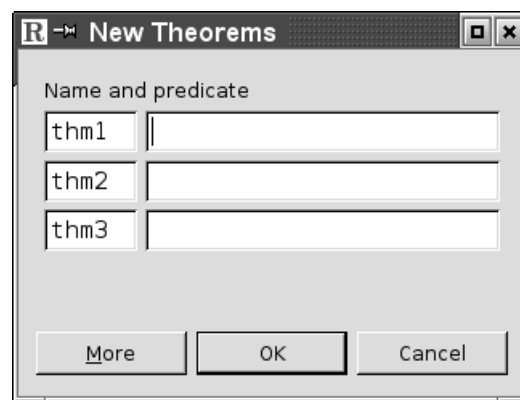
3.4 Theorems

3.4.1 Theorems Creation Wizard.

In order to activate the theorems creation wizard, you have to press the corresponding button:



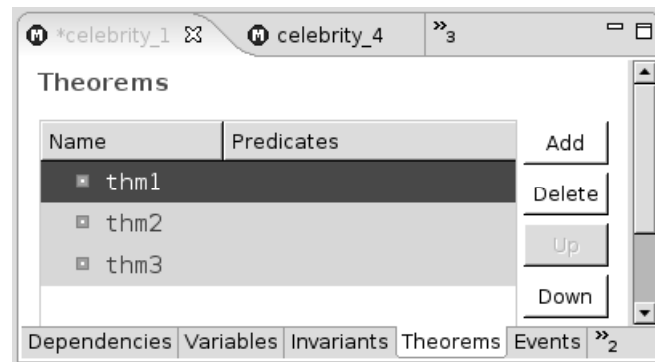
After pressing that button, the following wizard pops up:



You can then enter the theorems you want. If more theorems are needed then press "More". When you are finished, press the "OK" button.

3.4.2 Direct Editing of Theorems.

It is also possible to create (button "Add") or remove (button "Delete") theorems by using the central editing window. For this, you have first to select the "Theorems" tab of the editor. You can also change the relative place of a theorem: first select it and then press button "Up" or "Down".



3.5 Events

3.5.1 Events Creation Wizard.

In order to activate the events creation wizard, you have to press the corresponding button in the toolbar as indicated below:



After pressing that button, the following wizard pops up:

The 'New Events' dialog box contains the following fields and buttons:

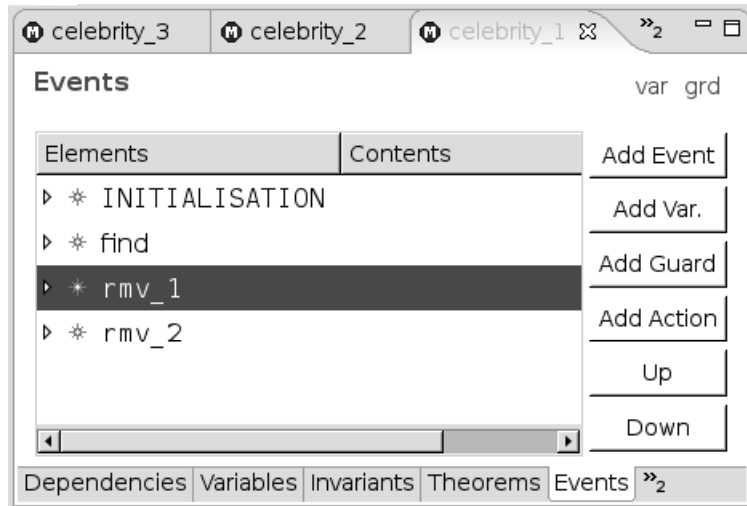
- Label:** A text field containing 'evt1'.
- Parameter identifier(s):** Three empty text fields.
- Guard label(s):** Three text fields containing 'grd1', 'grd2', and 'grd3'.
- Guard predicate(s):** Three empty text fields.
- Action label(s):** Three text fields containing 'act1', 'act2', and 'act3'.
- Action substitution(s):** Three empty text fields.
- Buttons:** 'Add', 'More Par.', 'More Grd.', 'More Act.', 'OK' (highlighted), and 'Cancel'.

You can then enter the events you want. As indicated, the following elements can be entered: name, parameters, guards, and actions. More parameters, guards and actions can be entered by pressing the corresponding buttons. If more events are needed then press "Add". Press "OK" when you're finished.

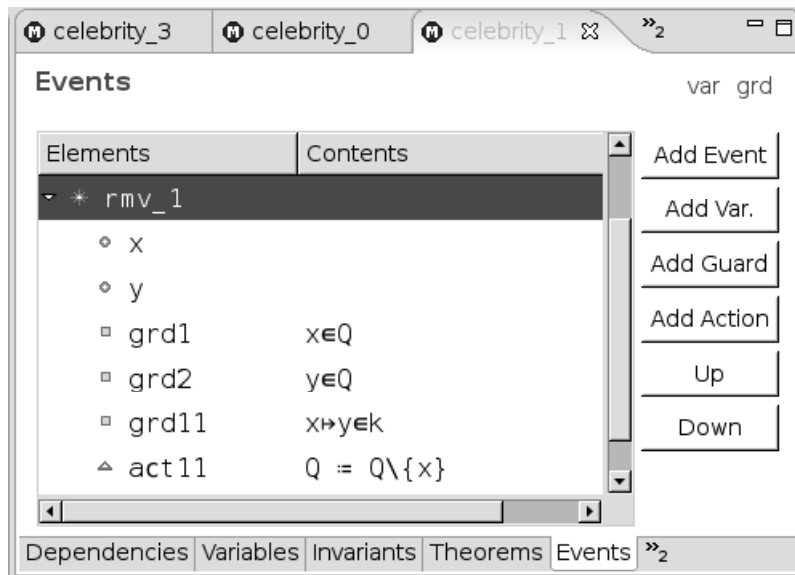
Note that an event with no guard is considered to have a true guard. Moreover, an event with no action is considered to have the "skip" action.

3.5.2 Direct Editing of Events.

It is also possible to perform a direct creation (button "Add Event") of variables by using the central editing window. For this, you have first to select the "Events" tab of the editor. You can also change the relative place of a variable: first select it and then press button "Up" or "Down".



Once an event is selected you can add parameters, guards, and actions. The components of an events can be seen by pressing the little triangle situated on the left of the event name:



As can be seen, event `rmv_1` is made of two parameters, x and y , three guards, $x \in Q$, $y \in Q$, and $x \mapsto y \in k$, and one action $Q := Q \setminus \{x\}$. These elements can be modified (select and edit) or removed (select, right click on the mouse, and press "Delete"). Similar elements can be added by pressing the relevant buttons on the right of the window.

3.6 Adding Comments

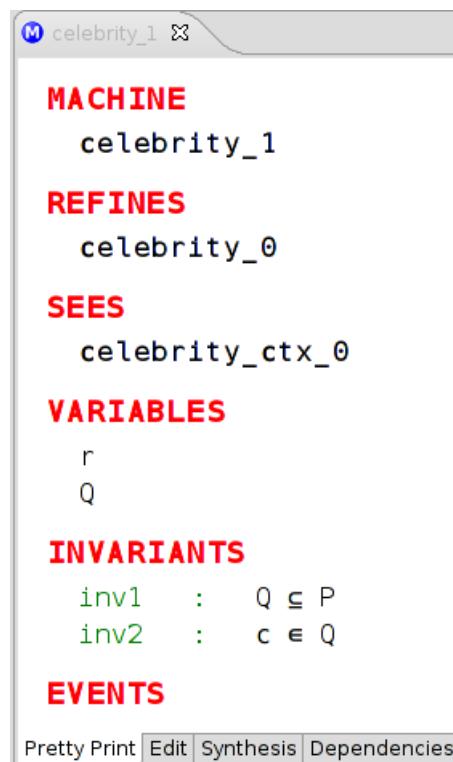
It is possible to add comments to variables, invariants, theorems, events, guards, and actions. For doing so, select the corresponding modeling element and enter the "Properties" window as indicated below where it is shown how one can add comments on a certain guard:



Multiline comments can be added in the editing area labeled "Comments".

3.7 Pretty Print

The pretty print of a machine looks like an input file. It is produced as an output of the editing process:



```
MACHINE
  celebrity_1

REFINES
  celebrity_0

SEES
  celebrity_ctx_0

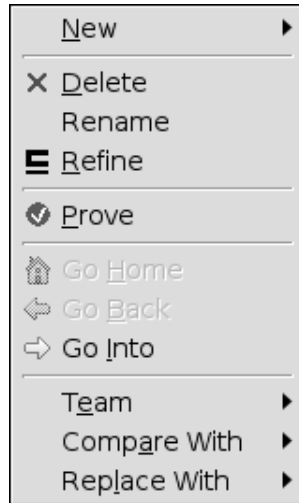
VARIABLES
  r
  Q

INVARIANTS
  inv1   :   Q ⊆ P
  inv2   :   c ∈ Q

EVENTS
```

3.8 Dependencies: Refining a Machine

A machine can be refined by other ones. This can be done directly by selecting the machine to be refined in the "Project Explorer" window. A right click on the mouse yields the following contextual menu:

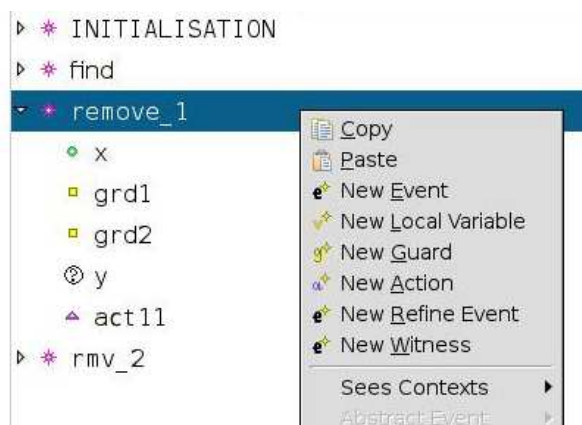


You then press button "Refine". A wizard will ask you to enter the name of the refined machine. The abstract machine is entirely copied in the refined machine: this is very convenient as, quite often, the refined machine has lots of elements in common with its abstraction.

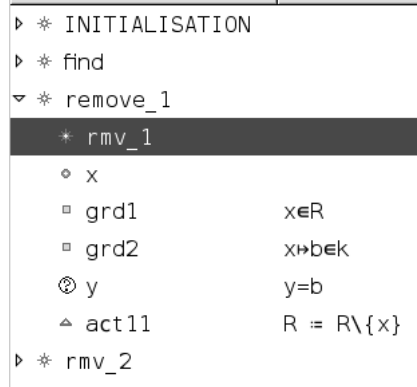
3.9 Adding more Dependencies

3.9.1 Abstract Event

The abstraction of an event is denoted by a "Refine Event" element. Most of the time the concrete and abstract events bear the same name. But, it is always possible to change the name of a concrete event or the name of its abstraction. If you want to specify the abstraction of an event, first select the "Event" tab of the editor and right click on the event name. The following contextual menu will pop up:



You have then to choose the "New Refine Event" option. The abstract event can then be entered by adding the name of the abstract event: here `rmv_1`.



3.9.2 Splitting an Event

An abstract event can be split into two or more concrete events by just saying that these events refine the former (as explained in previous section).

3.9.3 Merging Events

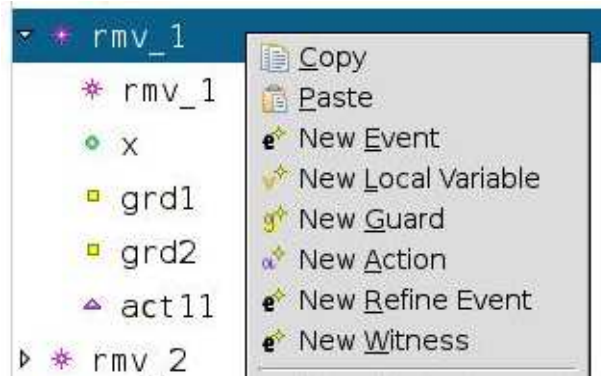
Two or more abstract events can be merged into a single concrete event by saying that the latter refines all the former. This is done by using several times the approach explained in the previous case. The constraints is that the abstract events to be merged must have exactly the same actions (including the labels of these actions). A proof obligation is generated which states that the guard of the concrete event implies the disjunction of the guards of the abstract events

3.9.4 Witnesses

When an abstract event contains some parameters, the refinement proof obligation involves proving an existentially quantified statement. In order to simplify the proof, the user is required to give witnesses for those abstract parameters which are not present in the refinement (those appearing in the refinement are implicitly taken as witnesses for their corresponding abstract counterparts). Here is an example of an abstract event (left) and its refinement (right):

<pre> rmv_1 ANY x y WHERE grd1: x ∈ Q grd2: y ∈ Q grd11: x ↦ y ∈ k THEN act11: Q := Q \ {x} END </pre>	<pre> rmv_1 REFINES rmv_1 ANY x WHERE grd1: x ∈ R grd2: x ↦ b ∈ k THEN act11: R := R \ {x} END </pre>
--	---

The parameter x , being common to both the abstraction and the refinement, does not require a witness, whereas one is needed for abstract parameter y . In order to define the witness, one has first to select the "Events" tab of the editor for the concrete machine where the concrete event (here `rmv_1`) is selected. After a right click, a menu appears in the window as indicated:



You press button "New Witness" and then you enter the parameter name (here y) and a predicate involving y (here $y = b$) as indicated below

* rmv_1	
* rmv_1	
◇ x	
▣ grd1	$x \in R$
▣ grd2	$x \mapsto b \in k$
⊙ y	$y = b$
△ act11	$R := R \setminus \{x\}$

Most of the time, the predicate is an equality as in the previous example, meaning that the parameter is defined in a deterministic way. But it can also be any predicate, in which case the parameter is defined in a non-deterministic way.

3.9.5 Variant

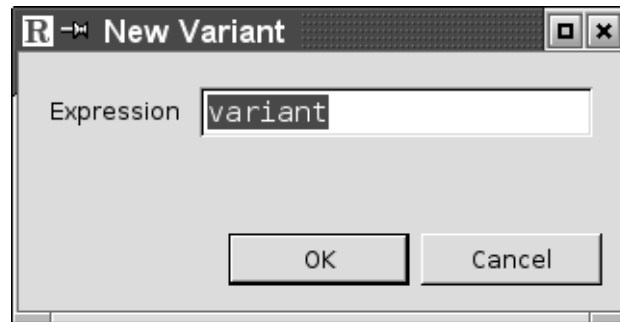
New events can be defined in a concrete machine. Such events have no abstract counterparts. They must refine the implicit "empty" abstract event which does nothing.

Some of the new events can be selected to decrease a variant so that they do not take control for ever. Such events are said to be CONVERGENT. In order to make a new event CONVERGENT, select it in the "Events" tab and open the "Properties" window. You can edit the "conv." area. There are three options: ORDINARY (the default), CONVERGENT, or ANTICIPATED. The latter corresponds to a new event which is not yet declared to be CONVERGENT but will be in a subsequent refinement.

In order to define the variant, use the variant wizard as shown below:



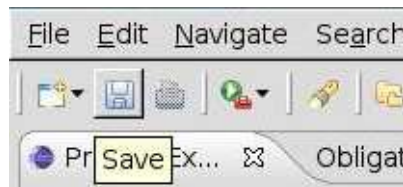
After pressing that button, the following wizard will pop up



You can enter the variant and then press "OK". The variant is either a natural number expression or a finite set expression.

4 Saving a Context or a Machine

Once a machine or context is (possibly partly only) edited, you can save it by using the following button:



4.1 Automatic Tool Invocations

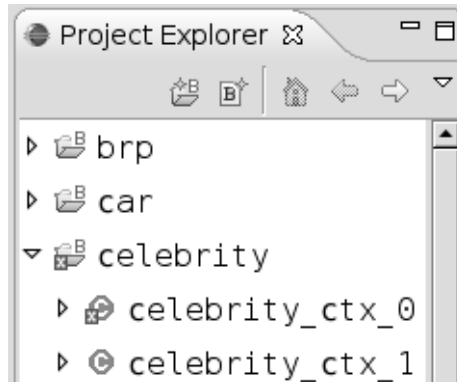
Once a "Save" is done, three tools are called automatically, these are:

- the Static Checker,
- the Proof Obligation Generator,
- the Auto-Prover.

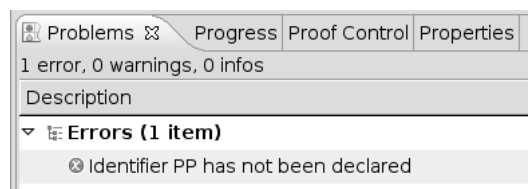
This can take a certain time. A "Progress" window can be opened at the bottom right of the screen to see which tools are working (most of the time, it is the auto-prover).

4.2 Errors. The Problems Window

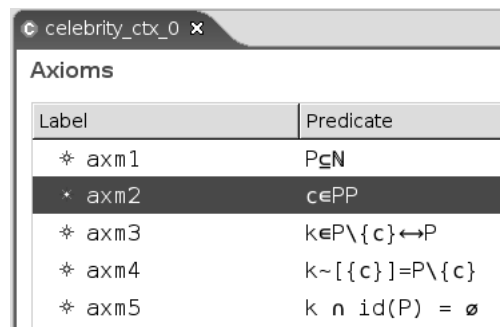
When the Static Checker discovers an error in a project, a little "x" is added to this project and to the faulty component in the "Project Explorer" window as shown in the following screen shot:



The error itself is shown by opening the "Problems" window.

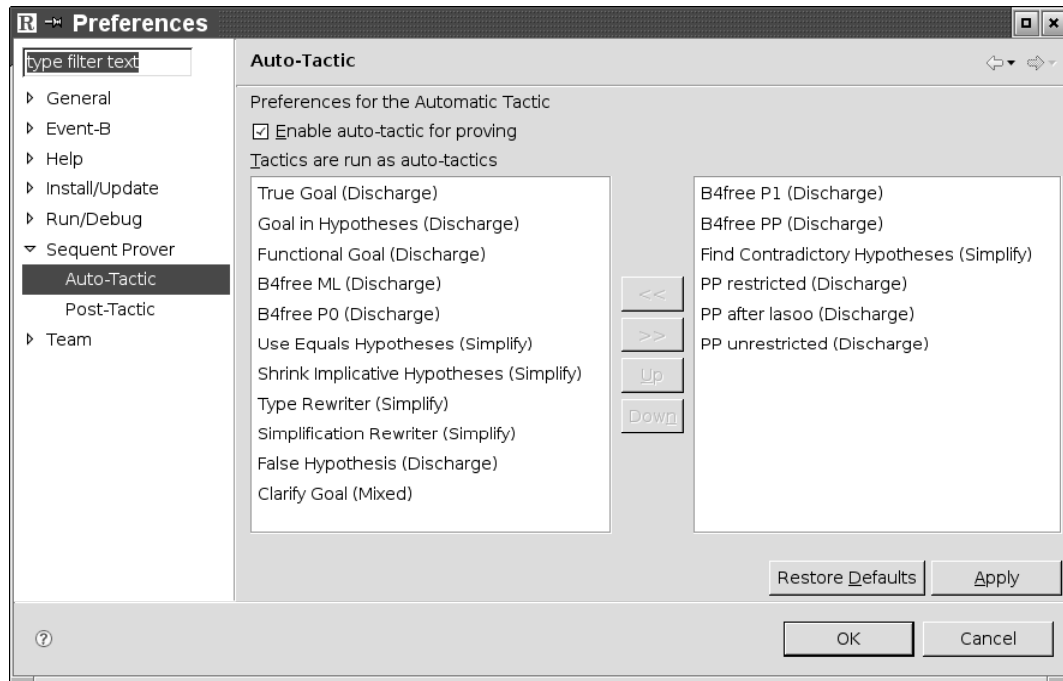


By double-clicking on the error statement, you are transferred automatically into the place where the error has been detected so that you can correct it easily as shown below:



4.3 Preferences for the Auto-prover

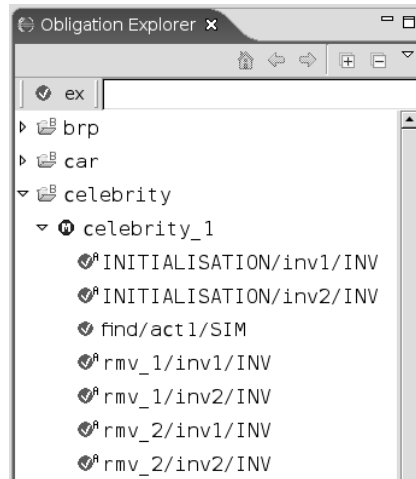
The auto-prover can be configured by means of a preference page, which can be obtained as follows: press the "Window" button on the top toolbar. On the coming menu, press the "Preferences" button. On the coming menu, press the "Event-B" menu, then the "Sequent Prover", and finally the "Auto-Tactic" button. This yields the following window:



On the left part you can see the ordered sequence of individual tactics composing the auto-prover, whereas the right part contains further tactics you can incorporate in the left part. By selecting a tactic you can move it from one part to the other or change the order in the left part.

5 The Proof Obligation Explorer

The "Proof Obligation Explorer" window has the same main entries as the "Project Explorer" window, namely the projects and its components (context and machines). When expanding a component in the Proof Obligation Explorer you get a list of its proof obligations as generated automatically by the Proof Obligation Generator:



As can be seen in this screen shot, machine "celebrity_1" of project "celebrity" is expanded. We find seven proof obligations. Each of them has got a compound name as indicated in the tables below. A green logo situated on the left of the proof obligation name states that it has been proved (an A means it has been proved automatically). By clicking on the proof obligation name, you are transferred into the Proving Perspective which we are going to describe in subsequent sections.

Next is a table describing the names of context proof obligations:

Well-definedness of an Axiom	axm / WD	axm is the axiom name
Well-definedness of a Theorem	thm / WD	thm is the theorem name
Theorem	thm / THM	thm is the theorem name

Next is a table showing the name of machine proof obligations:

Well-definedness of an Invariant	inv / WD	inv is the invariant name
Well-definedness of a Theorem	thm / WD	thm is the theorem name
Well-definedness of an event Guard	evt / grd / WD	evt is the event name grd is the guard name
Well-definedness of an event Action	evt / act / WD	evt is the event name act is the action name
Feasibility of a non-det. event Action	evt / act / FIS	evt is the event name act is the action name
Theorem	thm / THM	thm is the theorem name
Invariant Establishment	INIT. / inv / INV	inv is the invariant name
Invariant Preservation	evt / inv / INV	evt is the event name inv is the invariant name

Next are the proof obligations concerned with machine refinements:

Guard Strengthening	$evt / grd / GRD$	evt is the concrete event name grd is the abstract guard name
Guard Strengthening (merge)	evt / MRG	evt is the concrete event name
Action Simulation	$evt / act / SIM$	evt is the concrete event name act is the abstract action name
Equality of a preserved Variable	$evt / v / EQL$	evt is the concrete event name v is the preserved variable

Next are the proof obligations concerned with the new events variant:

Well definedness of Variant	VWD	
Finiteness for a set Variant	FIN	
Natural number for a numeric Variant	evt / NAT	evt is the new event name
Decreasing of Variant	evt / VAR	evt is the new event name

Finally, here are the proof obligations concerned with witnesses:

Well definedness of Witness	$evt / p / WWD$	evt is the concrete event name p is parameter name or a primed variable name
Feasibility of non-det. Witness	$evt / p / WFIS$	evt is the concrete event name p is parameter name or a primed variable name

Remark: At the moment, the deadlock freeness proof obligation generation is missing. If you need it, you can generate it yourself as a theorem saying the the disjunction of the abstract guards imply the disjunction of the concrete guards.

6 The Proving Perspective

The Proving Perspective is made of a number of windows: the proof tree, the goal, the selected hypotheses, the proof control, the proof information, and the searched hypotheses. In subsequent sections, we study these windows, but before that let us see how one can load a proof.

6.1 Loading a Proof

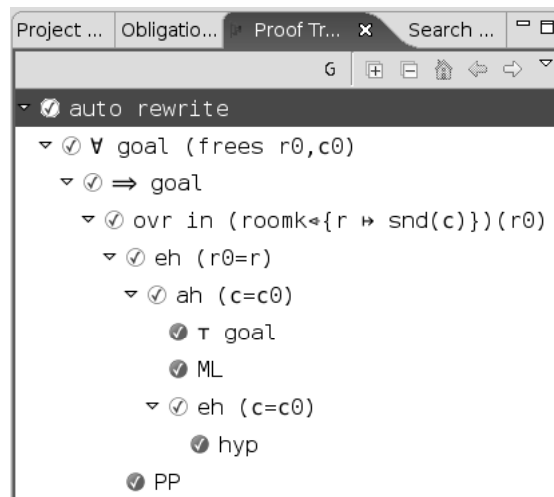
In order to load a proof, enter the Proof Obligation window, select the project, select and expand the component, finally select the proof obligation: the corresponding proof will be loaded. As a consequence:

- the proof tree is loaded in the Proof Tree window. As we shall see in section 6.2, each node of the proof tree is associated with a sequent.
- In case the proof tree has some "pending" nodes (whose sequents are not discharged yet) then the sequent corresponding to the first pending node is decomposed: its goal is loaded in the Goal window (section 6.3), whereas parts of its hypotheses (the "selected" ones) are loaded in the Selected Hypotheses window (section 6.3).
- In case the proof tree has no pending node, then the sequent of the root node is loaded as explained previously.

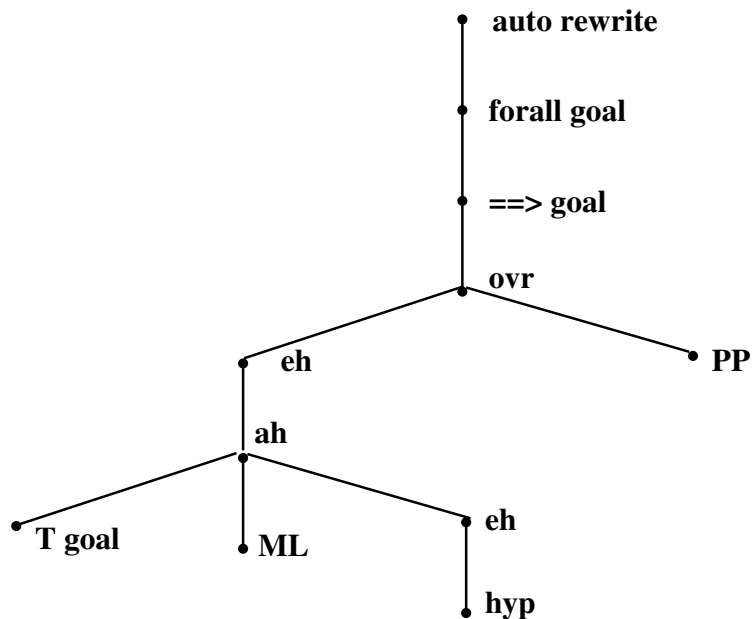
6.2 The Proof Tree

6.2.1 Description of the Proof Tree

The proof tree can be seen in the corresponding window as shown in the following screen shot:



Each line in the proof tree corresponds to a node which is a sequent. A line is right shifted when the corresponding node is a direct descendant of the node of the previous line. Here is an illustration of the previous tree:



Each node is labelled with a comment explaining how it can be discharged. By selecting a node in the proof tree, the corresponding sequent is decomposed and loaded in the Goal and Selected Hypotheses windows as explained in section 6.1.

6.2.2 Decoration

The leaves of the tree are decorated with three kinds of logos:

- a green logo with a "√" in it means that this leaf is discharged,
- a red logo with a "?" in it means that this leaf is not discharged,
- a blue logo with a "R" in it means that this leaf has been reviewed.

Internal nodes in the proof tree are decorated in the same (but lighter) way. Note that a "reviewed" leaf is one that is not discharged yet by the prover. Instead, it has been "seen" by the user who decided to have it discharged later. Marking nodes as "reviewed" is very convenient in order to perform an interactive proof in a gradual fashion. In order to discharge a "reviewed" node, select it and prune the tree at that node (section 6.2.5): the node will become "red" again (undischarged) and you can now try to discharge it.

6.2.3 Navigation within the Proof Tree

On top of the proof tree window one can see three buttons:

- the "G" buttons allows you to see the goal of the sequent corresponding to the node

- the "+" button allows you to fully expand the proof tree
- the "-" allows you to fully collapse the tree: only the root stays visible.

6.2.4 Hiding

The little triangle next to each node in the proof tree allows you to expand or collapse the subtree starting at that node.

6.2.5 Pruning

The proof tree can be pruned from a node: it means that the subtree starting at that node is eliminated. The node in question becomes a leaf and is red decorated. This allows you to resume the proof from that node. After selecting a sequent in the proof tree, pruning can be performed in two ways:

- by right-clicking and then selecting "Prune",
- by pressing the "Scissors" button in the proof control window (section 6.4).

Note that after pruning, the post-tactic (section 6.8) is not applied to the new current sequent: if needed you have to press the "post-tactic" button in the Proof Control window (section 6.4). This happens in particular when you want to redo a proof from the beginning: you prune the proof tree from the root node and then you have to press the "post-tactic" button in order to be in exactly the same situation as the one delivered automatically initially.

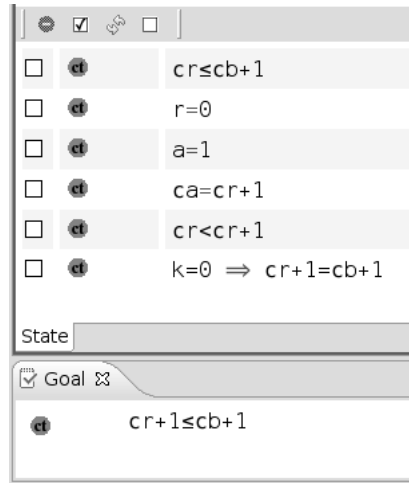
When you want to redo a proof from a certain node, it might be advisable to do it after copying the tree so that in case your new proof fails you can still resume the previous situation by pasting the copied version (see next section).

6.2.6 Copy/Paste

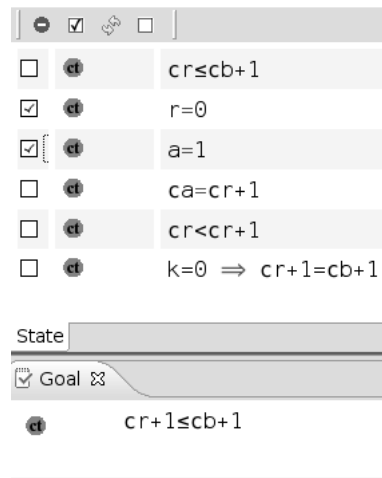
By selecting a node in the proof tree and then clicking on the right key of the mouse, you can copy the part of the proof tree starting at that sequent: it can later be pasted in the same way. This allows you to reuse part of a proof tree in the same (or even another) proof.

6.3 Goal and Selected Hypotheses

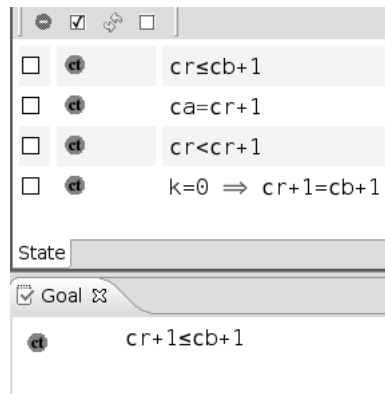
The "Goal" and "Selected Hypotheses" windows display the current sequent you have to prove at a given moment in the proof. Here is an example:



A selected hypothesis can be deselected by first clicking in the box situated next to it (you can click on several boxes) and then by pressing the red (-) button at the top of the selected hypothesis window:



Here is the result:



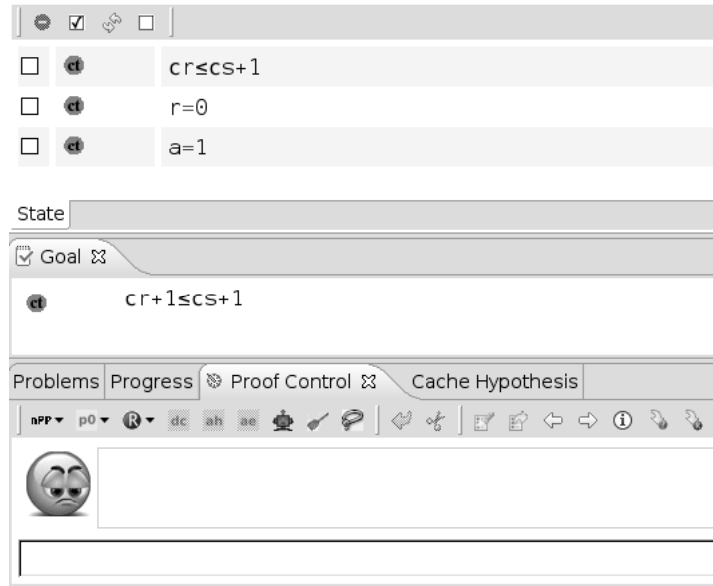
Notice that the deselected hypotheses are not lost: you can get them back by means of the Searched Hypotheses window (section 6.7).

The three other buttons next to the red (-) button allow you to do the reverse operation, namely keeping some hypotheses. The (ct) button next to the goal allows you to do a proof by contradiction: pressing it makes the negation of the goal being a selected hypothesis whereas the goal becomes "false". The (ct) button next to a selected hypothesis allows you to do another kind of proof by contradiction: pressing it makes the negation of the hypothesis the goal whereas the negated goal becomes an hypothesis.

6.4 The Proof Control Window

The Proof Control window contains the buttons which you can use to perform an interactive proof. Next is a screen shot where you can see successively from top to bottom:

- some selected hypotheses,
- the goal,
- the "Proof Control" window,
- a small editing area within which you can enter parameters used by some buttons of the Proof Control window
- the smiley (section 6.5)



The Proof Control window offers a number of buttons which we succinctly describe from left to right:

- (nPP): the prover newPP attempts to prove the goal (other cases in the list),
- (p0): the prover PP attempts to prove the goal (other cases in the list)
- (R) review: the user forces the current sequent to be discharged, it is marked as being reviewed (it's logo is blue-colored)
- (dc) proof by cases: the goal is proved first under the predicate written in the editing area and then under its negation,
- (ah) lemma: the predicate in the editing area is proved and then added as a new selected hypothesis,
- (ae) abstract expression: the expression in the editing area is given a fresh name,
- the auto-prover attempts to discharge the goal. The auto-prover is the one which is applied automatically on all proof obligations (as generated automatically by the proof obligation generator after a "save") without any intervention of the user. With this button, you can call yourself the auto-prover within an interactive proof.
- the post-tactic is executed (see section 6.8),
- lasso: load in the Selected Hypotheses window those unseen hypotheses containing identifiers which are common with identifiers in the goal and selected hypotheses,
- backtrack from the current node (i.e., prune its parent),
- scissors: prune the proof tree from the node selected in the proof tree,
- show (in the Search Hypotheses window) hypotheses containing the character string as in the editing area,
- show the Cache Hypotheses window,
- load the previous non-discharged proof obligation,

- load the next non-discharged proof obligation,
- (i) show information corresponding to the current proof obligation in the corresponding window. This information correspond to the elements that took directly part in the proof obligation generation (events, invariant, etc.),
- goto the next pending node of the current proof tree,
- load the next reviewed node of the current proof tree.

6.5 The Smiley

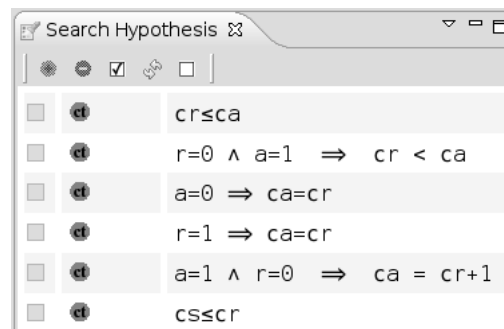
The smiley can take three different colors: (1) red, meaning that the proof tree contains one or more non-discharged sequents, (2) blue, meaning that all non-discharged sequents of the proof tree have been reviewed, (3) green, meaning that all sequents of the proof tree are discharged.

6.6 The Operator "Buttons"

In the goal and in the selected, searched, or cache hypotheses some operators are colored in red. It means that they are "buttons" you can press. When doing so, the meaning (sometimes several) is shown in a menu where you can select various options. The operation performed by these options is described in sections 6.9.1 and 6.9.2.

6.7 The Search Hypotheses Window

A window is provided which contains the hypotheses having a character string in common with the one in the editing area. For example, if we search for hypotheses involving the character string "cr", then after pressing the "search hypothesis" button in the proof control window, we obtain the following:



Such hypotheses can be moved to the "Selected Hypotheses" window (button (+)) or removed from the "Search Hypotheses" window (button (-)). As for the selected hypotheses, other buttons situated next to the previous ones, allow you to move or remove all hypotheses. By pressing the (ct) button the negation of the corresponding hypothesis becomes the new goal.

6.8 The Automatic Post-tactic

In this section, we present the various rewrite or inference rules which are applied automatically as a systematic post-tactic after each proof step. Note that the post-tactic can be disabled by using the "P" button situated on the right of the proof control window.

The post-tactic is made of two different rules: rewrite rules, which are applied on any sub-formula of the goal or selected hypotheses (section 6.8.1) and inference rules which are applied on the current sequent (section 6.8.2).

6.8.1 Rewrite rules

The following rewrite rules are *applied automatically* in a systematic fashion from left to right either in the goal or in the selected hypotheses. They all correspond to predicate logical equivalences or expression equalities and result in simplifications. They are sorted according to their main purpose.

Conjunction

$$P \wedge \dots \wedge \top \wedge \dots \wedge Q == P \wedge \dots \wedge Q$$

$$P \wedge \dots \wedge \perp \wedge \dots \wedge Q == \perp$$

$$P \wedge \dots \wedge Q \wedge \dots \wedge \neg Q \wedge \dots \wedge R == \perp$$

$$P \wedge \dots \wedge Q \wedge \dots \wedge Q \wedge \dots \wedge R == P \wedge \dots \wedge Q \wedge \dots \wedge R$$

Disjunction

$$P \vee \dots \vee \top \vee \dots \vee Q == \top$$

$$P \vee \dots \vee \perp \vee \dots \vee Q == P \vee \dots \vee Q$$

$$P \vee \dots \vee Q \vee \dots \vee \neg Q \vee \dots \vee R == \top$$

$$P \vee \dots \vee Q \vee \dots \vee Q \vee \dots \vee R == P \vee \dots \vee Q \vee \dots \vee R$$

Implication

$$\top \Rightarrow P == P$$

$$\perp \Rightarrow P == \top$$

$$P \Rightarrow \top == \top$$

$$P \Rightarrow \perp == \neg P$$

$$P \Rightarrow P == \top$$

Equivalence

$$P \Leftrightarrow \top \quad == \quad P$$

$$\top \Leftrightarrow P \quad == \quad P$$

$$P \Leftrightarrow \perp \quad == \quad \neg P$$

$$\perp \Leftrightarrow P \quad == \quad \neg P$$

$$P \Leftrightarrow P \quad == \quad \top$$

Negation

$$\neg \top \quad == \quad \perp$$

$$\neg \perp \quad == \quad \top$$

$$\neg \neg P \quad == \quad P$$

$$E \neq F \quad == \quad \neg E = F$$

$$E \notin F \quad == \quad \neg E \in F$$

$$E \not\subset F \quad == \quad \neg E \subset F$$

$$E \not\subseteq F \quad == \quad \neg E \subseteq F$$

$$\neg a \leq b \quad == \quad a > b$$

$$\neg a \geq b \quad == \quad a < b$$

$$\neg a > b \quad == \quad a \leq b$$

$$\neg a < b \quad == \quad a \geq b$$

$$\neg (E = \text{FALSE}) \quad == \quad (E = \text{TRUE})$$

$$\neg (E = \text{TRUE}) \quad == \quad (E = \text{FALSE})$$

$$\neg (\text{FALSE} = E) \quad == \quad (\text{TRUE} = E)$$

$$\neg (\text{TRUE} = E) \quad == \quad (\text{FALSE} = E)$$

Quantification

$$\forall x \cdot (P \wedge Q) \quad == \quad (\forall x \cdot P) \wedge (\forall x \cdot Q)$$

$$\exists x \cdot (P \vee Q) \quad == \quad (\exists x \cdot P) \vee (\exists x \cdot Q)$$

$$\forall \dots, z, \dots \cdot P(z) \quad == \quad \forall z \cdot P(z)$$

$$\exists \dots, z, \dots \cdot P(z) \quad == \quad \exists z \cdot P(z)$$

Equality

$$E = E \implies \top$$

$$E \neq E \implies \perp$$

$$E \mapsto F = G \mapsto H \implies E = G \wedge F = H$$

$$\text{TRUE} = \text{FALSE} \implies \perp$$

$$\text{FALSE} = \text{TRUE} \implies \perp$$

Set Theory

$$S \cap \dots \cap \emptyset \cap \dots \cap T \implies \emptyset$$

$$S \cap \dots \cap T \cap \dots \cap T \cap \dots \cap U \implies S \cap \dots \cap T \cap \dots \cap U$$

$$S \cup \dots \cup \emptyset \cup \dots \cup T \implies S \cup \dots \cup T$$

$$S \cup \dots \cup T \cup \dots \cup T \cup \dots \cup U \implies S \cup \dots \cup T \cup \dots \cup U$$

$$\emptyset \subseteq S \implies \top$$

$$S \subseteq S \implies \top$$

$$E \in \emptyset \implies \perp$$

$$B \in \{A, \dots, B, \dots, C\} \implies \top$$

$$\{A, \dots, B, \dots, B, \dots, C\} \implies \{A, \dots, B, \dots, C\}$$

$$E \in \{x \mid P(x)\} \implies P(E)$$

$$S \setminus S \implies \emptyset$$

$$S \setminus \emptyset \implies S$$

$$\emptyset \setminus S \implies \emptyset$$

$$r[\emptyset] \implies \emptyset$$

$$\text{dom}(\emptyset) \implies \emptyset$$

$$\text{ran}(\emptyset) \implies \emptyset$$

$$r^{-1-1} \implies r$$

$$\text{dom}(\{x \mapsto a, \dots, y \mapsto b\}) \implies \{x, \dots, y\}$$

$$\text{ran}(\{x \mapsto a, \dots, y \mapsto b\}) \implies \{a, \dots, b\}$$

$$\begin{aligned}
(f \Leftarrow \dots \Leftarrow \{E \mapsto F\})(E) &== F \\
E \in \{F\} &== E = F \quad \text{where } F \text{ is a single expression} \\
(S \times \{F\})(x) &== F \quad \text{where } F \text{ is a single expression} \\
\{E\} = \{F\} &== E = F \quad \text{where } E \text{ and } F \text{ are single expressions} \\
\{x \mapsto a, \dots, y \mapsto b\}^{-1} &== \{a \mapsto x, \dots, b \mapsto y\} \\
Ty = \emptyset &== \perp \\
\emptyset = Ty &== \perp \\
t \in Ty &== \top
\end{aligned}$$

In the three previous rewrite rules, Ty denotes a type expression, that is either a basic type (\mathbb{Z} , BOOL , any carrier set), or $\mathbb{P}(\text{type expression})$, or type expression \times type expression, or type expression \leftrightarrow type expression, and t denotes an expression of type Ty .

$$\begin{aligned}
U \setminus U \setminus S &== S \\
S \cup \dots \cup U \cup \dots \cup T &== U \\
S \cap \dots \cap U \cap \dots \cap T &== S \cap \dots \cap T \\
S \setminus U &== \emptyset
\end{aligned}$$

In the four previous rules, S and T are supposed to be of type $\mathbb{P}(U)$ (U thus takes the rôle of a universal set for S and T).

$$\begin{aligned}
r ; \emptyset &== \emptyset \\
\emptyset ; r &== \emptyset \\
f(f^{-1}(E)) &== E \\
f^{-1}(f(E)) &== E \\
S \subseteq A \cup \dots \cup S \cup \dots \cup B &== \top \\
A \cap \dots \cap S \cap \dots \cap B \subseteq S &== \top \\
A \cup \dots \cup B \subseteq S &== A \subseteq S \wedge \dots \wedge B \subseteq S \\
S \subseteq A \cap \dots \cap B &== S \subseteq A \wedge \dots \wedge S \subseteq B \\
A \cup \dots \cup B \subset S &== A \subset S \wedge \dots \wedge B \subset S \\
S \subset A \cap \dots \cap B &== S \subset A \wedge \dots \wedge S \subset B \\
A \setminus B \subseteq S &== A \subseteq B \cup S
\end{aligned}$$

Finiteness

$$\text{finite}(\emptyset) == \top$$

$$\text{finite}(\{a, \dots, b\}) == \top$$

$$\text{finite}(S \cup T) == \text{finite}(S) \wedge \text{finite}(T)$$

$$\text{finite}(\mathbb{P}(S)) == \text{finite}(S)$$

$$\text{finite}(S \times T) == S = \emptyset \vee T = \emptyset \vee (\text{finite}(S) \wedge \text{finite}(T))$$

$$\text{finite}(r^{-1}) == \text{finite}(r)$$

$$\text{finite}(a .. b) == \top$$

Cardinality

$$\text{card}(\emptyset) == 0$$

$$\text{card}(\{E\}) == 1$$

$$\text{card}(\mathbb{P}(S)) == 2^{\text{card}(S)}$$

$$\text{card}(S \times T) == \text{card}(S) * \text{card}(T)$$

$$\text{card}(S \setminus T) == \text{card}(S) - \text{card}(S \cap T)$$

$$\text{card}(S \cup T) == \text{card}(S) + \text{card}(T) - \text{card}(S \cap T)$$

$$\text{card}(S) = 0 == S = \emptyset$$

$$0 = \text{card}(S) == S = \emptyset$$

$$\neg \text{card}(S) = 0 == \neg S = \emptyset$$

$$\neg 0 = \text{card}(S) == \neg S = \emptyset$$

$$\text{card}(S) > 0 == \neg S = \emptyset$$

$$0 < \text{card}(S) == \neg S = \emptyset$$

$$\text{card}(S) = 1 == \exists x \cdot S = \{x\}$$

$$1 = \text{card}(S) == \exists x \cdot S = \{x\}$$

Arithmetic

$$E + \dots + 0 + \dots + F == E + \dots + F$$

$$E - 0 == E$$

$$0 - E == -E$$

$$-(-E) == E$$

$$E - E == 0$$

$$E * \dots * 1 * \dots * F == E * \dots * F$$

$$E * \dots * 0 * \dots * F == 0$$

$$(-E) * \dots * (-F) == E * \dots * F \quad (\text{if an even number of } -)$$

$$(-E) * \dots * (-F) == -(E * \dots * F) \quad (\text{if an odd number of } -)$$

$$0/E == 0$$

$$(-E)/(-F) == E/F$$

$$E^1 == E$$

$$E^0 == 1$$

$$1^E == 1$$

$$-(i) == (-i) \quad \text{where } i \text{ is a literal}$$

$$-((-i)) == i \quad \text{where } i \text{ is a literal}$$

$$i = j == \top \text{ or } \perp \text{ (computation) where } i \text{ and } j \text{ are literals}$$

$$i \leq j == \top \text{ or } \perp \text{ (computation) where } i \text{ and } j \text{ are literals}$$

$$i < j == \top \text{ or } \perp \text{ (computation) where } i \text{ and } j \text{ are literals}$$

$$i \geq j == \top \text{ or } \perp \text{ (computation) where } i \text{ and } j \text{ are literals}$$

$$i > j == \top \text{ or } \perp \text{ (computation) where } i \text{ and } j \text{ are literals}$$

$$E \leq E == \top$$

$$E < E == \perp$$

$$E \geq E == \top$$

$$E > E == \perp$$

6.8.2 Automatic inference rules

The following inference rules are *applied automatically* in a systematic fashion at the end of each proof step. They have the following possible effects:

- they discharge the goal,
- they simplify the goal and add a selected hypothesis,
- they simplify the goal by decomposing it into several simpler goals,
- they simplify a selected hypothesis,
- they simplify a selected hypothesis by decomposing it into several simpler selected hypotheses.

Axioms

$$\frac{}{\mathbf{H, P \vdash P}} \text{ HYP}$$

$$\frac{}{\mathbf{H, Q \vdash P \vee \dots \vee Q \vee \dots \vee R}} \text{ HYP_OR}$$

$$\frac{}{\mathbf{H, P, \neg P \vdash Q}} \text{ CNTR}$$

$$\frac{}{\mathbf{H, \perp \vdash P}} \text{ FALSE_HYP}$$

$$\frac{}{\mathbf{H \vdash \top}} \text{ TRUE_GOAL}$$

Simplification

$$\frac{\mathbf{H, P \vdash Q}}{\mathbf{H, P, P \vdash Q}} \text{ DBL_HYP}$$

Conjunction

$$\frac{\mathbf{H, P, Q \vdash R}}{\mathbf{H, P \wedge Q \vdash R}} \text{ AND_L}$$

$$\frac{\mathbf{H \vdash P} \quad \mathbf{H \vdash Q}}{\mathbf{H \vdash P \wedge Q}} \text{ AND_R}$$

Implication

$$\frac{\mathbf{H, Q, P \wedge \dots \wedge R \Rightarrow S \vdash T}}{\mathbf{H, Q, P \wedge \dots \wedge Q \wedge \dots \wedge R \Rightarrow S \vdash T}} \text{ IMP_L1}$$

$$\frac{\mathbf{H, P \vdash Q}}{\mathbf{H \vdash P \Rightarrow Q}} \text{ IMP_R}$$

$$\frac{\mathbf{H, P \Rightarrow Q, P \Rightarrow R \vdash S}}{\mathbf{H, P \Rightarrow Q \wedge R \vdash S}} \text{ IMP_AND_L}$$

$$\frac{\mathbf{H, P \Rightarrow R, Q \Rightarrow R \vdash S}}{\mathbf{H, P \vee Q \Rightarrow R \vdash S}} \text{ IMP_OR_L}$$

Negation

$$\frac{\mathbf{H}, E \in \{a, \dots, c\} \vdash \mathbf{P}}{\mathbf{H}, E \in \{a, \dots, b, \dots, c\}, \neg(E = b) \vdash \mathbf{P}}$$

$$\frac{\mathbf{H}, E \in \{a, \dots, c\} \vdash \mathbf{P}}{\mathbf{H}, E \in \{a, \dots, b, \dots, c\}, \neg(b = E) \vdash \mathbf{P}}$$

Quantification

$$\frac{\mathbf{H}, \mathbf{P}(\mathbf{x}) \vdash \mathbf{Q}}{\mathbf{H}, \exists \mathbf{x} \cdot \mathbf{P}(\mathbf{x}) \vdash \mathbf{Q}} \quad \text{XST_L} \quad (\mathbf{x} \text{ not free in } \mathbf{H} \text{ and } \mathbf{Q})$$

$$\frac{\mathbf{H} \vdash \mathbf{P}(\mathbf{x})}{\mathbf{H} \vdash \forall \mathbf{x} \cdot \mathbf{P}(\mathbf{x})} \quad \text{ALL_R} \quad (\mathbf{x} \text{ not free in } \mathbf{H})$$

Equality

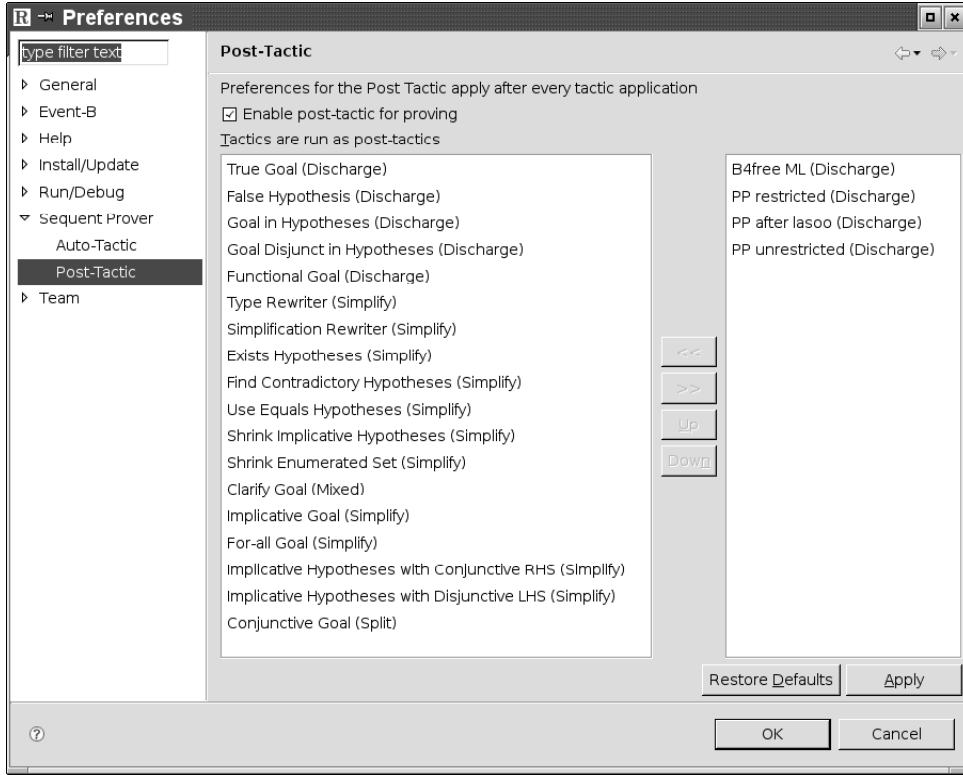
$$\frac{\mathbf{H}(\mathbf{E}) \vdash \mathbf{P}(\mathbf{E})}{\mathbf{H}(\mathbf{x}), x = E \vdash \mathbf{P}(\mathbf{x})} \quad \text{EQL_LR}$$

$$\frac{\mathbf{H}(\mathbf{E}) \vdash \mathbf{P}(\mathbf{E})}{\mathbf{H}(\mathbf{x}), E = x \vdash \mathbf{P}(\mathbf{x})} \quad \text{EQL_RL}$$

In the last two rules x is a variable which is not free in E

6.8.3 Preferences for the Post-tactic

The post-tactic can be configured by means of a preference page which can be obtained as follows: press the "Window" button on the top toolbar. On the coming menu, press the "Preferences" button. On the coming menu, press the "Event-B" menu, then the "Sequent Prover", and finally the "Post-Tactic" button. This yields the following window:



In the left part you can see the ordered sequence of individual tactics composing the post-tactic, whereas the right part contains further tactics you can incorporate in the left part. By selecting a tactic you can move it from one part to the other or change the order in the left part.

6.9 Interactive Tactics

In this section, the rewrite rules and inference rules, which you can use to perform an interactive proof, are presented. Each of these rules can be invoked by pressing "buttons" which corresponds to emphasized (red) operators in the goal or the hypotheses. A menu is proposed when there are several options.

6.9.1 Interactive Rewrite Rules

Most of the rewrite rules correspond to distributive laws. For associative operators in connection with such laws as in:

$$P \wedge (Q \vee \dots \vee R)$$

it has been decided to put the "button" on the first associative/commutative operator (here \vee). Pressing that button will generate a menu: the first option of this menu will be to distribute all associative/commutative operators, the second option will be to distribute only the first associative/commutative operator. In the following presentation, to simplify matters, we write associative/commutative operators with two parameters only, but it must always be understood implicitly that we have a sequence of them. For instance, we shall never write $Q \vee \dots \vee R$ but $Q \vee R$ instead. Rules are sorted according to their purpose.

Conjunction

$$P \wedge (Q \vee R) == (P \wedge Q) \vee (P \wedge R)$$

Disjunction

$$P \vee (Q \wedge R) == (P \vee Q) \wedge (P \vee R)$$

$$P \vee Q \vee \dots \vee R == \neg P \Rightarrow (Q \vee \dots \vee R)$$

Implication

$$P \Rightarrow Q == \neg Q \Rightarrow \neg P$$

$$P \Rightarrow (Q \Rightarrow R) == P \wedge Q \Rightarrow R$$

$$P \Rightarrow (Q \wedge R) == (P \Rightarrow Q) \wedge (P \Rightarrow R)$$

$$P \vee Q \Rightarrow R == (P \Rightarrow R) \wedge (Q \Rightarrow R)$$

Equivalence

$$P \Leftrightarrow Q == (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

Negation

$$\neg (P \wedge Q) == \neg P \vee \neg Q$$

$$\neg (P \vee Q) == \neg P \wedge \neg Q$$

$$\neg (P \Rightarrow Q) == P \wedge \neg Q$$

$$\neg \forall x \cdot P == \exists x \cdot \neg P$$

$$\neg \exists x \cdot P == \forall x \cdot \neg P$$

$$\neg (S = \emptyset) == \exists x \cdot x \in S$$

Set Theory

Most of the rules are concerned with simplifying set membership predicates.

$$E \mapsto F \in S \times T == E \in S \wedge F \in T$$

$$E \in \mathbb{P}(S) == E \subseteq S$$

$$S \subseteq T == \forall x \cdot (x \in S \Rightarrow x \in T)$$

In the previous rule, x denotes several variables when the type of S and T is a Cartesian product.

$$\begin{aligned}
E \in S \cup T &== E \in S \vee E \in T \\
E \in S \cap T &== E \in S \wedge E \in T \\
E \in S \setminus T &== E \in S \wedge \neg(E \in T) \\
E \in \{A, \dots, B\} &== E = A \vee \dots \vee E = B \\
E \in \text{union}(S) &== \exists s \cdot s \in S \wedge E \in s \\
E \in (\bigcup x \cdot x \in S \wedge P \mid T) &== \exists x \cdot x \in S \wedge P \wedge E \in T \\
E \in \text{inter}(S) &== \forall s \cdot s \in S \Rightarrow E \in s \\
E \in (\bigcap x \cdot x \in S \wedge P \mid T) &== \forall x \cdot x \in S \wedge P \Rightarrow E \in T \\
E \in \text{dom}(r) &== \exists y \cdot E \mapsto y \in r \\
F \in \text{ran}(r) &== \exists x \cdot x \mapsto F \in r \\
E \mapsto F \in r^{-1} &== F \mapsto E \in r \\
E \mapsto F \in S \triangleleft r &== E \in S \wedge E \mapsto F \in r \\
E \mapsto F \in r \triangleright T &== E \mapsto F \in r \wedge F \in T \\
E \mapsto F \in S \triangleleft r &== E \notin S \wedge E \mapsto F \in r \\
E \mapsto F \in r \triangleright T &== E \mapsto F \in r \wedge F \notin T \\
F \in r[w] &== \exists x \cdot x \in w \wedge x \mapsto F \in r \\
E \mapsto F \in (p; q) &== \exists x \cdot E \mapsto x \in p \wedge x \mapsto F \in q \\
p \triangleleft q &== (\text{dom}(q) \triangleleft p) \cup q \\
E \mapsto F \in \text{id}(S) &== E \in S \wedge F = E \\
E \mapsto (F \mapsto G) \in p \otimes q &== E \mapsto F \in p \wedge E \mapsto G \in q \\
(E \mapsto G) \mapsto (F \mapsto H) \in p \parallel q &== E \mapsto F \in p \wedge G \mapsto H \in q \\
r \in S \leftrightarrow T &== r \in S \leftrightarrow T \wedge \text{dom}(r) = S \\
r \in S \leftrightarrow T &== r \in S \leftrightarrow T \wedge \text{ran}(r) = T \\
r \in S \leftrightarrow T &== r \in S \leftrightarrow T \wedge \text{dom}(r) = S \wedge \text{ran}(r) = T \\
f \in S \twoheadrightarrow T &== f \in S \leftrightarrow T \wedge \forall x, y, z \cdot x \mapsto y \in f \wedge x \mapsto z \in f \Rightarrow y = z \\
f \in S \rightarrow T &== f \in S \twoheadrightarrow T \wedge \text{dom}(f) = S \\
f \in S \twoheadrightarrow T &== f \in S \twoheadrightarrow T \wedge f^{-1} \in T \twoheadrightarrow S
\end{aligned}$$

$$\begin{aligned}
f \in S \rightarrow T &== f \in S \leftrightarrow T \wedge \text{dom}(f) = S \\
f \in S \nrightarrow T &== f \in S \leftrightarrow T \wedge \text{ran}(f) = T \\
f \in S \twoheadrightarrow T &== f \in S \nrightarrow T \wedge \text{dom}(f) = S \\
f \in S \twoheadleftarrow T &== f \in S \rightarrow T \wedge \text{ran}(f) = T \\
(p \cup q)^{-1} &== p^{-1} \cup q^{-1} \\
(p \cap q)^{-1} &== p^{-1} \cap q^{-1} \\
(s \triangleleft r)^{-1} &== r^{-1} \triangleright s \\
(s \triangleleft r)^{-1} &== r^{-1} \triangleright s \\
(r \triangleright s)^{-1} &== s \triangleleft r^{-1} \\
(r \triangleright s)^{-1} &== s \triangleleft r^{-1} \\
(p ; q)^{-1} &== q^{-1} ; p^{-1} \\
(s \cup t) \triangleleft r &== (s \triangleleft r) \cup (t \triangleleft r) \\
(s \cap t) \triangleleft r &== (s \triangleleft r) \cap (t \triangleleft r) \\
(s \cup t) \triangleleft r &== (s \triangleleft r) \cap (t \triangleleft r) \\
(s \cap t) \triangleleft r &== (s \triangleleft r) \cup (t \triangleleft r) \\
s \triangleleft (p \cup q) &== (s \triangleleft p) \cup (s \triangleleft q) \\
s \triangleleft (p \cap q) &== (s \triangleleft p) \cap (s \triangleleft q) \\
s \triangleleft (p \cup q) &== (s \triangleleft p) \cup (s \triangleleft q) \\
s \triangleleft (p \cap q) &== (s \triangleleft p) \cap (s \triangleleft q) \\
r \triangleright (s \cup t) &== (r \triangleright s) \cup (r \triangleright t) \\
r \triangleright (s \cap t) &== (r \triangleright s) \cap (r \triangleright t) \\
r \triangleright (s \cup t) &== (r \triangleright s) \cap (r \triangleright t) \\
r \triangleright (s \cap t) &== (r \triangleright s) \cup (r \triangleright t) \\
(p \cup q) \triangleright s &== (p \triangleright s) \cup (q \triangleright s) \\
(p \cap q) \triangleright s &== (p \triangleright s) \cap (q \triangleright s) \\
(p \cup q) \triangleright s &== (p \triangleright s) \cup (q \triangleright s) \\
(p \cap q) \triangleright s &== (p \triangleright s) \cap (q \triangleright s) \\
r[S \cup T] &== r[S] \cup r[T]
\end{aligned}$$

$$\begin{aligned}
(p \cup q)[S] &== p[S] \cup q[S] \\
\text{dom}(p \cup q) &== \text{dom}(p) \cup \text{dom}(q) \\
\text{ran}(p \cup q) &== \text{ran}(p) \cup \text{ran}(q) \\
S \subseteq T &== (U \setminus T) \subseteq (U \setminus S)
\end{aligned}$$

In the previous rule, the type of S and T is $\mathbb{P}(U)$.

$$S = T == S \subseteq T \wedge T \subseteq S$$

In the previous rule, S and T are sets

$$\begin{aligned}
S \subseteq A \setminus B &== S \subseteq A \wedge S \cap B = \emptyset \\
p ; (q \cup r) &== (p ; q) \cup (p ; r) \\
(q \cup r) ; p &== (q ; p) \cup (r ; p) \\
(p ; q)[s] &== q[p[s]] \\
(s \triangleleft p) ; q &== s \triangleleft (p ; q) \\
(s \trianglelefteq p) ; q &== s \trianglelefteq (p ; q) \\
p ; (q \triangleright s) &== (p ; q) \triangleright s \\
p ; (q \trianglerighteq s) &== (p ; q) \trianglerighteq s \\
U \setminus (S \cap T) &== (U \setminus S) \cup (U \setminus T) \\
U \setminus (S \cup T) &== (U \setminus S) \cap (U \setminus T) \\
U \setminus (S \setminus T) &== (U \setminus S) \cup T
\end{aligned}$$

In the three previous rules, S and T are supposed to be of type $\mathbb{P}(U)$.

Cardinality

$$\begin{aligned}
\text{card}(S) \leq \text{card}(T) &== S \subseteq T \\
\text{card}(S) \geq \text{card}(T) &== T \subseteq S \\
\text{card}(S) < \text{card}(T) &== S \subset T \\
\text{card}(S) > \text{card}(T) &== T \subset S \\
\text{card}(S) = \text{card}(T) &== S = T
\end{aligned}$$

In the five previous rules, S and T must be of the same type.

6.9.2 Interactive Inference rules

Disjunction

$$\frac{\mathbf{H}, \mathbf{P} \vdash \mathbf{R} \quad \dots \quad \mathbf{H}, \mathbf{Q} \vdash \mathbf{R}}{\mathbf{H}, \mathbf{P} \vee \dots \vee \mathbf{Q} \vdash \mathbf{R}} \text{ CASE}$$

Implication

$$\frac{\mathbf{H} \vdash \mathbf{P} \quad \mathbf{H}, \mathbf{P}, \mathbf{Q} \vdash \mathbf{R}}{\mathbf{H}, \mathbf{P} \Rightarrow \mathbf{Q} \vdash \mathbf{R}} \text{ MH}$$

$$\frac{\mathbf{H} \vdash \neg \mathbf{Q} \quad \mathbf{H}, \neg \mathbf{Q}, \neg \mathbf{P} \vdash \mathbf{R}}{\mathbf{H}, \mathbf{P} \Rightarrow \mathbf{Q} \vdash \mathbf{R}} \text{ HM}$$

Equivalence

$$\frac{\mathbf{H}(\mathbf{Q}), \mathbf{P} \Leftrightarrow \mathbf{Q} \vdash \mathbf{G}(\mathbf{Q})}{\mathbf{H}(\mathbf{P}), \mathbf{P} \Leftrightarrow \mathbf{Q} \vdash \mathbf{G}(\mathbf{P})} \text{ EQV postponed}$$

Set Theory

$$\frac{\mathbf{H}, G = E, \mathbf{P}(F) \vdash \mathbf{Q} \quad \mathbf{H}, \neg(G = E), \mathbf{P}(f(G)) \vdash \mathbf{Q}}{\mathbf{H}, \mathbf{P}((f \Leftarrow \{E \mapsto F\})(G)) \vdash \mathbf{Q}} \text{ OV_L}$$

$$\frac{\mathbf{H}, G = E \vdash \mathbf{Q}(F) \quad \mathbf{H}, \neg(G = E) \vdash \mathbf{Q}(f(G))}{\mathbf{H} \vdash \mathbf{Q}((f \Leftarrow \{E \mapsto F\})(G))} \text{ OV_R}$$

$$\frac{\mathbf{H}, G \in \text{dom}(g), \mathbf{P}(g(G)) \vdash \mathbf{Q} \quad \mathbf{H}, \neg G \in \text{dom}(g), \mathbf{P}(f(G)) \vdash \mathbf{Q}}{\mathbf{H}, \mathbf{P}((f \Leftarrow g)(G)) \vdash \mathbf{Q}} \text{ OV_L}$$

$$\frac{\mathbf{H}, G \in \text{dom}(g) \vdash \mathbf{Q}(g(G)) \quad \mathbf{H}, \neg G \in \text{dom}(g) \vdash \mathbf{Q}(f(G))}{\mathbf{H} \vdash \mathbf{Q}((f \Leftarrow g)(G))} \text{OV_R}$$

In the four previous rules the \Leftarrow operator must appear at the "top level"

$$\frac{\mathbf{H} \vdash f^{-1} \in A \leftrightarrow B \quad \mathbf{H} \vdash \mathbf{Q}(f[S] \cap f[T])}{\mathbf{H} \vdash \mathbf{Q}(f[S \cap T])} \cap \text{DIS_R}$$

$$\frac{\mathbf{H} \vdash f \in A \leftrightarrow B \quad \mathbf{H} \vdash \mathbf{Q}(f^{-1}[S] \setminus f^{-1}[T])}{\mathbf{H} \vdash \mathbf{Q}(f^{-1}[S \setminus T])} \setminus \text{DIS_R}$$

In the two previous rules, the occurrence of f^{-1} must appear at the "top level". Moreover A and B denote some type. Similar left distribution rules exist

$$\frac{\mathbf{H} \vdash \text{WD}(\mathbf{Q}(\{f(E)\})) \quad \mathbf{H} \vdash \mathbf{Q}(\{f(E)\})}{\mathbf{H} \vdash \mathbf{Q}(f[\{E\}])} \text{SIM_R}$$

In the previous rule, the occurrence of f must appear at the "top level". A similar left simplification rule exists.

$$\frac{\mathbf{H} \vdash \text{WD}(\mathbf{Q}(g(f(x)))) \quad \mathbf{H} \vdash \mathbf{Q}(g(f(x)))}{\mathbf{H} \vdash \mathbf{Q}((f;g)(x))} \text{SIM_R}$$

In the previous rule, the occurrence of $f;g$ must appear at the "top level". A similar left simplification rule exists.

Finiteness

$$\frac{\mathbf{H} \vdash S \subseteq T \quad \mathbf{H} \vdash \text{finite}(T)}{\mathbf{H} \vdash \text{finite}(S)} \text{fin}_{\subseteq}\text{R}$$

For applying the previous rule, the user has to write the set corresponding to T in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash \text{finite}(S) \vee \dots \vee \text{finite}(T)}{\mathbf{H} \vdash \text{finite}(S \cap \dots \cap T)} \quad \text{fin}_{\cap}\text{-R}$$

$$\frac{\mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \text{finite}(S \setminus T)} \quad \text{fin}_{\setminus}\text{-R}$$

$$\frac{\mathbf{H} \vdash r \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(S) \quad \mathbf{H} \vdash \text{finite}(T)}{\mathbf{H} \vdash \text{finite}(r)} \quad \text{fin}_{\text{rel}}\text{-R}$$

For applying the previous rule, the user has to write the set corresponding to $S \leftrightarrow T$ in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash \text{finite}(r)}{\mathbf{H} \vdash \text{finite}(r[s])} \quad \text{fin}_{\text{rel_img}}\text{-R}$$

$$\frac{\mathbf{H} \vdash \text{finite}(r)}{\mathbf{H} \vdash \text{finite}(\text{ran}(r))} \quad \text{fin}_{\text{rel_ran}}\text{-R}$$

$$\frac{\mathbf{H} \vdash \text{finite}(r)}{\mathbf{H} \vdash \text{finite}(\text{dom}(r))} \quad \text{fin}_{\text{rel_dom}}\text{-R}$$

$$\frac{\mathbf{H} \vdash f \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \text{finite}(f)} \quad \text{fin}_{\text{fun1}}\text{-R}$$

For applying the previous rule, the user has to write the set corresponding to $S \leftrightarrow T$ in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash f^{-1} \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \text{finite}(f)} \quad \text{fin}_{\text{fun2}}\text{-R}$$

For applying the previous rule, the user has to write the set corresponding to $S \leftrightarrow T$ in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash f \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(s)}{\mathbf{H} \vdash \text{finite}(f[s])} \quad \text{fin_fun_img_R}$$

$$\frac{\mathbf{H} \vdash f \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \text{finite}(\text{ran}(f))} \quad \text{fin_fun_ran_R}$$

For applying the previous rule, the user has to write the set corresponding to $S \leftrightarrow T$ in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash f^{-1} \in S \leftrightarrow T \quad \mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \text{finite}(\text{dom}(f))} \quad \text{fin_fun_dom_R}$$

For applying the previous rule, the user has to write the set corresponding to $S \leftrightarrow T$ in the editing area of the Proof Control Window.

$$\frac{\mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow n \leq x)}$$

$$\frac{\mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow x \geq n)}$$

$$\frac{\mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow n \geq x)}$$

$$\frac{\mathbf{H} \vdash \text{finite}(S)}{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow x \leq n)}$$

In the four previous rules, S must not contain any bound variable.

$$\frac{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow n \leq x) \quad \mathbf{H} \vdash S \subseteq \mathbb{Z} \setminus \mathbb{N}_1}{\mathbf{H} \vdash \text{finite}(S)}$$

$$\frac{\mathbf{H} \vdash \exists n \cdot (\forall x \cdot x \in S \Rightarrow x \leq n) \quad \mathbf{H} \vdash S \subseteq \mathbb{N}}{\mathbf{H} \vdash \text{finite}(S)}$$

Cardinality

$$\frac{\mathbf{H}, a \leq b \vdash \mathbf{Q}(b - a + 1) \quad \mathbf{H}, b < a \vdash \mathbf{Q}(0)}{\mathbf{H} \vdash \mathbf{Q}(\text{card}(a \dots b))}$$

$$\frac{\mathbf{H}, a \leq b, \mathbf{P}(a - b + 1) \vdash \mathbf{Q} \quad \mathbf{H}, b < a, \mathbf{P}(0) \vdash \mathbf{Q}}{\mathbf{H}, \mathbf{P}(\text{card}(a \dots b)) \vdash \mathbf{Q}}$$

In the two previous rules, $\text{card}(a \dots b)$ must appear at "top-level".

$$\frac{\mathbf{H} \vdash \mathbf{P}(S \subseteq T)}{\mathbf{H} \vdash \mathbf{P}(\text{card}(S) \leq \text{card}(T))}$$

There are similar rules for other cases

A The Mathematical Language

The syntax, type-checking and well-definedness conditions of the mathematical language are described in a separate document entitled "The Event-B Mathematical Language".

B ASCII Representations of the Mathematical Symbols

The mathematical symbols used in the Event-B mathematical language are Unicode characters, beyond the ASCII subset. These characters are not always easy to input with a regular keyboard. To help users enter these characters, a list of standard input strings have been defined. When any of these strings is entered, it is automatically converted by the user interface to the corresponding Unicode character (i.e., mathematical symbol).

B.1 Atomic Symbols

ASCII	Symbol
true	\top
false	\perp
INT	\mathbb{Z}

ASCII	Symbol
NAT	\mathbb{N}
NAT1	\mathbb{N}_1
BOOL	BOOL

ASCII	Symbol
TRUE	TRUE
FALSE	FALSE
{}	\emptyset

B.2 Unary Operators

ASCII	Symbol
not	\neg
finite	finite
card	card
POW	\mathbb{P}
POW1	\mathbb{P}_1

ASCII	Symbol
union	union
inter	inter
dom	dom
ran	ran
prj1	prj_1

ASCII	Symbol
prj2	prj_2
id	id
min	min
max	max
—	—

B.3 Assignment Operators

ASCII	Symbol
:=	:=

ASCII	Symbol
:	:

ASCII	Symbol
::	: \in

B.4 Binary Operators

ASCII	Symbol
&	\wedge
or	\vee
=>	\Rightarrow
<=>	\Leftrightarrow
=	$=$
/=	\neq
:	\in
<<:	\subset
/<<:	$\not\subset$
<:	\subseteq
/<:	$\not\subseteq$
<	$<$
<=	\leq
>	$>$
>=	\geq
/:	\notin

ASCII	Symbol
->	\mapsto
<->	\leftrightarrow
<<->	\Leftrightarrow
<->>	\leftrightarrow
<<->>	\Leftrightarrow
+->	\mapsto
-->	\rightarrow
+-->>	\mapsto
-->>	\Rightarrow
>->>	\mapsto
/\	\cap
\	\cup
\	\setminus
**	\times
>->	\mapsto
>+>	\mapsto

ASCII	Symbol
<+	\Leftarrow
	\parallel
><	\otimes
;	$;$
<	\triangleleft
<<	\triangleleft
>	\triangleright
-	$-$
*	$*$
/	\div
mod	\bmod
..	\dots
^	\wedge
~	-1
+	$+$
>>	\triangleright

B.5 Quantifiers

ASCII	Symbol
!	\forall
#	\exists
%	λ

ASCII	Symbol
UNION	\cup
INTER	\cap

ASCII	Symbol
.	.

B.6 Bracketing

ASCII	Symbol
((
))

ASCII	Symbol
[[
]]

ASCII	Symbol
{	{
}	}

The Event-B Modelling Notation

J.-R. Abrial

October 2007

Version 1.5

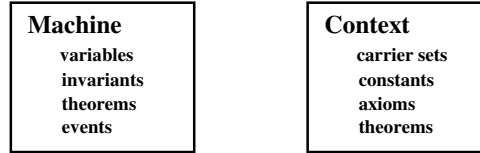
The Event-B Modelling Notation

Contents

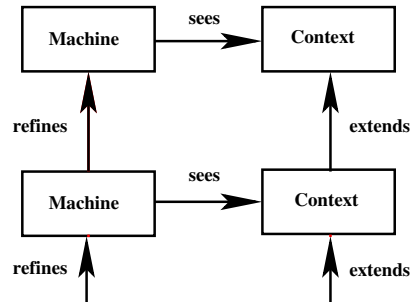
1	Machines and Contexts	1
2	Events	2
3	Variant	3
4	Actions	3
5	Witnesses	4
6	Syntax of the Event-B Mathematical Notation	4

1 Machines and Contexts

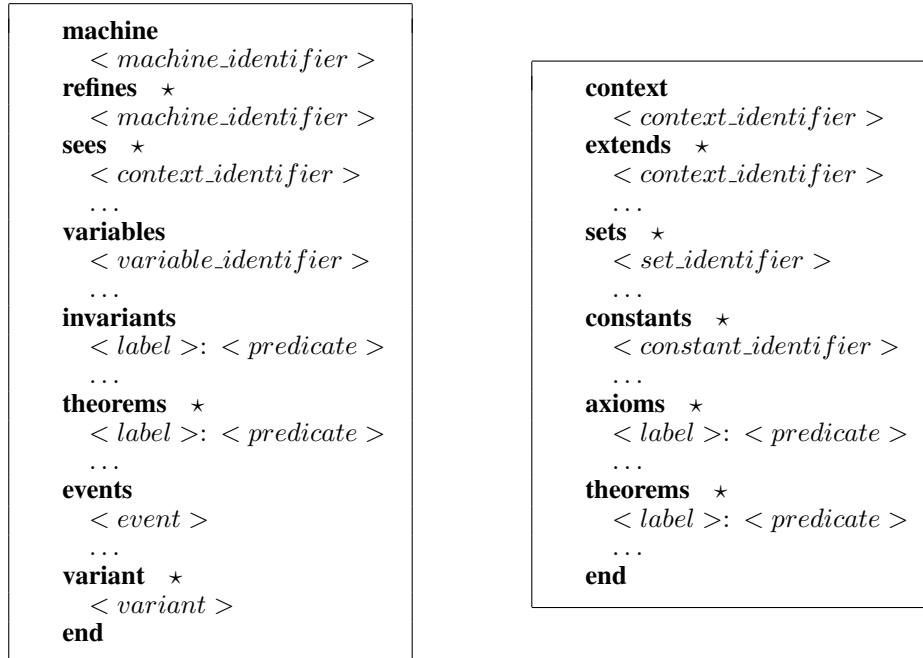
The primary concept in doing formal developments in Event-B is that of a *model*. A model contains the complete mathematical development of a *Discrete Transition System*. It is made of several components of two kinds: *machines* and *contexts*. Machines contain the variables, invariants, theorems, and events (section 2) of a model, whereas contexts contain carrier sets, constants, axioms, and theorems of a model:



Machines and contexts have various relationships: a machine can be "refined" by another one, and a context can be "extended" by another one (no cycles are allowed in both these relationships). Moreover, a machine can "see" one or several contexts. An example of machine and context relationship is as follows:



The following shows a more formal description of machines and contexts:



In these descriptions, section names (that is, **axioms**, **extends**, etc.) followed by a star \star are optional. Notice that all machine and context identifiers must be distinct in the same model

A model can only contain contexts, or only machines, or both. In the first case, the model represents a pure mathematical structure. In the third case, the machines of the model are parameterized by the contexts. Finally, the second case represents a machine which is not parameterized.

Notice that invariants, axioms and theorems denote predicates which are labelled: such labels must be distinct within a given section. Predicates are defined by means of the Event-B mathematical language (section 6).

The **variant** section is explained in section 3.

The previous representation of machines or contexts as well as the usage of keywords do not form a syntax as there is no such syntax in the Event-B notation. The only part of the Event-B notation which depends on a well defined syntax is the Event-B mathematical notation (section 6).

2 Events

A machine event represents a transition. It is essentially made of guards and actions (section 4). The guards together denote the necessary condition for the event to be enabled, whereas the actions together represent the way the variables of the corresponding machine are modified. Events can have no guards, they can be also simple and guarded (keyword **where**) or parameterized and guarded (keywords **any** and **where**). When an event lies in a machine which refines another one then the event may specify (if any) the abstract event(s) it refines (keyword **refines**). When a refining event refines an abstract event which is parameterized, one may provide some witnesses (keyword **with**), which are presented in section 5. All this is denoted as follows:

```

< event_identifier >  $\hat{=}$ 
status
  {normal, convergent, anticipated}
refines  $\star$ 
  < event_identifier >
  ...
any  $\star$ 
  < parameter_identifier >
  ...
where  $\star$ 
  < label >: < predicate >
  ...
with  $\star$ 
  < label >: < witness >
  ...
then  $\star$ 
  < label >: < action >
  ...
end

```

Notice that guards, witnesses, and actions are labelled. As said previously, such labels must be distinct within a given section.

Most of the time the **status** section states that the event is "normal". A "convergent" event must be a new event in a refined machine (one that does not appear in the abstraction). All convergent events in a

machine are concerned with the **variant** section of that machine (see section 3). An "anticipated" event is a new event which is not "convergent" yet but should become "convergent" in a subsequent refinement.

Notice that a new event might be "normal": it means that it is not concerned by the **variant** section. This is used quite often when using refinement to add more detail to a specification (without considering any liveness property of the system).

3 Variant

The **variant** section appears in a refined machine (section 1) containing some "convergent" events (section 2). This machine section contains either a natural number expression which must be decreased by each "convergent" event, or a finite set expression which must be made strictly smaller by each "convergent" event.

4 Actions

An action can be deterministic, in which case it is made of a list of distinct variable identifiers, followed by $:=$, followed by a list of expressions. The latter must be of the same length as the former. Variables which do not occur in the list are not modified. This is illustrated below:

$$< variable_identifier_list > := < expression_list >$$

Here is an example of a deterministic action in an event situated in machine with variables x , y , and z :

$$x, y := x + z, y - x$$

Variables x and y are modified as indicated whereas variable z is not modified.

Alternatively, an action can be non-deterministic, in which case it is made of a list of distinct variable identifiers, followed by $:\mid$, followed by a before-after predicate. Variables which do not occur in the list are not modified. This is illustrated below:

$$< variable_identifier_list > :\mid < before_after_predicate >$$

The before-after predicate may contain all variables of the machine: they denote the corresponding values just before the action takes place. It can also contain some of the variable identifiers of the list: such identifiers are primed, they denote the corresponding values just after the action has taken place. Example: suppose we have three variables x , y and z . Here is a non-deterministic action:

$$x, y :\mid x' > y \wedge y' > x' + z$$

Variable x becomes greater than y and the variable y becomes greater than x' (the new value for x) added to z (a variable which is not modified).

A final option is to define a non-deterministic action as a variable identifier (not a list), followed by $:\in$, followed by a set expression. This is illustrated below:

$$\boxed{\langle \text{variable_identifier} \rangle : \in \langle \text{set_expression} \rangle}$$

This form is just a special case of the previous one. It can always be translated to a non-deterministic case as shown in the following example. Suppose a machine with variables A , x , and y . Here is an action:

$$x : \in A \cup \{y\} \quad \text{which is the same as} \quad x : | \quad x' \in A \cup \{y\}$$

Variable x becomes a member of the set $A \cup \{y\}$, whereas variables A and y are not modified.

Finally, notice that all actions in the same event must be concerend with different variables. As an example, it is not possible to have the following actions in the same event:

$$\begin{array}{l} x := 5 \\ x : | \quad x' > x \end{array}$$

5 Witnesses

When a concrete event refines an abstract one which is parameterized, then all abstract parameters must receive a value in the concrete event. Such values are called the witnesses. Each witness is labelled with the concerned abstract parameter. The witness is defined by a predicate involving the abstract parameter. Most of the time, this predicate is a simple equality. Next is an example showing two witnesses. On the left hand side we have an abstract event named **pass** with two parameters. On the right hand side we have a concrete event named **new_pass** refining **pass**

```

pass ≐
  any
    p
    l
  where
    grd1 : p ↦ l ∈ aut
    grd2 : sit(p) ↦ l ∈ com
  then
    act1 : sit(p) := l
  end

```

```

new_pass ≐
  refines
    pass
  any
    d
  where
    grd1 : d ∈ ran(dap)
  with
    p : p = dap-1(d)
    l : l = dst(d)
  then
    act1 : sit(dap-1(d)) := dst(d)
    act2 : dap := dap ⋈ {d}
  end

```

When the concrete event is also parameterized then an abstract parameter which is the same as a concrete need not be given an explicit witness: it is always the corresponding concrete parameter.

6 Syntax of the Event-B Mathematical Notation

It is defined in a separate document entitled "The Event-B Mathematical Notation"

The Event-B Mathematical Language

Christophe Métayer (ClearSy)

Laurent Voisin (ETH Zurich)

October 26, 2007

Contents

1	Introduction	1
2	Language Lexicon	2
2.1	Whitespace	2
2.2	Identifiers	2
2.3	Integer Literals	3
2.4	Predicate symbols	4
2.5	Expression symbols	5
3	Language Syntax	7
3.1	Notation	7
3.2	Predicates	7
3.2.1	A first attempt	7
3.2.2	Associativity of operators	8
3.2.3	Priority of operators	9
3.2.4	Final syntax for predicates	10
3.3	Expressions	11
3.3.1	Some Fine Points	11
3.3.2	A First Attempt	13
3.3.3	Operator Groups	14
3.3.4	Priority of Operator Groups	16
3.3.5	Associativity of operators	16
3.3.6	Final syntax for expressions	18
4	Static Checking	21
4.1	Abstract Syntax	21
4.2	Well-formedness	22
4.3	Type Checking	26
4.3.1	Typing Concepts	26
4.3.2	Specification of Type Check	27
4.3.3	Examples	35
5	Dynamic Checking	40
5.1	Predicate Well-Definedness	40
5.2	Expression Well-Definedness	40

1 Introduction

This document presents the technical aspects of the kernel mathematical language of event-B. Beyond the pure syntax of the language, it also describes its lexical structure and various checks (both static and dynamic) that can be done on formulas on the language.

The main design principle of the language is to have intuitive priorities for operators and to use a minimal set of parenthesis (except when needed to resolve common ambiguities). So, the emphasis is really on having formulas unambiguous and easy to read.

The first chapter describes the lexicon used by the language, then chapter two describes its (concrete) syntax. Chapter three introduces the notion of well-formed and well-typed formula (static checks). Finally, chapter four gives the well-definedness conditions for a formula (dynamic check).

Revision History

Date	Contents
2005/05/31	Initial revision (Rodin Deliverable D7).
2006/05/24	Added min and max unary operators.
2007/10/26	Minor corrections in the text.

2 Language Lexicon

This chapter describes the lexicon of the mathematical language, that is the way that terminal tokens of the language grammar are built from a stream of characters.

Here, we assume that the input stream is made of Unicode characters, as defined in the Unicode standard 4.0 [4]. As we use only characters of the Basic Multilingual Plane, all characters are designated by their code points, that is an uppercase letter ‘U’ followed by a plus sign and an integer value (made of four hexadecimal digits). For instance, the classical space character is designated by U+0020.

Each token is formed by considering the longest sequence of characters that matches one of the definition below.

2.1 Whitespace

Whitespace characters are used to separate tokens or to improve the legibility of the formula. They are otherwise ignored during lexical analysis.

The whitespace characters of the mathematical language are the Unicode 4.0 space characters:

U+0020	U+00A0	U+1680	U+180E	U+2000	U+2001
U+2002	U+2003	U+2004	U+2005	U+2006	U+2007
U+2008	U+2009	U+200A	U+200B	U+2028	U+2029
U+202F	U+205F	U+3000			

together with the following control characters (these are the same as in the Java Language):

U+0009	U+000A	U+000B	U+000C	U+000D
U+001C	U+001D	U+001E	U+001F	

2.2 Identifiers

The identifiers of the mathematical language are defined in the same way as in the Unicode standard [4, par. 5.15]. This definition is not repeated here. Basically, an identifier is a sequence of characters that enjoy some special property, like referring to a letter or a digit.

Some identifiers are reserved for the mathematical language, where a predefined meaning is assigned to them. These reserved keywords are the following

identifiers made of ASCII letters and digits:

BOOL	FALSE	TRUE		
bool	card	dom	finite	id
inter	max	min	mod	pred
prj1	prj2	ran	succ	union

together with those other identifiers that use non-ASCII characters:

Token	Code points	Token name
\mathbb{N}	U+2115	SET OF NATURAL NUMBERS
\mathbb{N}_1	U+2115 U+0031	SET OF POSITIVE NUMBERS
\mathbb{P}	U+2119	POWERSET
\mathbb{P}_1	U+2119 U+0031	SET OF NON-EMPTY SUBSETS
\mathbb{Z}	U+2124	SET OF INTEGERS

2.3 Integer Literals

Integer literals consists of a non-empty sequence of ASCII decimal digits:

U+0030	U+0031	U+0032	U+0033	U+0034
U+0035	U+0036	U+0037	U+0038	U+0039

Note: There are two ways to tokenize integer literals: either signed or unsigned. The first case as the advantage that it corresponds to classical usage in mathematics. For instance, the string -1 is thought as representing a number, not a unary minus operator followed by a number. But, as we use the same character to designate both unary and binary minus, this causes problems: the lexical analysis is no longer context-free, but depends on the syntax of the language.

There are basically two solutions to this problem. One, taken in some functional languages in the ML family and in the Z notation, is to use different characters to represent the unary and binary minus operator. However, this comes against mathematical tradition and is thus rejected. The second solution is to consider that integer literals are unsigned. This second solution has been chosen here.

2.4 Predicate symbols

The tokens used in the pure predicate calculus are:

Token	Code point	Token name
(U+0028	LEFT PARENTHESIS
)	U+0029	RIGHT PARENTHESIS
\Leftrightarrow	U+21D4	LOGICAL EQUIVALENCE
\Rightarrow	U+21D2	LOGICAL IMPLICATION
\wedge	U+2227	LOGICAL AND
\vee	U+2228	LOGICAL OR
\neg	U+00AC	NOT SIGN
\top	U+22A4	TRUE PREDICATE
\perp	U+22A5	FALSE PREDICATE
\forall	U+2200	FOR ALL
\exists	U+2203	THERE EXISTS
,	U+002C	COMMA
.	U+00B7	MIDDLE DOT

The symbolic tokens used to build predicates from expressions are:

Token	Code point	Token name
=	U+003D	EQUALS SIGN
\neq	U+2260	NOT EQUAL TO
<	U+003C	LESS-THAN SIGN
\leq	U+2264	LESS THAN OR EQUAL TO
>	U+003E	GREATER-THAN SIGN
\geq	U+2265	GREATER THAN OR EQUAL TO
\in	U+2208	ELEMENT OF
\notin	U+2209	NOT AN ELEMENT OF
\subset	U+2282	SUBSET OF
$\not\subset$	U+2284	NOT A SUBSET OF
\subseteq	U+2286	SUBSET OF OR EQUAL TO
$\not\subseteq$	U+2288	NEITHER A SUBSET OF NOR EQUAL TO

2.5 Expression symbols

The following symbolic tokens are used to build sets of relations (or functions):

Token	Code point	Token name
\leftrightarrow	U+2194	RELATION
\Leftrightarrow	U+E100	TOTAL RELATION
\Rrightarrow	U+E101	SURJECTIVE RELATION
\Leftrightarrow	U+E102	TOTAL SURJECTIVE RELATION
\rightarrow	U+21F8	PARTIAL FUNCTION
\rightarrow	U+2192	TOTAL FUNCTION
\rightarrow	U+2914	PARTIAL INJECTION
\rightarrow	U+21A3	TOTAL INJECTION
\rightarrow	U+2900	PARTIAL SURJECTION
\rightarrow	U+21A0	TOTAL SURJECTION
\rightarrow	U+2916	BIJECTION

The following symbolic tokens are used for manipulating sets:

Token	Code point	Token name
$\{$	U+007B	LEFT CURLY BRACKET
$\}$	U+007D	RIGHT CURLY BRACKET
\mapsto	U+21A6	MAPLET
\emptyset	U+2205	EMPTY SET
\cap	U+2229	INTERSECTION
\cup	U+222A	UNION
\setminus	U+2216	SET MINUS
\times	U+00D7	CARTESIAN PRODUCT

The following symbolic tokens are used for manipulating relations and functions:

Token	Code point	Token name
$[$	U+005B	LEFT SQUARE BRACKET
$]$	U+005D	RIGHT SQUARE BRACKET
\mapsto	U+21A6	MAPLET
\Leftarrow	U+E103	RELATION OVERRIDING
\circ	U+2218	BACKWARD COMPOSITION
$;$	U+003B	FORWARD COMPOSITION
\otimes	U+2297	DIRECT PRODUCT
\parallel	U+2225	PARALLEL PRODUCT
-1	U+223C	TILDE OPERATOR
\triangleleft	U+25C1	DOMAIN RESTRICTION
\triangleleft	U+2A64	DOMAIN SUBTRACTION
\triangleright	U+25B7	RANGE RESTRICTION
\triangleright	U+2A65	RANGE SUBTRACTION

The following symbolic tokens are used in quantified expressions:

Token	Code point	Token name
λ	U+03BB	LAMBDA
\cap	U+22C2	N-ARY INTERSECTION
\cup	U+22C3	N-ARY UNION
$ $	U+2223	SUCH THAT

The following symbolic tokens are used in arithmetic expressions:

Token	Code point	Token name
\dots	U+2025	UPTO OPERATOR
$+$	U+002B	PLUS SIGN
$-$	U+2212	MINUS SIGN
$*$	U+2217	ASTERISK OPERATOR
\div	U+00F7	DIVISION SIGN
\wedge	U+005E	EXPONENTIATION SIGN

3 Language Syntax

This chapter describes the syntax of the mathematical language, giving the rationale behind the design decisions made.

We first present the notation we use to describe the syntax of the mathematical language. Then, we present the syntax of predicates and of expressions. In each case, we first present a simple ambiguous grammar, then we tackle with associativity and priorities of operators, giving a rationale for each choice made. Finally, we give a complete and non-ambiguous syntax.

3.1 Notation

In this document, we use an Extended Backus-Naur Form (EBNF) to describe syntax. In that notation, non-terminals are surrounded by angle brackets and terminals surrounded by single quotes. The other symbols are meta-symbols:

- Symbol $::=$ defines the non-terminal appearing on its left in terms of the syntax on its right.
- Parenthesis (and) are used for grouping.
- A vertical bar | denotes alternation.
- Square brackets [and] surround an optional part.
- Curly brackets { and } surround a part that can be repeated zero or more times.

3.2 Predicates

The point here is to define a grammar which is quite similar to the one used commonly when writing mathematical formulae but that should also be non-ambiguous to the (human) reader.

3.2.1 A first attempt

The grammar commonly used for predicates can loosely be defined as follows:

$$\begin{aligned} \langle predicate \rangle &::= '(\langle predicate \rangle) ' \\ &\quad | \langle predicate \rangle ' \Leftrightarrow ' \langle predicate \rangle \\ &\quad | \langle predicate \rangle ' \Rightarrow ' \langle predicate \rangle \\ &\quad | \langle predicate \rangle ' \wedge ' \langle predicate \rangle \end{aligned}$$

$$\begin{array}{l}
| \langle predicate \rangle ' \vee ' \langle predicate \rangle \\
| ' \neg ' \langle predicate \rangle \\
| ' \top ' \\
| ' \perp ' \\
| ' \forall ' \langle ident-list \rangle ' . ' \langle predicate \rangle \\
| ' \exists ' \langle ident-list \rangle ' . ' \langle predicate \rangle \\
| ' finite ' ' (' \langle expression \rangle ') ' \\
| \langle expression \rangle ' = ' \langle expression \rangle \\
| \langle expression \rangle ' \in ' \langle expression \rangle \\
| \langle expression \rangle ' \leq ' \langle expression \rangle \\
| \dots
\end{array}$$

$$\begin{array}{l}
\langle ident-list \rangle ::= \langle ident-list \rangle ' , ' \langle ident \rangle \\
| \langle ident \rangle
\end{array}$$

The ellipsis which appears at the end of the $\langle predicate \rangle$ production rule means that there are still more alternatives combining two expressions into a predicate. All those alternatives are not really relevant at this point of the document, but will be fully listed in the final syntax (see section 3.2.4 on page 10).

3.2.2 Associativity of operators

In this document, we use the term *associativity* with somewhat two different meanings. In a mathematical context, when we write that an operator, say \circ , is associative, we mean that it has a special mathematical property, namely that $(x \circ y) \circ z$ has the same value as $x \circ (y \circ z)$. In a syntactical context, we say that an operator is left-associative when formula $x \circ y \circ z$ (without any parenthesis) is parsed as if it would have been written $(x \circ y) \circ z$. To avoid any ambiguity, we will always write *associative in the algebraic sense* when we refer to the first meaning, the bare word *associative* always having the syntactical meaning.

Caution

Getting back to our predicate grammar defined above, we see that it is somewhat ambiguous. A first point is that it doesn't specify how one should parse formulae containing twice the same binary predicate operator without any parenthesis such as

$$P \Rightarrow Q \Rightarrow R$$

$$P \wedge Q \wedge R$$

To solve that ambiguity, one specifies that each binary operator has a property called *associativity*. The associativities defined for the event-B language are the following:

Operator	Associativity
\Leftrightarrow	none
\Rightarrow	none
\wedge	left
\vee	left

As a consequence, formula $P \Rightarrow Q \Rightarrow R$ is considered as ill-formed and not part of the event-B language, whereas formula $P \wedge Q \wedge R$ will be parsed as if it actually were written as $(P \wedge Q) \wedge R$.

The rationale for these associativities is quite simple. Operator \wedge is associative in the algebraic sense, so formulae $(P \wedge Q) \wedge R$ and $P \wedge (Q \wedge R)$ have the same meaning. Hence, one can pick up either left or right associativity for this operator. We arbitrarily chose left associativity as it is the most commonly used to our knowledge. The same rationale explains the choice of left associativity for operator \vee .

On the other hand, operator \Rightarrow is not associative in the algebraic sense ($(P \Rightarrow Q) \Rightarrow R$ is not the same as $P \Rightarrow (Q \Rightarrow R)$ (just suppose that predicates P , Q and R are all \perp). As a consequence, we keep it non associative in the language, rather than choosing an arbitrary associativity.

The case of operator \Leftrightarrow is somewhat special. This operator is indeed associative in the algebraic sense. However, mathematicians often write formula $P \Leftrightarrow Q \Leftrightarrow R$ when they actually mean $(P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R)$. Hence, we chose to make that operator non associative in the event-B language to avoid any ambiguity.

Finally, for the operators that build a predicate from two expressions (such as $=$, \in , etc.), the grammar given above doesn't allow formulae like $x = y = z$, so those operator can not be associative.

3.2.3 Priority of operators

Another source of ambiguity is the case where formulae contain two different predicate operators without any parenthesis such as

$$P \Rightarrow Q \Leftrightarrow R$$

$$P \wedge Q \vee R$$

$$\neg P \wedge Q$$

$$\forall x. P \vee Q$$

This kind of ambiguity is generally resolved by defining priorities among operators which define how much *binding power* each operator has. We will use that mechanism here, retaining the most commonly used priorities. But, with the addition that we want to forbid cases where those priorities are not so well-accepted.

For instance, some people expect operators ' \wedge ' and ' \vee ' to have the same priority, while others expect operator ' \wedge ' to have higher priority. So when faced with formula $P \vee Q \wedge R$, some people read it as $(P \vee Q) \wedge R$ while others read it as $P \vee (Q \wedge R)$, which is quite different (just replace P and Q by \top and R by \perp to convince yourself).

To solve that ambiguity, we decided that operators ' \wedge ' and ' \vee ' indeed have the same priority, but that one cannot mix them together without using parenthesis. So, $P \wedge Q \vee R$ is considered ill-formed. One should write either $(P \wedge Q) \vee R$ or $P \wedge (Q \vee R)$.

The priorities defined for the event-B language are the following (from lower

to higher priority)

$$\begin{aligned}
& \forall x \cdot P \text{ and } \exists x \cdot P \quad (\text{mixing allowed}) \\
& P \Rightarrow Q \text{ and } P \Leftrightarrow Q \quad (\text{mixing not allowed}) \\
& P \wedge Q \text{ and } P \vee Q \quad (\text{mixing not allowed}) \\
& \neg P
\end{aligned}$$

We choose to give quantified predicates the lowest priority in order to ease their reading when embedded in long formulae. The main consequence of this choice is that the scope of the variables introduced by a quantifier is the longest sub-formula. For instance, in formula $(\forall x \cdot P \Rightarrow Q) \Rightarrow R$, the scope of variable x extends until predicate Q as can be easily seen by looking at matching parenthesis.

The following formulae show some examples of how those priorities are used to replace parenthesis in some common cases:

$$\begin{aligned}
P \wedge Q \Rightarrow R & \text{ is parsed as } (P \wedge Q) \Rightarrow R \\
\forall x \cdot \exists y \cdot P & \text{ is parsed as } \forall x \cdot (\exists y \cdot P) \\
\forall x \cdot P \Rightarrow Q & \text{ is parsed as } \forall x \cdot (P \Rightarrow Q) \\
\forall x \cdot P \wedge Q & \text{ is parsed as } \forall x \cdot (P \wedge Q) \\
\forall x \cdot \neg P & \text{ is parsed as } \forall x \cdot (\neg P) \\
\neg P \Rightarrow Q & \text{ is parsed as } (\neg P) \Rightarrow Q \\
\neg P \wedge Q & \text{ is parsed as } (\neg P) \wedge Q
\end{aligned}$$

One should notice the difference with *classical* B [1] where $\forall x \cdot P \Rightarrow Q$ is parsed as $(\forall x \cdot P) \Rightarrow Q$ whereas, again, it is parsed here as $\forall x \cdot (P \Rightarrow Q)$.

3.2.4 Final syntax for predicates

As a result, we obtain the following non ambiguous grammar for predicates:

$$\begin{aligned}
\langle \text{predicate} \rangle & ::= \{ \langle \text{quantifier} \rangle \} \langle \text{unquantified-predicate} \rangle \\
\langle \text{quantifier} \rangle & ::= ' \forall ' \langle \text{ident-list} \rangle ' \cdot ' \\
& \quad | ' \exists ' \langle \text{ident-list} \rangle ' \cdot ' \\
\langle \text{ident-list} \rangle & ::= \langle \text{ident} \rangle \{ ' , ' \langle \text{ident} \rangle \} \\
\langle \text{unquantified-predicate} \rangle & ::= \langle \text{simple-predicate} \rangle [' \Rightarrow ' \langle \text{simple-predicate} \rangle] \\
& \quad | \langle \text{simple-predicate} \rangle [' \Leftrightarrow ' \langle \text{simple-predicate} \rangle] \\
\langle \text{simple-predicate} \rangle & ::= \langle \text{literal-predicate} \rangle \{ ' \wedge ' \langle \text{literal-predicate} \rangle \} \\
& \quad | \langle \text{literal-predicate} \rangle \{ ' \vee ' \langle \text{literal-predicate} \rangle \} \\
\langle \text{literal-predicate} \rangle & ::= \{ ' \neg ' \} \langle \text{atomic-predicate} \rangle
\end{aligned}$$

$$\begin{aligned}
\langle \textit{atomic-predicate} \rangle & ::= ' \bot ' \\
& \quad | ' \top ' \\
& \quad | \text{'finite' } ' (' \langle \textit{expression} \rangle ') ' \\
& \quad | \langle \textit{pair-expression} \rangle \langle \textit{relop} \rangle \langle \textit{pair-expression} \rangle \\
& \quad | ' (' \langle \textit{predicate} \rangle ') ' \\
\\
\langle \textit{relop} \rangle & ::= '=' | '\neq' \\
& \quad | '\in' | '\notin' | '<' | '<\neq' | '\subseteq' | '\not\subseteq' \\
& \quad | '<' | '\leq' | '>' | '\geq'
\end{aligned}$$

Please note that for relational predicates, we are using `[pair-expression]` instead of `[expression]`. That change will only allow expressions without quantifiers on each side of the relational operator. As a consequence, when one wants to use a quantified expression on either side, one will have to surround it with parenthesis. For instance, predicate $\lambda x. x \in \mathbb{Z} \mid x = id(\mathbb{Z})$ is not well-formed, one must write instead $(\lambda x. x \in \mathbb{Z} \mid x) = id(\mathbb{Z})$.

3.3 Expressions

The design principle for the syntax of expressions is the same as that of predicates, namely to enhance readability. To fulfill this goal, we use the same techniques: minimize the need for parenthesis where they are not really needed and prevent mixing operators when such a mix would be ambiguous.

3.3.1 Some Fine Points

Before presenting a first attempt of the syntax of expressions, we shall study some fine points about pairs, set comprehension, lambda abstraction, quantified expressions, and first and second projections.

Pair Construction. Pairs of expressions are constructed using the *maplet* operator \mapsto . Contrary to classical B [1], it is not possible to use a comma anymore. This change is due to the ambiguity of using commas for two different purposes in classical B: as a pair constructor and as a separator. For instance, set $\{1, 2\}$ can be seen as either a set containing the pair $(1, 2)$ or as a set containing the two elements 1 and 2. That was very confusing.

In event-B, a comma is always a separator and a maplet is a pair constructor. Below are some examples showing the consequences of this new approach:

Classical-B	Event-B
$x, y \in S$	$x \mapsto y \in S$
$x, y = z, t$	$x \mapsto y = z \mapsto t$
$f(x, y)$	$f(x \mapsto y)$

The last example is particularly blatant of the confusion between separator and pair constructor in classical B. When looking at formula $f(x, y)$, one has the impression that function f takes two separate arguments. But, this is not always true. For instance, variable x could hide a non scalar value. For instance,

suppose that $x = a \mapsto b$, then the function application could be rewritten as either $f(a \mapsto b, y)$ or even as $f(a, b, y)$. In that latter case, function f now appears to take three arguments. This is clearly not satisfactory. In fact, function f only takes one argument, which can happen to be a pair. In that latter case, one should use a pair constructor to create that pair, that is use a maplet operator.

Set Comprehension. There are now two forms of set comprehension. The most general one is $\{x \cdot P(x) \mid E(x)\}$ which describes the set whose elements are $E(x)$, for all x such that $P(x)$ holds. For instance, the set of all even natural numbers can be written as $\{x \cdot x \in \mathbb{N} \mid 2 * x\}$.

The second form $\{E \mid P\}$ is just a short-hand for the first-one, which allows to write things more compactly. The difference from the first form is that the variables that are bound by the construct are not listed explicitly. They are inferred from the expression part. Continuing with our previous example, the set of all even natural numbers can then be written more compactly as $\{2 * x \mid x \in \mathbb{N}\}$, which corresponds more to the classical mathematical notation.

The rule for determining the variables which are bound by this second form is to take all variables that occur free in E . Thus, if we denote by x the list of the variables that occur free in E , then the second form is equivalent to $\{x \cdot P \mid E\}$.

Lambda Abstraction. For lambda abstraction, classical B [1] uses the form $(\lambda x \cdot P \mid E)$ where x is a list of variables, P a predicate and E an expression. This notation is fine when x is reduced to only one variable. For instance, expression $(\lambda x \cdot x \in \mathbb{N} \mid x + 1)$ denotes the classical successor function on natural numbers. It is equal by definition to the set $\{x \cdot x \in \mathbb{N} \mid x + 1\}$.

But things get more complicated when x represents more than one variable. For instance, what is the meaning of expression $(\lambda a, b \cdot P \mid E)$. In classical B, the latter expression is defined as being the set $\{a, b \cdot P \mid a \mapsto b \mapsto E\}$. This is clearly unsatisfactory for event-B, as it turns out that, in the former expression, the comma that appears between a and b is not only a separator between two variables, but also a hidden pair constructor, as one can see when writing the equivalent set comprehension.

The crux of the matter is that the list of variables x introduced above, is much more than a simple list. Indeed, it describes the structure of the domain of the function defined by the lambda abstraction. For instance, when one writes, in classical B, the expression $(\lambda a, b \cdot P \mid E)$, one means that the domain of that function is $A \times B$ (where A and B are the types of bound variables a and b). Hence, the use of a comma is not appropriate here, as advocated in the paragraph above about *Pair Construction*.

The cure is easy, just say that x is not a list of variables, but a pattern that specifies the structure of the domain of the lambda abstraction. The example above is then to be written as $(\lambda a \mapsto b \cdot P \mid E)$. Moreover, this can be generalized to arbitrary domain structure by allowing arbitrary patterns after the lambda operator. The only constraints are that those patterns should be constructed out of distinct variables, pair constructors and parenthesis. The definition of the lambda abstraction $(\lambda x \cdot P \mid E)$ becomes $\{X \cdot P \mid x \mapsto E\}$ where X is the list of the variables that occur in x .

Other Quantified Expressions. The other quantified expressions are the quantified union and intersection. In this paragraph, we shall only consider quantified intersection, but everything will also apply to quantified union, *mutatis mutandis*.

A quantified intersection expression has the form $(\bigcap x \cdot P \mid E)$ where x is a list of variables, P a predicate and E an expression. It's defined as being a short form for the equivalent expression $\text{inter}(\{x \cdot P \mid E\})$ which mixes generalized intersection and set comprehension. But, as we have seen above, we also have a short form for writing set comprehension. The question then arises whether we could also define a short form for generalized intersection. The answer is yes. We then have a second form which is $(\bigcap E \mid P)$ and which is defined as being equal to $\text{inter}(\{E \mid P\})$.

Projections. In classical B [1], the first and second projection operators take two sets as arguments, like for instance in the expression $\text{prj}_1(A, B)$. In that expression, arguments A and B are used for two different purposes. On the one end, they allow to infer the type associated to the instantiated operator. On the other hand, they define the domain of the instantiated operator, which is $A \times B$.

This approach seems unnecessarily restrictive, as it puts a strong constraint on the domain of the operator, namely that it must be a cartesian product. So, in event-B, these operators become unary and take a relation as argument. The argument is then their domain. The upgrade path from classical B is quite straightforward, just replace $\text{prj}_1(A, B)$ by $\text{prj}_1(A \times B)$.

3.3.2 A First Attempt

An ambiguous grammar for event-B expressions can loosely be defined as follows:

$$\begin{aligned}
 \langle \text{expression} \rangle & ::= \langle \text{expression} \rangle \langle \text{binary-operator} \rangle \langle \text{expression} \rangle \\
 & \quad | \langle \text{unary-operator} \rangle \langle \text{expression} \rangle \\
 & \quad | \langle \text{expression} \rangle ^{-1} \\
 & \quad | \langle \text{expression} \rangle [' \langle \text{expression} \rangle '] \\
 & \quad | \langle \text{expression} \rangle (' \langle \text{expression} \rangle ') \\
 & \quad | \lambda \langle \text{ident-pattern} \rangle . \langle \text{predicate} \rangle [' \langle \text{expression} \rangle] \\
 & \quad | \langle \text{quantifier} \rangle \langle \text{ident-list} \rangle . \langle \text{predicate} \rangle [' \langle \text{expression} \rangle] \\
 & \quad | \langle \text{quantifier} \rangle \langle \text{expression} \rangle [' \langle \text{predicate} \rangle] \\
 & \quad | \{ \langle \text{ident-list} \rangle . \langle \text{predicate} \rangle [' \langle \text{expression} \rangle] \} \\
 & \quad | \{ \langle \text{expression} \rangle [' \langle \text{predicate} \rangle] \} \\
 & \quad | \text{bool} (' \langle \text{predicate} \rangle ') \\
 & \quad | \{ [\langle \text{expression-list} \rangle] \} \\
 & \quad | (' \langle \text{expression} \rangle ') \\
 & \quad | \emptyset \\
 & \quad | \mathbb{Z} \mid \mathbb{N} \mid \mathbb{N}_1 \\
 & \quad | \text{BOOL} \mid \text{TRUE} \mid \text{FALSE} \\
 & \quad | \langle \text{ident} \rangle \\
 & \quad | \langle \text{integer-literal} \rangle
 \end{aligned}$$

$$\begin{aligned}
\langle \text{binary-operator} \rangle &::= \text{'}\mapsto\text{' } | \text{'}\leftrightarrow\text{' } | \text{'}\longleftrightarrow\text{' } | \text{'}\longleftrightarrow\text{' } | \text{'}\longleftrightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } \\
&| \text{'}\rightarrow\text{' } | \text{'}\rightarrow\text{' } | \text{'}\cup\text{' } | \text{'}\cap\text{' } | \text{'}\backslash\text{' } | \text{'}\times\text{' } | \text{'}\otimes\text{' } | \text{'}\parallel\text{' } | \text{'}\circ\text{' } | \text{'};\text{' } | \text{'}\triangleleft\text{' } | \\
&\text{'}\triangleleft\text{' } | \text{'}\triangleleft\text{' } | \text{'}\triangleright\text{' } | \text{'}\triangleright\text{' } | \text{'}\dots\text{' } | \text{'}\text{+}\text{' } | \text{'}\text{-}\text{' } | \text{'}\text{*}\text{' } | \text{'}\text{/}\text{' } | \text{'}\text{mod}\text{' } | \text{'}\text{^}\text{' } \\
\langle \text{unary-operator} \rangle &::= \text{'}\text{-}\text{' } | \text{'}\text{card}\text{' } | \text{'}\mathbb{P}\text{' } | \text{'}\mathbb{P}_1\text{' } | \text{'}\text{union}\text{' } | \text{'}\text{inter}\text{' } | \text{'}\text{dom}\text{' } | \text{'}\text{ran}\text{' } | \\
&\text{'}\text{prj}_1\text{' } | \text{'}\text{prj}_2\text{' } | \text{'}\text{id}\text{' } | \text{'}\text{min}\text{' } | \text{'}\text{max}\text{' } \\
\langle \text{quantifier} \rangle &::= \text{'}\cup\text{' } | \text{'}\cap\text{' } \\
\langle \text{ident-pattern} \rangle &::= \langle \text{ident-pattern} \rangle \text{'}\mapsto\text{' } \langle \text{ident-pattern} \rangle \\
&| \text{'}\langle \text{ident-pattern} \rangle \text{' } \\
&| \langle \text{ident} \rangle \\
\langle \text{expression-list} \rangle &::= \langle \text{expression-list} \rangle \text{'},\text{' } \langle \text{expression} \rangle \\
&| \langle \text{expression} \rangle
\end{aligned}$$

As can be seen, there are many expression operators in the event-B language. So, we'll need to take a divide and conquer approach: to make things easier to grasp, we will first try to group all those operators into some categories.

3.3.3 Operator Groups

Basically, there are several kinds of expressions. The most important ones are shown in Figure 3.1. This figure reads as follows: there are three top-level kinds of expressions: sets, pairs and scalars. Relations and sets of relations are some special kinds of set. For instance, a relation between a set A and a set B is a subset of $A \times B$. The set of all relations between A and B is the set of all subsets of $A \times B$. Integers and booleans are also some special kind of scalar expression.

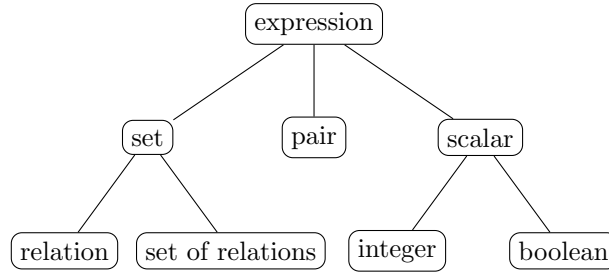


Figure 3.1: Kinds of expressions

We now define groups of similar expression operators (see Table 3.1 on the following page). The groups are defined by considering the shape of the operator (binary, unary, quantified, etc.) but also the kind of operator arguments and result. For each group, we will give one operator which will be used in the sequel as a distinguished representative of its group.

When examining that table, we can remark an interesting point: the operators that belong to the last three groups have the special property of being

Group	Description	Repr.
Quantification operators	Given a list of quantified identifiers, a predicate and an expression, these operators produce a new expression.	$\lambda x.P \mid E$
Pair constructor	Given two expressions, it produces a pair.	$E \mapsto F$
Set of relations constructors	Given two sets, these operators produce a set of relations.	$S \leftrightarrow T$
Binary set operators	Given two sets, these operators produce a new set.	$S \cup T$
Interval constructor	Given two integers, this operator produces a set.	$i \dots j$
Arithmetic operators	Given one or two integers, these operators produce a new integer.	$i + j$
Relational and functional image	Given a relation and an expression, these operators produce a new expression.	$r[s]$
Unary relation operator	Given a relation, this operator produces a new relation.	r^{-1}
Tightly bound unary operators	Given an expression, these operators produce another expression.	$\mathbb{P}(S)$
Predicate conversion	Given a predicate, this operator produces a new boolean expression.	$\text{bool}(P)$
Set enumeration and comprehension	Given a list of expressions, or a list of quantified variables, a predicate and an expression, this operator produces a set.	$\{\dots\}$

Table 3.1: Groups of similar expression operators

bounded: when one encounters such an operator, one can find easily where the expression involving that operator starts and where it ends: unary and ‘bool’ operators are always followed by a formula enclosed within parenthesis; set enumerations and comprehensions are enclosed within curly brackets. This is also the case of atomic expressions like integer and boolean literals or identifiers.

On the other hand, the operators of the other groups are not bounded by themselves, so one needs to define priorities and associativity laws for them in order to resolve potential ambiguities. We will first start by defining priorities between groups, then we will refine each group separately.

3.3.4 Priority of Operator Groups

We arbitrarily choose to define relative priorities such that groups of operators are sorted by increasing priority in table 3.1 on the previous page. As a consequence, quantification operators have the lowest priority.

That order has been chosen because it reduces the number of needed parenthesis when writing most common expressions. Here are a few example to illustrate this. Each expression is stated twice, first without parenthesis, then fully parenthesized:

$$\begin{aligned}
A \cup B \mapsto C & \text{ is parsed as } (A \cup B) \mapsto C \\
a + b \mapsto c & \text{ is parsed as } (a + b) \mapsto c \\
a .. b \cup C & \text{ is parsed as } (a .. b) \cup C \\
a + b .. c & \text{ is parsed as } (a + b) .. c \\
r^{-1} \cup s & \text{ is parsed as } (r^{-1}) \cup s \\
r^{-1}(s) & \text{ is parsed as } (r^{-1})(s)
\end{aligned}$$

Also, we give the lowest priority to quantification operators so that, when embedded in a formula, they have to be written surrounded by parenthesis. This is consistent with the choice made for quantified predicates. An example formula is

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1)^{-1}(3) = 2$$

3.3.5 Associativity of operators

Now, that priorities of groups have been defined, we will resolve remaining ambiguities separately for each group, defining how operators of each group can be mixed.

Quantification Operators. In this group, there is not much room for ambiguity, as when we encounter two quantification operators, it comes right from their syntax that the second one will be embedded in the first one. The only option left is whether the second quantified expression should be enclosed within parenthesis or not. We decide not to enforce parenthesis in this case. As a consequence, formula

$$\bigcap x \cdot x \subseteq \mathbb{Z} \mid \lambda y \cdot y = x \mid y \cup \{0\}$$

is parsed as

$$\bigcap x \cdot x \subseteq \mathbb{Z} \mid (\lambda y \cdot y = x \mid y \cup \{0\}) .$$

Pair Constructor. This group contains only the maplet operator, so we only have to define an associativity property for that operator. Although the maplet operator is not associative in the algebraic sense, it is very common usage to parse it as left-associative, so we shall keep that property. Then, an expression of the form $a \mapsto b \mapsto c$ will be parsed as $(a \mapsto b) \mapsto c$.

Set of Relations Constructors. No operator in this group is associative in the algebraic sense. However, we decide to parse them as right-associative. That choice is justified by the fact that we will parse function application as left-associative (this will be stated when we reach the *Relational and functional image* paragraph on the following page). As a consequence, one can write $f(a)(b)$ when one actually means $(f(a))(b)$. Then, to be consistent, one should also be able to describe properties of function f without parenthesis, so formula $f \in A \leftrightarrow B \leftrightarrow C$ should be parsed as $f \in A \leftrightarrow (B \leftrightarrow C)$.

Binary Set Operators. This group contains various operators which are more or less compatible each with the other. So, let's first see how one can safely mix these operators in a formula, from a mathematical point of view. Table 3.2 on the next page shows operator compatibility. We write a cross at the intersection of a row and a column if the two operators are compatible in the following sense: operator op_{row} is compatible with operator op_{col} if and only if the following equality holds

$$(A \text{ op}_{\text{row}} B) \text{ op}_{\text{col}} C = A \text{ op}_{\text{row}} (B \text{ op}_{\text{col}} C).$$

For instance, the cross at the intersection of row two and column three tells us that $(A \cap B) \setminus C = A \cap (B \setminus C)$ and the cross at the intersection of row nine and column seven tells us that $(A \triangleleft r) \otimes s = A \triangleleft (r \otimes s)$.

We can see that the shape is quite irregular and that there are not so many cases where operators are compatible. So, to have an unambiguous language, we should stick to that compatibility relation and forbid any unparenthesized combination of incompatible operators. When two operators are compatible, we parse them as left-associative. Otherwise, one needs to use parenthesis to resolve ambiguities. For instance, formula $S \cup T \cup U$ is parsed as $(S \cup T) \cup U$, while formula $S \cup T \cap U$ is ill-formed and is rejected. One has to make precise the meaning of that last formula, writing either $(S \cup T) \cap U$ or $S \cup (T \cap U)$.

There is only one case where we want to allow the combination of two incompatible operators: we parse the cartesian product operator as left-associative. This exception to the above rule is justified by the fact that we want to be consistent with the left-associativity we have given to the maplet operator. Then, one can write $a \mapsto b \mapsto c \in A \times B \times C$ when one actually means $(a \mapsto b) \mapsto c \in (A \times B) \times C$.

Interval Constructor. This group contains only one operator: $\text{'..'}.$ There is no point in having this operator used twice in the same formula (which would give the nonsensical formula $a .. b .. c$). So, this operator is parsed as non-associative.

	\cup	\cap	\setminus	\times	\circ	$;$	\otimes	\parallel	\Leftarrow	\triangleleft	\ll	\triangleright	\gg
\cup	\times												
\cap		\times	\times									\times	\times
\setminus													
\times													
\circ					\times								
$;$						\times						\times	\times
\otimes													
\parallel													
\Leftarrow									\times				
\triangleleft		\times	\times			\times	\times					\times	\times
\ll		\times	\times			\times	\times					\times	\times
\triangleright													
\gg													

Table 3.2: Compatibility of binary set operators

Arithmetic Operators. For these operators, we choose to retain the Ada language specification for defining priorities and associativity: operators ‘+’ and ‘−’ both have the same priority and are parsed as left-associative, operators ‘*’, ‘÷’ and ‘mod’ have higher priority and are also parsed as left-associative. Note that this choice is different from the one made for instance in the C language, where there is a special priority for unary ‘−’. We did not retain that last point as it can lead to valid but hard to read expressions like $a + - - - - b$ which means $a + b$.

Finally, the exponentiation operator has the least priority and is parsed as non-associative.

Relational and Functional Image. We choose to make these operations left-associative, although they are not associative in the algebraic sense. This follows common usage and is indeed important to have easy to read formulas. If these operators were not associative, one would have to write quite intricate formulas just to express successive function application: $((f(a))(b))(c)$. With the left-associativity we’ve added, this becomes $f(a)(b)(c)$.

Unary Relation Operator. This group contains only one operator ‘ $^{-1}$ ’, which can be repeated, obviously, so that r^{-1-1} is parsed as $(r^{-1})^{-1}$.

3.3.6 Final syntax for expressions

As a result, we obtain the following non ambiguous grammar for expressions. An important point is that non-terminals are named after the group of the top-level operators appearing in their production rule. This can be somewhat misleading as, for instance, $\langle pair-expression \rangle$ can be derived as formula \mathbb{Z} , which is clearly not a pair. This naming policy was chosen not to leave any information (just

numbering non-terminals 1, 2, ... would miss some structural property of the grammar).

$\langle \text{expression} \rangle$	$::= \lambda' \langle \text{ident-pattern} \rangle \cdot' \langle \text{predicate} \rangle \mid \langle \text{expression} \rangle$ $\mid \cup' \langle \text{ident-list} \rangle \cdot' \langle \text{predicate} \rangle \mid \langle \text{expression} \rangle$ $\mid \cup' \langle \text{expression} \rangle \mid \langle \text{predicate} \rangle$ $\mid \cap' \langle \text{ident-list} \rangle \cdot' \langle \text{predicate} \rangle \mid \langle \text{expression} \rangle$ $\mid \cap' \langle \text{expression} \rangle \mid \langle \text{predicate} \rangle$ $\mid \langle \text{pair-expression} \rangle$
$\langle \text{ident-pattern} \rangle$	$::= \langle \text{ident-pattern} \rangle \{ \mapsto' \langle \text{ident-pattern} \rangle \}$ $\mid \langle \langle \text{ident-pattern} \rangle \rangle$ $\mid \langle \text{ident} \rangle$
$\langle \text{pair-expression} \rangle$	$::= \langle \text{relation-set-expr} \rangle \{ \mapsto' \langle \text{relation-set-expr} \rangle \}$
$\langle \text{relation-set-expr} \rangle$	$::= \langle \text{set-expr} \rangle \{ \langle \text{relational-set-op} \rangle \langle \text{set-expr} \rangle \}$
$\langle \text{relational-set-op} \rangle$	$::= \leftrightarrow' \mid \Leftrightarrow' \mid \longleftrightarrow' \mid \longleftrightarrow'$ $\mid \rightarrowtail' \mid \rightarrow' \mid \rightharpoonup' \mid \rightharpoonup' \mid \dashrightarrow' \mid \dashrightarrow' \mid \twoheadrightarrow'$
$\langle \text{set-expr} \rangle$	$::= \langle \text{interval-expr} \rangle \{ \cup' \langle \text{interval-expr} \rangle \}$ $\mid \langle \text{interval-expr} \rangle \{ \times' \langle \text{interval-expr} \rangle \}$ $\mid \langle \text{interval-expr} \rangle \{ \Leftarrow' \langle \text{interval-expr} \rangle \}$ $\mid \langle \text{interval-expr} \rangle \{ \circ' \langle \text{interval-expr} \rangle \}$ $\mid \langle \text{interval-expr} \rangle \parallel' \langle \text{interval-expr} \rangle$ $\mid [\langle \text{domain-modifier} \rangle] \langle \text{relation-expr} \rangle$
$\langle \text{domain-modifier} \rangle$	$::= \langle \text{interval-expr} \rangle (\triangleleft' \mid \triangleleft')$
$\langle \text{relation-expr} \rangle$	$::= \langle \text{interval-expr} \rangle \otimes' \langle \text{interval-expr} \rangle$ $\mid \langle \text{interval-expr} \rangle \{ ;' \langle \text{interval-expr} \rangle \}$ $[\langle \text{range-modifier} \rangle]$ $\mid \langle \text{interval-expr} \rangle \{ \cap' \langle \text{interval-expr} \rangle \}$ $[\backslash' \langle \text{interval-expr} \rangle \mid \langle \text{range-modifier} \rangle]$
$\langle \text{range-modifier} \rangle$	$::= (\triangleright' \mid \triangleright') \langle \text{interval-expr} \rangle$
$\langle \text{interval-expr} \rangle$	$::= \langle \text{arithmetic-expr} \rangle [\dots' \langle \text{arithmetic-expr} \rangle]$
$\langle \text{arithmetic-expr} \rangle$	$::= [-'] \langle \text{term} \rangle \{ (+' \mid -') \langle \text{term} \rangle \}$
$\langle \text{term} \rangle$	$::= \langle \text{factor} \rangle \{ (*' \mid \div' \mid \text{mod}') \langle \text{factor} \rangle \}$
$\langle \text{factor} \rangle$	$::= \langle \text{image} \rangle [\wedge' \langle \text{image} \rangle]$
$\langle \text{image} \rangle$	$::= \langle \text{primary} \rangle \{ [\langle \text{expression} \rangle]' \mid (\langle \text{expression} \rangle)' \}$
$\langle \text{primary} \rangle$	$::= \langle \text{simple-expr} \rangle \{ ^{-1}' \}$

$$\begin{aligned}
\langle \textit{simple-expr} \rangle & ::= \text{'bool' '(' } \langle \textit{predicate} \rangle \text{' ')} \\
& \quad | \langle \textit{unary-op} \rangle \text{'(' } \langle \textit{expression} \rangle \text{' ')} \\
& \quad | \text{'(' } \langle \textit{expression} \rangle \text{' ')} \\
& \quad | \text{'{' } \langle \textit{ident-list} \rangle \text{'.' } \langle \textit{predicate} \rangle \text{'|' } \langle \textit{expression} \rangle \text{' '}} \\
& \quad | \text{'{' } \langle \textit{expression} \rangle \text{'|' } \langle \textit{predicate} \rangle \text{' '}} \\
& \quad | \text{'{' } [\langle \textit{expression} \rangle \text{'{' } \langle \textit{expression} \rangle \text{' '}}] \text{' '}} \\
& \quad | \text{'Z' | 'N' | 'N}_1 \text{' | 'BOOL' | 'TRUE' | 'FALSE' | '\emptyset'} \\
& \quad | \langle \textit{ident} \rangle \\
& \quad | \langle \textit{integer-literal} \rangle \\
\\
\langle \textit{unary-op} \rangle & ::= \text{'card' | 'P' | 'P}_1 \text{' | 'union' | 'inter' | 'dom' | 'ran' | 'prj}_1 \text{' } \\
& \quad | \text{'prj}_2 \text{' | 'id' | 'min' | 'max'}
\end{aligned}$$

4 Static Checking

This chapter describes how mathematical formulae (predicates and expressions) are to be statically checked for being meaningful. We first describe an abstract syntax for formulae. Then, we state the static checks that are to be done, based on that abstract syntax:

- well-formedness,
- type-check.

4.1 Abstract Syntax

In this section, we specify an abstract syntax for mathematical formulae. This abstract syntax is based on the concrete syntax described in Section 3.2.4 on page 10 and Section 3.3.6 on page 18. The difference is that the abstract syntax only conserves the essence of the concrete syntax. So, all concrete matter like priorities and tokens do not appear anymore.

The abstract syntax is described using production rules. Each rule has its own label. It is made of a left-hand part which denotes some kind of formula (predicate, expression, identifier list, expression list) and a right hand part which denotes a list of sub-formulae together with some attributes. To distinguish an attribute from a sub-formulae, we enclose the former within square brackets. Moreover, to make rules short, we use single letters, possibly subscripted, to denote formulae: a P denotes a predicate, E an expression, L a list of identifiers, I an identifier, M a list of expressions, and Q a pattern for lambda abstraction.

The production rules for predicates are:

$$\begin{aligned}
 \text{pred-bin: } P &::= P_1 P_2 [pred\text{-}binop] \\
 \text{pred-una: } P &::= P_1 \\
 \text{pred-quant: } P &::= L_1 P_1 [pred\text{-}quant] \\
 \text{pred-lit: } P &::= [pred\text{-}lit] \\
 \text{pred-simp: } P &::= E_1 \\
 \text{pred-rel: } P &::= E_1 E_2 [pred\text{-}relop]
 \end{aligned}$$

where

$$\begin{aligned}
 pred\text{-}binop &\in \{\text{land, lor, limp, leqv}\} \\
 pred\text{-}quant &\in \{\text{forall, exists}\} \\
 pred\text{-}lit &\in \{\text{btrue, bfalse}\} \\
 pred\text{-}relop &\in \left\{ \begin{array}{l} \text{equal, notequal, lt, le, gt, ge,} \\ \text{in, notin, subset, notsubset, subseteq, notsubseteq} \end{array} \right\}.
 \end{aligned}$$

The production rules for lists of identifiers and identifiers are:

$$\begin{aligned}\text{ident-list: } L &::= I_1 I_2 \dots I_n \\ \text{ident: } I &::= [name]\end{aligned}$$

where

$$\begin{aligned}1 &\leq n \\ name &\text{ is a string of characters.}\end{aligned}$$

The production rules for expressions are:

$$\begin{aligned}\text{expr-bin: } E &::= E_1 E_2 [\text{expr-binop}] \\ \text{expr-una: } E &::= E_1 [\text{expr-unop}] \\ \text{expr-lambda: } E &::= Q_1 P_1 E_1 \\ \text{expr-quant1: } E &::= L_1 P_1 E_1 [\text{expr-quant}] \\ \text{expr-quant2: } E &::= E_1 P_1 [\text{expr-quant}] \\ \text{expr-bool: } E &::= P_1 \\ \text{expr-eset: } E &::= M_1 \\ \text{expr-ident: } E &::= I_1 \\ \text{expr-atom: } E &::= [\text{expr-lit}] \\ \text{expr-int: } E &::= [\text{int-lit}] \\ \text{pattern: } Q &::= Q_1 Q_2 \\ \text{pattern-ident: } Q &::= I_1 \\ \text{expr-list: } M &::= E_1 E_2 \dots E_n\end{aligned}$$

where

$$\begin{aligned}\text{expr-binop} &\in \left\{ \begin{array}{l} \text{funimage, relimage, mapsto,} \\ \text{rel, trel, srel, strel,} \\ \text{pfun, tfun, pinj, tinj, psur, tsur, tbij,} \\ \text{bunion, binter, setminus, cprod, dprod, pprod,} \\ \text{bcomp, fcomp, ovl, domres, domsub, ranres, ransub,} \\ \text{upto, plus, minus, mul, div, mod, expn} \end{array} \right\} \\ \text{expr-unop} &\in \left\{ \begin{array}{l} \text{uminus, converse, card, pow, pow1,} \\ \text{union, inter, dom, ran, prj1, prj2, id, min, max} \end{array} \right\} \\ \text{expr-quant} &\in \{\text{qunion, qinter, cset}\} \\ \text{expr-lit} &\in \{\text{integer, natural, natural1, bool, true, false, emptyset}\} \\ \text{int-lit} &\text{ is an integer number.}\end{aligned}$$

4.2 Well-formedness

Each occurrence of an identifier in a formula (that is a predicate or an expression) can be either free or bound. Intuitively, a free occurrence of an identifier refers to a declaration of that identifier in a scope outside of the formula, while a bound occurrence corresponds to a local declaration introduced by a quantifier in the formula itself.

For a formula to be considered well-formed, we ask that, beyond being syntactically correct, it also satisfies the two following conditions:

1. Any identifier that occurs in the formula, should have only free occurrences or bound occurrences, but not both.

2. Any identifier that occurs bound in the formula, should be bound in exactly one place (i.e., by only one quantifier).

These conditions have been coined so that any occurrence of an identifier in a formula always denotes exactly the same data. This is a big win in formula legibility.

For instance, the following formula is ill-formed (it doesn't satisfy the first condition)

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1)(x) = x + 1$$

it should be written

$$(\lambda y \cdot y \in \mathbb{Z} \mid y + 1)(x) = x + 1 .$$

And the following formula is also ill-formed (failing to satisfy the second condition)

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1) = (\lambda x \cdot x \in \mathbb{Z} \mid x + 1)$$

it should be written

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1) = (\lambda y \cdot y \in \mathbb{Z} \mid y + 1) .$$

The rest of this section formalizes these well-formedness conditions using an attribute grammar formalism on the abstract syntax of formulae. For that, we add three attributes to the nodes of the abstract syntax tree:

- Attribute *bound* is synthesized and contains the set of identifiers that occur bound in the formula rooted at the current node.
- Attribute *free* is synthesized and contains the set of identifiers that occur free in the formula rooted at the current node.
- Attribute *woff* is synthesized and contains a boolean value which is TRUE if and only if the formula rooted at the current node is well-formed.

The value of these three attributes are given by the following set of equations on the production rules of the abstract syntax:

$$\begin{aligned} \text{pred-bin: } P &::= P_1 P_2 [\text{pred-binop}] \\ P.\text{bound} &= P_1.\text{bound} \cup P_2.\text{bound} \\ P.\text{free} &= P_1.\text{free} \cup P_2.\text{free} \\ P.\text{woff} &= \text{bool} \left(\begin{array}{l} P_1.\text{woff} = \text{TRUE} \\ \wedge P_2.\text{woff} = \text{TRUE} \\ \wedge P_1.\text{free} \cap P_2.\text{bound} = \emptyset \\ \wedge P_1.\text{bound} \cap P_2.\text{free} = \emptyset \\ \wedge P_1.\text{bound} \cap P_2.\text{bound} = \emptyset \end{array} \right) \end{aligned}$$

$$\begin{aligned} \text{pred-una: } P &::= P_1 \\ P.\text{bound} &= P_1.\text{bound} \\ P.\text{free} &= P_1.\text{free} \\ P.\text{woff} &= P_1.\text{woff} \end{aligned}$$

$$\begin{aligned}
\text{pred-quant: } P &::= L_1 \ P_1 \ [pred\text{-}quant] \\
P.\text{bound} &= P_1.\text{bound} \cup L_1.\text{free} \\
P.\text{free} &= P_1.\text{free} \setminus L_1.\text{free} \\
P.\text{wff} &= \text{bool} \left(\begin{array}{l} L_1.\text{wff} = \text{TRUE} \\ \wedge \ P_1.\text{wff} = \text{TRUE} \\ \wedge \ P_1.\text{bound} \cap L_1.\text{free} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{pred-lit: } P &::= [pred\text{-}lit] \\
P.\text{bound} &= \emptyset \\
P.\text{free} &= \emptyset \\
P.\text{wff} &= \text{TRUE}
\end{aligned}$$

$$\begin{aligned}
\text{pred-simp: } P &::= E_1 \\
P.\text{bound} &= E_1.\text{bound} \\
P.\text{free} &= E_1.\text{free} \\
P.\text{wff} &= E_1.\text{wff}
\end{aligned}$$

$$\begin{aligned}
\text{pred-rel: } P &::= E_1 \ E_2 \ [pred\text{-}relop] \\
P.\text{bound} &= E_1.\text{bound} \cup E_2.\text{bound} \\
P.\text{free} &= E_1.\text{free} \cup E_2.\text{free} \\
P.\text{wff} &= \text{bool} \left(\begin{array}{l} E_1.\text{wff} = \text{TRUE} \\ \wedge \ E_2.\text{wff} = \text{TRUE} \\ \wedge \ E_1.\text{free} \cap E_2.\text{bound} = \emptyset \\ \wedge \ E_1.\text{bound} \cap E_2.\text{free} = \emptyset \\ \wedge \ E_1.\text{bound} \cap E_2.\text{bound} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{ident-list: } L &::= I_1 \ I_2 \ \dots \ I_n \\
L.\text{bound} &= \emptyset \\
L.\text{free} &= \{k \cdot k \in 1 \dots n \mid I_k.\text{name}\} \\
L.\text{wff} &= \text{bool}(\forall i, j \cdot i \in 1 \dots n \wedge j \in 1 \dots n \wedge i \neq j \Rightarrow I_i.\text{name} \neq I_j.\text{name})
\end{aligned}$$

$$\begin{aligned}
\text{expr-bin: } E &::= E_1 \ E_2 \ [expr\text{-}binop] \\
E.\text{bound} &= E_1.\text{bound} \cup E_2.\text{bound} \\
E.\text{free} &= E_1.\text{free} \cup E_2.\text{free} \\
E.\text{wff} &= \text{bool} \left(\begin{array}{l} E_1.\text{wff} = \text{TRUE} \\ \wedge \ E_2.\text{wff} = \text{TRUE} \\ \wedge \ E_1.\text{free} \cap E_2.\text{bound} = \emptyset \\ \wedge \ E_1.\text{bound} \cap E_2.\text{free} = \emptyset \\ \wedge \ E_1.\text{bound} \cap E_2.\text{bound} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{expr-una: } E &::= E_1 \ [expr\text{-}unop] \\
E.\text{bound} &= E_1.\text{bound} \\
E.\text{free} &= E_1.\text{free} \\
E.\text{wff} &= E_1.\text{wff}
\end{aligned}$$

$$\text{expr-lambda: } E ::= Q_1 \ P_1 \ E_1$$

$$\begin{aligned}
E.\text{bound} &= P_1.\text{bound} \cup E_1.\text{bound} \cup Q_1.\text{free} \\
E.\text{free} &= (P_1.\text{free} \cup E_1.\text{free}) \setminus Q_1.\text{free} \\
E.\text{wff} &= \text{bool} \left(\begin{array}{l} Q_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{wff} = \text{TRUE} \\ \wedge \quad E_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{free} \cap E_1.\text{bound} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{free} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{bound} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap Q_1.\text{free} = \emptyset \\ \wedge \quad E_1.\text{bound} \cap Q_1.\text{free} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{expr-quant1: } E &::= L_1 \ P_1 \ E_1 \ [\text{expr-quant}] \\
E.\text{bound} &= P_1.\text{bound} \cup E_1.\text{bound} \cup L_1.\text{free} \\
E.\text{free} &= (P_1.\text{free} \cup E_1.\text{free}) \setminus L_1.\text{free} \\
E.\text{wff} &= \text{bool} \left(\begin{array}{l} L_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{wff} = \text{TRUE} \\ \wedge \quad E_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{free} \cap E_1.\text{bound} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{free} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{bound} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap L_1.\text{free} = \emptyset \\ \wedge \quad E_1.\text{bound} \cap L_1.\text{free} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{expr-quant2: } E &::= E_1 \ P_1 \ [\text{expr-quant}] \\
E.\text{bound} &= P_1.\text{bound} \cup E_1.\text{bound} \cup E_1.\text{free} \\
E.\text{free} &= P_1.\text{free} \setminus E_1.\text{free} \\
E.\text{wff} &= \text{bool} \left(\begin{array}{l} E_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{wff} = \text{TRUE} \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{bound} = \emptyset \\ \wedge \quad P_1.\text{bound} \cap E_1.\text{free} = \emptyset \end{array} \right)
\end{aligned}$$

$$\begin{aligned}
\text{expr-bool: } E &::= P_1 \\
E.\text{bound} &= P_1.\text{bound} \\
E.\text{free} &= P_1.\text{free} \\
E.\text{wff} &= P_1.\text{wff}
\end{aligned}$$

$$\begin{aligned}
\text{expr-eset: } E &::= M \\
E.\text{bound} &= M.\text{bound} \\
E.\text{free} &= M.\text{free} \\
E.\text{wff} &= M.\text{wff}
\end{aligned}$$

$$\begin{aligned}
\text{expr-ident: } E &::= I_1 \\
E.\text{bound} &= \emptyset \\
E.\text{free} &= \{I_1.\text{name}\} \\
E.\text{wff} &= \text{TRUE}
\end{aligned}$$

$$\begin{aligned}
\text{expr-atom: } E &::= [\text{expr-lit}] \\
E.\text{bound} &= \emptyset \\
E.\text{free} &= \emptyset \\
E.\text{wff} &= \text{TRUE}
\end{aligned}$$

expr-int: $E ::= [int-lit]$
 $E.bound = \emptyset$
 $E.free = \emptyset$
 $E.wff = \text{TRUE}$

pattern: $Q ::= Q_1 Q_2$
 $Q.bound = \emptyset$
 $Q.free = Q_1.free \cup Q_2.free$
 $Q.wff = \text{TRUE}$

pattern-ident: $Q ::= I_1$
 $Q.bound = \emptyset$
 $Q.free = \{I_1.name\}$
 $Q.wff = \text{TRUE}$

expr-list: $M ::= E_1 E_2 \dots E_n$
 $M.bound = (\bigcup k \cdot k \in 1 \dots n \mid E_k.bound)$
 $M.free = (\bigcup k \cdot k \in 1 \dots n \mid E_k.free)$
 $M.wff = \text{bool} \left(\begin{array}{l} \wedge \left(\begin{array}{l} (\forall k \cdot k \in 1 \dots n \Rightarrow E_k.wff = \text{TRUE}) \\ \wedge \left(\begin{array}{l} \forall i, j \cdot i \in 1 \dots n \wedge j \in 1 \dots n \wedge i \neq j \\ \Rightarrow E_i.bound \cap E_j.bound = \emptyset \end{array} \right) \end{array} \right) \\ \wedge \left(\begin{array}{l} \forall i, j \cdot i \in 1 \dots n \wedge j \in 1 \dots n \wedge i \neq j \\ \Rightarrow E_i.bound \cap E_j.free = \emptyset \end{array} \right) \end{array} \right)$

4.3 Type Checking

Type checking consists of checking, statically, that a formula is meaningful in a certain context. For that, we associate a type with each expression that occurs in a formula. This type is the set of all values that the expression can take. Then, we check that the formula abides by some type checking rules. Those rules enforce that the operators used can be meaningful. Unfortunately, type checking, as it is a static check, cannot by itself prove that a formula is meaningful. For some operators, like integer division, we will also need to check some additional dynamic constraints (e.g., that the denominator is not zero). This will be specified in the well-definedness dynamic checks (see chapter 5 on page 40).

The result of type checking is twofold. Firstly, it says whether a given formula is well-typed (that is abides by the type checking rules). Secondly, it computes an enriched context that associates a type with every identifier occurring free in the formula.

In the sequel of this section, we shall first specify more formally concepts such as type, type variable, typing environment and typing equation. Then, we shall specify type checking using an attribute grammar formalism as was done for well-formedness. Finally, we give some illustrating examples of type-checking.

4.3.1 Typing Concepts

As said previously, a type denotes the set of values that an expression can take. Moreover, we want this set to be derived statically, based on the form of the

expression and the context in which it appears. As a consequence, a type can take one of the three following forms:

- a basic set, that is a predefined set (\mathbb{Z} or BOOL) or a carrier set provided by the user (i.e., an identifier);
- a power set of another type, such as $\mathbb{P}(\mathbb{Z})$;
- a cartesian product of two types, such as $\mathbb{Z} \times \text{BOOL}$.

A type variable is a meta-variable that can denote any type. In the sequel, we shall use lowercase Greek letters ($\alpha, \beta, \gamma, \dots$) to denote type variables.

A typing environment represents the context in which a formula is to be type checked. A typing environment is a partial function from the set of all identifiers to the set of all possible types. For instance, the typing environment

$$\{\text{'a'} \mapsto \mathbb{Z}, \text{'b'} \mapsto \mathbb{P}(\mathbb{Z} \times \text{BOOL}), \text{'c'} \mapsto \alpha\}$$

says that identifier ‘a’ has type \mathbb{Z} , identifier ‘b’ has type $\mathbb{P}(\mathbb{Z} \times \text{BOOL})$ (i.e., is a relation between integers and booleans) and identifier ‘c’ is typed by type variable α .

If an identifier i has been defined as a carrier set, then it will appear in the typing environment as the pair $i \mapsto \mathbb{P}(i)$.

A typing equation is a pair of types. In the sequel, we will write typing equations as $\tau_1 \equiv \tau_2$, instead of the more classical pair $\tau_1 \mapsto \tau_2$. This is mere syntactical sugar to enhance legibility.

A typing equation is said to be *satisfiable* if, and only if, there exists an assignment to the type variables it contains such that, when replacing these type variables by their value, the two components of the pair are equal (i.e., denote the same type). For instance, typing equation $\alpha \times \text{BOOL} \equiv \mathbb{Z} \times \beta$ is satisfiable (take \mathbb{Z} for α and BOOL for β). In contrast, type equation $\mathbb{P}(\alpha) \equiv \mathbb{Z}$ and $\mathbb{Z} \equiv \text{'S'}$ are unsatisfiable (in the last sentence, remember that ‘S’ denotes a carrier set).

Similarly, a typing equation is said to be *uniquely satisfiable* if, and only if, there exists a unique assignment of type variables that satisfies it. For instance, $\alpha \equiv \mathbb{Z}$ is uniquely satisfiable (the only assignment that satisfies it is to take \mathbb{Z} for α), while the type equation $\alpha \equiv \beta$, although satisfiable, is not uniquely satisfiable (to satisfy it, we only need that α and β are assigned the same type, but that type is arbitrary).

These two notions of satisfiability are extended to sets of type equations, with the additional proviso, that the satisfying assignment of type variables is done globally for all type equations in the set. For instance, the set $\{\alpha \equiv \mathbb{Z}, \beta \equiv \text{BOOL}\}$ is (uniquely) satisfiable, while the set $\{\alpha \equiv \mathbb{Z}, \alpha \equiv \text{BOOL}\}$ is not satisfiable, although each equation, taken separately, is satisfiable.

4.3.2 Specification of Type Check

The abstract grammar of expressions is extended with the following attributes:

- Attribute *ityvars* (resp. *styvars*) is inherited (resp. synthesized) and contains the set of type variables that have been used so far.

- Attribute *ityenv* (resp. *styenv*) is inherited (resp. synthesized) and contains the current typing environment.
- Attribute *ityeqs* (resp. *styeqs*) is inherited (resp. synthesized) and contains the set of typing equations that have been collected so far.
- Attribute *type* is synthesized and contains a type.

These attributes are added to all non-terminals, except *type* which is not defined for predicates (there is no type associated with a predicate) nor list of identifiers.

Type checking then consists of initializing the attribute grammar by giving values to inherited attributes of the root R of the tree and then evaluating the attribute grammar. Type check succeeds iff, after evaluation, the set of typing equations $R.styeqs$ is uniquely satisfiable. Moreover, in case of success, the resulting typing environment is $R.styenv$, where all type variables have been replaced by the values that satisfy the latter set of typing equations.

Initialization of the attribute grammar consists of the following three equations (where R denotes the root of the tree):

$$\begin{aligned} R.ityvars &= \emptyset \\ R.ityenv &= \text{initial typing environment} \\ R.ityeqs &= \emptyset \end{aligned}$$

Please note that the initial typing environment must not contain any type variable.

The rest of this section describes the equations for each production rule of the attribute grammar. In some places, we use a shortcut to denote some set of equations. The notation

$$A.inherited = B.synthesized$$

means

$$\begin{aligned} A.ityvars &= B.styvars \\ A.ityenv &= B.styenv \\ A.ityeqs &= B.styeqs \end{aligned}$$

We also use the term *fresh type variable* to denote a type variable which doesn't occur in attribute *ityvars* of the left hand side of a production rule. For instance, in the equations of production rule **pred-rel**, α denotes a type variable such that $\alpha \notin P.ityvars$.

The set of equations of the attribute grammar is:

$$\begin{aligned} \text{pred-bin: } P &::= P_1 P_2 [pred\text{-}binop] \\ P_1.inherited &= P.inherited \\ P_2.inherited &= P_1.synthesized \\ P.synthesized &= P_2.synthesized \end{aligned}$$

$$\begin{aligned} \text{pred-una: } P &::= P_1 \\ P_1.inherited &= P.inherited \\ P.synthesized &= P_1.synthesized \end{aligned}$$

pred-quant: $P ::= L_1 P_1 [pred\text{-}quant]$
 $L_1.inherited = P.inherited$
 $P_1.inherited = L_1.synthesized$
 $P.synthesized = P_1.synthesized$

pred-lit: $P ::= [pred\text{-}lit]$
 $P.synthesized = P.inherited$

pred-simp: $P ::= E_1$
 Let α be a fresh type variable in
 $E_1.itvars = P.itvars \cup \{\alpha\}$
 $E_1.itenv = P.itenv$
 $E_1.ityeqs = P.ityeqs$
 $P.stvars = E_1.stvars$
 $P.stenv = E_1.stenv$
 $P.styeqs = E_1.styeqs \cup \{E_1.type \equiv \mathbb{P}(\alpha)\}$

pred-rel: $P ::= E_1 E_2 [pred\text{-}relop]$
 Let α be a fresh type variable in
 $E_1.itvars = P.itvars \cup \{\alpha\}$
 $E_1.itenv = P.itenv$
 $E_1.ityeqs = P.ityeqs$
 $E_2.inherited = E_1.synthesized$
 $P.stvars = E_2.stvars$
 $P.stenv = E_2.stenv$
 $P.styeqs = E_2.styeqs \cup \mathcal{E}$
 where \mathcal{E} is defined in the following table.

$P.pred\text{-}relop$	\mathcal{E}
equal, notequal	$\left\{ \begin{array}{l} E_1.type \equiv \alpha \\ E_2.type \equiv \alpha \end{array} \right\}$
lt, le, gt, ge	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{Z} \\ E_2.type \equiv \mathbb{Z} \end{array} \right\}$
in, notin	$\left\{ \begin{array}{l} E_1.type \equiv \alpha \\ E_2.type \equiv \mathbb{P}(\alpha) \end{array} \right\}$
subset, notsubset, subsest, notsubsest	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha) \\ E_2.type \equiv \mathbb{P}(\alpha) \end{array} \right\}$

ident-list: $L ::= I_1 I_2 \dots I_n$
 $I_1.inherited = L.inherited$
 $I_2.inherited = I_1.synthesized$
 \vdots
 $I_n.inherited = I_{n-1}.synthesized$
 $L.synthesized = I_n.synthesized$

$\text{ident: } I ::= [name]$
 if $I.name \in \text{dom}(I.itylv)$ then
 $I.synthesized = I.inherited$
 $I.type = I.itylv(I.name)$
 else let α be a fresh type variable in
 $I.stylv = I.itylv \cup \{\alpha\}$
 $I.stylv = I.itylv \cup \{I.name \mapsto \alpha\}$
 $I.styeqs = I.ityeqs$
 $I.type = \alpha$

$\text{expr-bin: } E ::= E_1 E_2 [expr-binop]$
 Let α, β, γ and δ be distinct fresh type variables in
 $E_1.itylv = E.itylv \cup \{\alpha, \beta, \gamma, \delta\}$
 $E_1.itylv = E.itylv$
 $E_1.ityeqs = E.ityeqs$
 $E_2.inherited = E_1.synthesized$
 $E.stylv = E_2.stylv$
 $E.stylv = E_2.stylv$
 $E.styeqs = E_2.styeqs \cup \mathcal{E}$
 $E.type = \tau$

where \mathcal{E} and τ are defined in Table 4.1 on the next page.

$\text{expr-una: } E ::= E_1 [expr-unop]$
 Let α and β be distinct fresh type variables in
 $E_1.itylv = E.itylv \cup \{\alpha, \beta\}$
 $E_1.itylv = E.itylv$
 $E_1.ityeqs = E.ityeqs$
 $E.stylv = E_1.stylv$
 $E.stylv = E_1.stylv$
 $E.styeqs = E_1.styeqs \cup \mathcal{E}$
 $E.type = \tau$

where \mathcal{E} and τ are defined in Table 4.2 on page 32.

$\text{expr-lambda: } E ::= Q_1 P_1 E_1$
 $Q_1.inherited = E.inherited$
 $P_1.inherited = Q_1.synthesized$
 $E_1.inherited = P_1.synthesized$
 $E.synthesized = E_1.synthesized$
 $E.type = \mathbb{P}(Q_1.type \times E_1.type)$

$E.expr-binop$	\mathcal{E}	τ
funimage	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \alpha \end{array} \right\}$	β
relimage	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \mathbb{P}(\alpha) \end{array} \right\}$	$\mathbb{P}(\beta)$
mapsto	\emptyset	$E_1.type \times E_2.type$
rel, trel, srel, strel, pfun, tfun, pinj, tinj, psur, tsur, tbij	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha) \\ E_2.type \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\mathbb{P}(\alpha \times \beta))$
bunion, binter, setminus	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha) \\ E_2.type \equiv \mathbb{P}(\alpha) \end{array} \right\}$	$\mathbb{P}(\alpha)$
cprod	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha) \\ E_2.type \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
dprod	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \mathbb{P}(\alpha \times \gamma) \end{array} \right\}$	$\mathbb{P}(\alpha \times (\beta \times \gamma))$
pprod	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \gamma) \\ E_2.type \equiv \mathbb{P}(\beta \times \delta) \end{array} \right\}$	$\mathbb{P}((\alpha \times \beta) \times (\gamma \times \delta))$
bcomp	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\beta \times \gamma) \\ E_2.type \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \gamma)$
fcomp	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \mathbb{P}(\beta \times \gamma) \end{array} \right\}$	$\mathbb{P}(\alpha \times \gamma)$
ovl	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
domres, domsub	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha) \\ E_2.type \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
ranres, ransub	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.type \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
upto	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{Z} \\ E_2.type \equiv \mathbb{Z} \end{array} \right\}$	$\mathbb{P}(\mathbb{Z})$
plus, minus, mul, div, mod, expn	$\left\{ \begin{array}{l} E_1.type \equiv \mathbb{Z} \\ E_2.type \equiv \mathbb{Z} \end{array} \right\}$	\mathbb{Z}

Table 4.1: Typing equations and resulting type for binary expressions.

$E.expr-unop$	\mathcal{E}	τ
uminus	$\{ E_1.type \equiv \mathbb{Z} \}$	\mathbb{Z}
converse	$\{ E_1.type \equiv \mathbb{P}(\alpha \times \beta) \}$	$\mathbb{P}(\beta \times \alpha)$
card	$\{ E_1.type \equiv \mathbb{P}(\alpha) \}$	\mathbb{Z}
pow, pow1	$\{ E_1.type \equiv \mathbb{P}(\alpha) \}$	$\mathbb{P}(\mathbb{P}(\alpha))$
union, inter	$\{ E_1.type \equiv \mathbb{P}(\mathbb{P}(\alpha)) \}$	$\mathbb{P}(\alpha)$
dom	$\{ E_1.type \equiv \mathbb{P}(\alpha \times \beta) \}$	$\mathbb{P}(\alpha)$
ran	$\{ E_1.type \equiv \mathbb{P}(\alpha \times \beta) \}$	$\mathbb{P}(\beta)$
prj1	$\{ E_1.type \equiv \mathbb{P}(\alpha \times \beta) \}$	$\mathbb{P}(\alpha \times \beta \times \alpha)$
prj2	$\{ E_1.type \equiv \mathbb{P}(\alpha \times \beta) \}$	$\mathbb{P}(\alpha \times \beta \times \beta)$
id	$\{ E_1.type \equiv \mathbb{P}(\alpha) \}$	$\mathbb{P}(\alpha \times \alpha)$
min, max	$\{ E_1.type \equiv \mathbb{P}(\mathbb{Z}) \}$	\mathbb{Z}

Table 4.2: Typing equations and resulting type for unary expressions.

expr-quant1: $E ::= L_1 P_1 E_1 [expr-quant]$

Let α be a fresh type variable in

$$L_1.itvvars = E.itvvars \cup \{\alpha\}$$

$$L_1.ityenv = E.ityenv$$

$$L_1.ityeqs = E.ityeqs$$

$$P_1.inherited = L_1.synthesized$$

$$E_1.inherited = P_1.synthesized$$

$$E.styvars = E_1.styvars$$

$$E.styenv = E_1.styenv$$

$$E.styeqs = E_1.styeqs \cup \mathcal{E}$$

$$E.type = \tau$$

where \mathcal{E} and τ are defined in the following table.

$E.expr-quant$	\mathcal{E}	τ
qunion, qinter	$\{ E_1.type \equiv \mathbb{P}(\alpha) \}$	$\mathbb{P}(\alpha)$
cset	\emptyset	$\mathbb{P}(E_1.type)$

expr-quant2: $E ::= E_1 P_1 [expr-quant]$

Let α be a fresh type variable in

$$E_1.itvvars = E.itvvars \cup \{\alpha\}$$

$$E_1.ityenv = E.ityenv$$

$$E_1.ityeqs = E.ityeqs$$

$$P_1.inherited = E_1.synthesized$$

$$E.styvars = P_1.styvars$$

$$E.styenv = P_1.styenv$$

$$E.styeqs = P_1.styeqs \cup \mathcal{E}$$

$$E.type = \tau$$

where \mathcal{E} and τ are defined in the following table.

$E.expr-quant$	\mathcal{E}	τ
qunion, qinter	$\{ E_1.type \equiv \mathbb{P}(\alpha) \}$	$\mathbb{P}(\alpha)$
cset	\emptyset	$\mathbb{P}(E_1.type)$

expr-bool: $E ::= P_1$

$$P_1.inherited = E.inherited$$

$$E.synthesized = P_1.synthesized$$

$$E.type = \text{BOOL}$$

expr-eset: $E ::= M$

$$M.inherited = E.inherited$$

$$E.synthesized = M.synthesized$$

$$E.type = \mathbb{P}(M.type)$$

expr-ident: $E ::= I_1$
 $I_1.inherited = E.inherited$
 $E.synthesized = I_1.synthesized$
 $E.type = I_1.type$

expr-atom: $E ::= [expr-lit]$
 Let α be a fresh type variable in
 $E.styvars = E.itvars \cup \{\alpha\}$
 $E.styenv = E.itenv$
 $E.styeqs = E.iteqs$
 $E.type = \tau$

where τ is defined in the following table.

$E.expr-lit$	τ
integer, natural, natural1	$\mathbb{P}(\mathbb{Z})$
bool	$\mathbb{P}(\text{BOOL})$
true, false	BOOL
emptyset	$\mathbb{P}(\alpha)$

expr-int: $E ::= [int-lit]$
 $E.synthesized = E.inherited$
 $E.type = \mathbb{Z}$

pattern: $Q ::= Q_1 Q_2$
 $Q_1.inherited = Q.inherited$
 $Q_2.inherited = Q_1.synthesized$
 $Q.synthesized = Q_2.synthesized$
 $Q.type = Q_1.type \times Q_2.type$

pattern-ident: $Q ::= I_1$
 $I_1.inherited = Q.inherited$
 $Q.synthesized = I_1.synthesized$
 $Q.type = I_1.type$

$$\begin{aligned}
\text{expr-list: } M &::= E_1 E_2 \dots E_n \\
E_1.inherited &= M.inherited \\
E_2.inherited &= E_1.synthesized \\
&\vdots \\
E_n.inherited &= E_{n-1}.synthesized \\
M.styvars &= E_n.itvars \\
M.styenv &= E_n.itenv \\
M.styeqs &= E_n.ityeqs \cup \left\{ \begin{array}{l} E_1.type \equiv E_2.type \\ E_2.type \equiv E_3.type \\ \vdots \\ E_{n-1}.type \equiv E_n.type \end{array} \right\} \\
M.type &= E_n.type
\end{aligned}$$

4.3.3 Examples

In this subsection, we present a few examples of the type-checking algorithm in action on various formulae.

Formula $x \in \mathbb{Z} \wedge 1 \leq x$. Figure 4.1 shows the dataflow for the type-checking of this formula. Each step of the type-checking algorithm is shown as a circled number, with edges relating them. The numbers appearing on the left of a node corresponds to the computation of inherited attributes, numbers on the right to the computation of synthesized attributes.

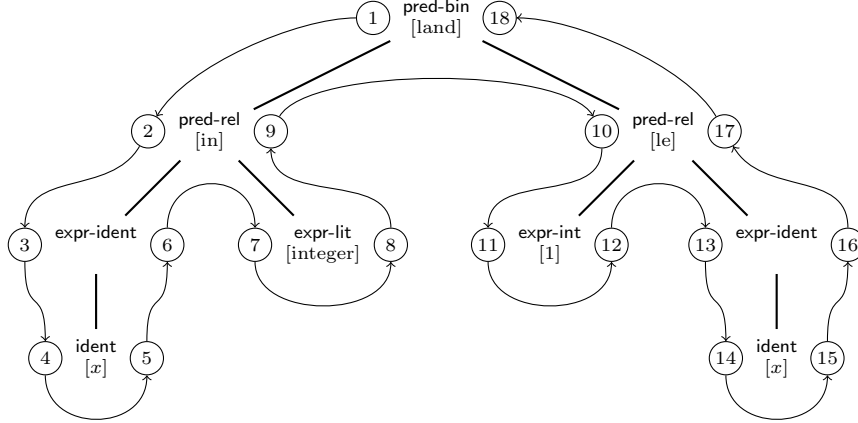


Figure 4.1: Type-check of formula $x \in \mathbb{Z} \wedge 1 \leq x$.

Assuming that the typing environment is initially empty, the initial computation at step 1 is:

$$1: \left\{ \begin{array}{l} itvars = \emptyset \\ itenv = \emptyset \\ ityeqs = \emptyset \end{array} \right.$$

Then, we process down the tree, adding a type variable at the \in operator:

$$2: \left| \begin{array}{l} ityvars = \emptyset \\ ityenv = \emptyset \\ ityeqs = \emptyset \end{array} \right. \quad 3, 4: \left| \begin{array}{l} ityvars = \{\alpha\} \\ ityenv = \emptyset \\ ityeqs = \emptyset \end{array} \right.$$

Examining the first occurrence of variable x , we find that it is not present in the environment, so we create a new type variable for it. This is then propagated in the tree:

$$5, 6: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \emptyset \\ type = \beta \end{array} \right. \quad 7: \left| \begin{array}{l} ityvars = \{\alpha, \beta\} \\ ityenv = \{x \mapsto \beta\} \\ ityeqs = \emptyset \end{array} \right. \quad 8: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \emptyset \\ type = \mathbb{P}(\mathbb{Z}) \end{array} \right.$$

We now reach the \in operator again, where we add our first type equations and propagate the attribute values:

$$9: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right. \quad 10, 11: \left| \begin{array}{l} ityvars = \{\alpha, \beta, \gamma\} \\ ityenv = \{x \mapsto \beta\} \\ ityeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right.$$

Continuing our traversal of the tree, we get:

$$12: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \end{array} \right\} \\ type = \mathbb{Z} \end{array} \right. \quad 13, 14: \left| \begin{array}{l} ityvars = \{\alpha, \beta, \gamma\} \\ ityenv = \{x \mapsto \beta\} \\ ityeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right.$$

We now reach the second occurrence of variable x and, now, it is present in the typing environment, so we just read its type from there, and propagate it:

$$15, 16: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \end{array} \right\} \\ type = \beta \end{array} \right.$$

Reaching operator \leq , we add two new typing equations and propagate them to the root:

$$17, 18: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma\} \\ styenv = \{x \mapsto \beta\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) \equiv \mathbb{P}(\alpha) \\ \mathbb{Z} \equiv \mathbb{Z} \\ \beta \equiv \mathbb{Z} \end{array} \right\} \end{array} \right.$$

In the end, we obtain a system of four typing equations with two type variables. This system is uniquely satisfiable by taking $\alpha = \mathbb{Z}$ and $\beta = \mathbb{Z}$. Hence, the formula type checks. Moreover, its resulting typing environment is $\{x \mapsto \mathbb{Z}\}$.

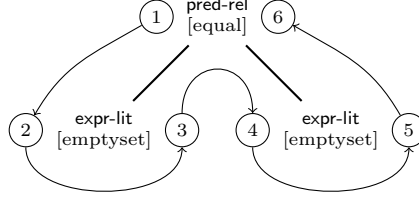


Figure 4.2: Type-check of formula $\emptyset = \emptyset$.

Formula $\emptyset = \emptyset$. The type-checking dataflow for this formula is given in Figure 4.2.

The attribute values computed by the algorithm are (supposing that the initial typing environment is empty):

1: $\begin{cases} ityvars = \emptyset \\ ityenv = \emptyset \\ ityeqs = \emptyset \end{cases}$	2: $\begin{cases} ityvars = \{\alpha\} \\ ityenv = \emptyset \\ ityeqs = \emptyset \end{cases}$	3: $\begin{cases} styvars = \{\alpha, \beta\} \\ styenv = \emptyset \\ styeqs = \emptyset \\ type = \beta \end{cases}$
4: $\begin{cases} ityvars = \{\alpha, \beta\} \\ ityenv = \emptyset \\ ityeqs = \emptyset \end{cases}$	5: $\begin{cases} styvars = \{\alpha, \beta, \gamma\} \\ styenv = \emptyset \\ styeqs = \emptyset \\ type = \gamma \end{cases}$	6: $\begin{cases} styvars = \{\alpha, \beta, \gamma\} \\ styenv = \emptyset \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \alpha, \\ \gamma \equiv \alpha \end{array} \right\} \end{cases}$

In the end, we obtain a system of two typing equations with three typing variables. This system is satisfiable, but not uniquely. Hence formula $\emptyset = \emptyset$ does not type-check.

Formula $x \subseteq S \wedge \emptyset \subset x$. The type-checking dataflow for this formula is given in Figure 4.3.

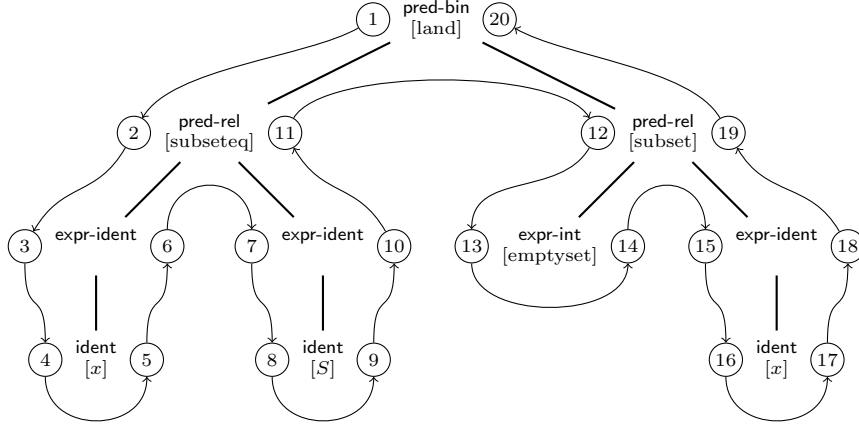


Figure 4.3: Type-check of formula $x \subseteq S \wedge \emptyset \subset x$.

Here, we assume that variable S denotes a given set. Thus, our initial typing environment is $\{S \mapsto \mathbb{P}(S)\}$. The attribute values computed by the

type-checking algorithm are:

$$\begin{array}{ll}
1, 2: \left| \begin{array}{l} ityvars = \emptyset \\ ityenv = \{S \mapsto \mathbb{P}(S)\} \\ ityeqs = \emptyset \end{array} \right. & 3, 4: \left| \begin{array}{l} ityvars = \{\alpha\} \\ ityenv = \{S \mapsto \mathbb{P}(S)\} \\ ityeqs = \emptyset \end{array} \right. \\
5, 6: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \emptyset \\ type = \beta \end{array} \right. & 7, 8: \left| \begin{array}{l} ityvars = \{\alpha, \beta\} \\ ityenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ ityeqs = \emptyset \end{array} \right. \\
9, 10: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \emptyset \\ type = \mathbb{P}(S) \end{array} \right. & 11: \left| \begin{array}{l} styvars = \{\alpha, \beta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right. \\
12: \left| \begin{array}{l} ityvars = \{\alpha, \beta\} \\ ityenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ ityeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right. & 13: \left| \begin{array}{l} ityvars = \{\alpha, \beta, \gamma\} \\ ityenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ ityeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right. \\
14: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma, \delta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \\ type = \mathbb{P}(\delta) \end{array} \right. & 15, 16: \left| \begin{array}{l} ityvars = \{\alpha, \beta, \gamma, \delta\} \\ ityenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ ityeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \end{array} \right. \\
17, 18: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma, \delta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha) \end{array} \right\} \\ type = \beta \end{array} \right. & 19, 20: \left| \begin{array}{l} styvars = \{\alpha, \beta, \gamma, \delta\} \\ styenv = \left\{ \begin{array}{l} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{array} \right\} \\ styeqs = \left\{ \begin{array}{l} \beta \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(\delta) \equiv \mathbb{P}(\gamma), \\ \beta \equiv \mathbb{P}(\gamma) \end{array} \right\} \end{array} \right.
\end{array}$$

In the end, we obtain a system of four typing equations with four typing variables. This system is uniquely satisfiable taking $\alpha = \gamma = \delta = S$ and $\beta = \mathbb{P}(S)$. Hence formula $x \subseteq S \wedge \emptyset \subset x$ type-checks and the resulting typing environment is $\{S \mapsto \mathbb{P}(S), x \mapsto \mathbb{P}(S)\}$.

Formula $x = \text{TRUE}$. The type-checking dataflow for this formula is given in Figure 4.4 on the following page.

Assuming that initially x denotes an integer (non empty initial typing environment), we obtain the following values for attributes:

$$\begin{array}{ll}
1: \left| \begin{array}{l} ityvars = \emptyset \\ ityenv = \{x \mapsto \mathbb{Z}\} \\ ityeqs = \emptyset \end{array} \right. & 2, 3: \left| \begin{array}{l} ityvars = \{\alpha\} \\ ityenv = \{x \mapsto \mathbb{Z}\} \\ ityeqs = \emptyset \end{array} \right.
\end{array}$$

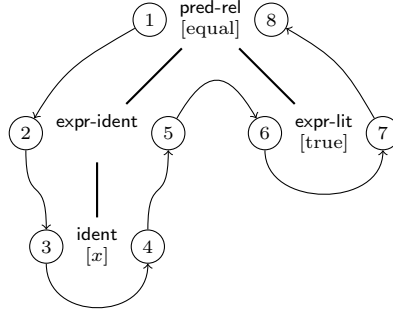


Figure 4.4: Type-check of formula $x = \text{TRUE}$.

$$\begin{array}{ll}
 4, 5: \left\{ \begin{array}{l} styvars = \{\alpha\} \\ styenv = \{x \mapsto \mathbb{Z}\} \\ styeqs = \emptyset \\ type = \mathbb{Z} \end{array} \right. & 6: \left\{ \begin{array}{l} ityvars = \{\alpha\} \\ ityenv = \{x \mapsto \mathbb{Z}\} \\ ityeqs = \emptyset \end{array} \right. \\
 7: \left\{ \begin{array}{l} styvars = \{\alpha\} \\ styenv = \{x \mapsto \mathbb{Z}\} \\ styeqs = \emptyset \\ type = \text{BOOL} \end{array} \right. & 8: \left\{ \begin{array}{l} styvars = \{\alpha\} \\ styenv = \{x \mapsto \mathbb{Z}\} \\ styeqs = \left\{ \begin{array}{l} \mathbb{Z} \equiv \alpha \\ \text{BOOL} \equiv \alpha \end{array} \right\} \end{array} \right.
 \end{array}$$

In the end, we obtain a system of two typing equations with one typing variable. This system is not satisfiable, therefore the formula does not type-check (remember that we initially assumed that variable x denotes an integer). If the initial typing environment would have been empty, then the formula would type-check.

5 Dynamic Checking

Static checks are not enough to ensure that a formula is meaningful. For instance, expression $x \div y$ passes all the static checks described above, nevertheless it is meaningless if y is zero. The aim of dynamic checking [2, 3] is to detect these kind of meaningless formulas. This is done by generating (and then proving) some well-definedness lemma.

The rest of this chapter specifies how to produce these well-definedness lemmas. This is done by specifying a WD operator that takes a formula as argument and the result of which is the well-definedness lemma of that formula.

5.1 Predicate Well-Definedness

Table 5.1 on the next page specifies the WD operator for predicates. In that table, letters P and Q denote arbitrary predicates, letters E and F denote expressions, and letter L denotes a list of identifiers.

5.2 Expression Well-Definedness

Tables 5.2 on page 42 and 5.3 on page 43 specify the WD operator for expressions. In these tables, letter P denotes an arbitrary predicate, letters E and F denote expressions, letter Q denotes a lambda pattern, letter L denotes a list of identifiers, letter I denotes an identifier, letter n denotes a literal integer. We also denote by \mathcal{F}_E the list of the free variables that appear in expression E (that is $E.free$) and by \mathcal{F}_Q the list of the free variables that appear in pattern Q . Finally, letter x denotes a fresh variable (that is a variable that does not occur free in the formula for which we compute the well-definedness lemma).

Predicate	WD Lemma
$P \wedge Q \quad P \Rightarrow Q$	$\text{WD}(P) \wedge (P \Rightarrow \text{WD}(Q))$
$P \vee Q$	$\text{WD}(P) \wedge (P \vee \text{WD}(Q))$
$P \Leftrightarrow Q$	$\text{WD}(P) \wedge \text{WD}(Q)$
$\neg P$	$\text{WD}(P)$
$\forall L.P \quad \exists L.P$	$\forall L.\text{WD}(P)$
$\top \quad \perp$	\top
$\text{finite}(E)$	$\text{WD}(E)$
$E = F \quad E \neq F$ $E \in F \quad E \notin F$ $E \subset F \quad E \not\subset F$ $E \subseteq F \quad E \not\subseteq F$	$\text{WD}(E) \wedge \text{WD}(F)$

Table 5.1: WD lemmas for predicates.

Expression	WD Lemma
$F(E)$	$\text{WD}(F) \wedge \text{WD}(E) \wedge E \in \text{dom}(F) \wedge F^{-1}; (\{E\} \triangleleft F) \subseteq \text{id}(\text{ran}(F))$
$E[F] \quad E \mapsto F$ $E \leftrightarrow F \quad E \leftrightarrow F$ $E \leftrightarrow\!\!\!\rightarrow F \quad E \leftrightarrow\!\!\!\rightarrow F$ $E \mapsto\!\!\!\rightarrow F \quad E \mapsto\!\!\!\rightarrow F$ $E \mapsto\!\!\!\rightarrow\!\!\!\rightarrow F \quad E \mapsto\!\!\!\rightarrow\!\!\!\rightarrow F$ $E \mapsto\!\!\!\rightarrow\!\!\!\rightarrow\!\!\!\rightarrow F \quad E \mapsto\!\!\!\rightarrow\!\!\!\rightarrow\!\!\!\rightarrow F$ $E \cup F \quad E \cup F$ $E \cap F \quad E \cap F$ $E \times F \quad E \times F$ $E \parallel F \quad E \parallel F$ $E ; F \quad E ; F$ $E \triangleleft F \quad E \triangleleft F$ $E \triangleright F \quad E \triangleright F$ $E .. F \quad E + F$ $E - F \quad E * F$	$\text{WD}(E) \wedge \text{WD}(F)$
$E \div F \quad E \bmod F$	$\text{WD}(E) \wedge \text{WD}(F) \wedge F \neq 0$
$E \wedge F$	$\text{WD}(E) \wedge 0 \leq E \wedge \text{WD}(F) \wedge 0 \leq F$
$-E \quad E^{-1}$ $\mathbb{P}(E) \quad \mathbb{P}_1(E)$ $\text{dom}(E) \quad \text{ran}(E)$ $\text{prj}_1(E) \quad \text{prj}_2(E)$ $\text{id}(E) \quad \text{union}(E)$	$\text{WD}(E)$
$\text{card}(E)$	$\text{WD}(E) \wedge \text{finite}(E)$
$\text{inter}(E)$	$\text{WD}(E) \wedge E \neq \emptyset$
$\text{min}(E)$	$\text{WD}(E) \wedge E \neq \emptyset \wedge (\exists b \cdot \forall x \cdot x \in E \Rightarrow b \leq x)$
$\text{max}(E)$	$\text{WD}(E) \wedge E \neq \emptyset \wedge (\exists b \cdot \forall x \cdot x \in E \Rightarrow x \leq b)$

Table 5.2: WD lemmas for binary and unary expressions.

Expression	WD Lemma
$\lambda Q \cdot P \mid E$	$\forall \mathcal{F}_Q \cdot \text{WD}(P) \wedge (P \Rightarrow \text{WD}(E))$
$\bigcup \begin{matrix} L \cdot P \mid E \\ \{L \cdot P \mid E\} \end{matrix}$	$\forall L \cdot \text{WD}(P) \wedge (P \Rightarrow \text{WD}(E))$
$\bigcup \begin{matrix} E \mid P \\ \{E \mid P\} \end{matrix}$	$\forall \mathcal{F}_E \cdot \text{WD}(P) \wedge (P \Rightarrow \text{WD}(E))$
$\bigcap L \cdot P \mid E$	$\begin{matrix} (\forall L \cdot \text{WD}(P) \wedge (P \Rightarrow \text{WD}(E))) \\ \wedge (\exists L \cdot P) \end{matrix}$
$\bigcap E \mid P$	$\begin{matrix} (\forall \mathcal{F}_E \cdot \text{WD}(P) \wedge (P \Rightarrow \text{WD}(E))) \\ \wedge (\exists \mathcal{F}_E \cdot P) \end{matrix}$
$\text{bool}(P)$	$\text{WD}(P)$
$\{E_1, E_2, \dots, E_n\}$	$\text{WD}(E_1) \wedge \text{WD}(E_2) \wedge \dots \wedge \text{WD}(E_n)$
$\begin{matrix} I & \mathbb{Z} \\ \mathbb{N} & \mathbb{N}_1 \\ \text{BOOL} & \text{TRUE} \\ \text{FALSE} & \emptyset \\ n \end{matrix}$	\top

Table 5.3: WD lemmas for other expressions.

Bibliography

- [1] Abrial, J.-R. (1996). *The B-Book. Assigning Programs to Meanings*. Cambridge University Press.
- [2] Abrial, J.-R and Mussat, L. (2002). *On Using Conditional Definitions in Formal Theories*. In D. Bert et al. (Eds), *ZB2002: Formal Specification and Development in Z and B*, LNCS 2272, pp. 242–269, Springer-Verlag.
- [3] Burdy, L. (2000). *Traitement des expressions dépourvues de sens de la théorie des ensembles. Application à la méthode B*. Thèse de doctorat. Conservatoire National des Arts et Métiers.
- [4] The Unicode Consortium (2003). *The Unicode Standard 4.0*. Addison-Wesley.

Tutorials for RODIN

October 26, 2007

Introduction

This tutorial should provide the user with a tour through the most important functionalities of RODIN, so that he gets a understanding of how the program works. The tutorial is divided into 5 sections:

In the first section, a very simple project is created from scratch. The essential steps in working with components are illustrated here.

The second section provides an example that shows how events of different machines can be connected together. It also gives an introduction on working with the prover

After section 1 and 2, the user should have developed a feel of the basic windows of RODIN, and when they are needed. He might want to combine the two default perspectives into one. Section 3 explains how this can be done. This section may be omitted by users who feel comfortable switching between two perspectives.

Section 4 shows how to use apply reasoning on models in the prover.

Section 5 shows a proof in a mathematical setting, and then provides a few examples, on which the user can work on by himself. In these proofs, everything except the basic rewrite rules has to be done by the user.

Files

For this tutorial, you will be needing 4 example files. They are:

- **Celebrity.zip**, which will be needed in section 2.
- **Doors.zip**, which contains the model that is used in section 4.
- **Closure.zip**. This is the model on which you perform proofs in section 5.

- **Galois.zip**. This model is not really needed, but serves as an explanation to why the proof in section 5 works.

1 A First Example: The Birthday Book

To begin with, let us start with a simple model of a “birthday book”, similar to the one in the Z Notation Reference Manual¹. This little book is a simple tool that can be used to keep birthdays of different people. All we can do with it is writing people’s birthdays into it. So, the initial model only has one event. We create the model as follows:

1.1 The Event-B Perspective

1. Start RODIN. You should be directed automatically to the Event-B perspective. If this does not happen, you will have to change to this perspective manually (Eclipse help describes how this is done).
2. Create a new Project as described in section 1.3 of the Manual. Name the project **BirthdayBook**.
3. Create a new Context Component named **BirthdayBook_C0** in the Project you created. This can be done similarly to creating the new project. (See also Section 1.8 of the Manual)
4. In this model, we are not interested in the specific structure of the date. All we want this context to contain are two carrier sets, one for dates and one for people. We will call them **DATE** and **PERSON**. You can add them to the context as described in section 2.1 of the manual. Now either press Ctrl+S or click on the save Icon to save the context.
5. Now, we need to create a Machine Component. Proceed as you did to create the Context Component, just choose “Machine” this time. Give the Machine Component the name **BirthdayBook_0**. Once the component is created, a window with the machine’s dependencies appears. Add **BirthdayBook_C0** to the seen contexts so that you can access the carrier sets.
6. We need a variable that reflects the contents of the book. We call it “birthday”. The following information on the variable can be entered into the New Variable Wizard (Manual Section 3.2.1), which can be accessed by an icon on the tool bar. Since every person has at most one entry in the birthday book, but not every person has an entry in it, “birthday” should be a partial function from **PERSON** to **DATE**. At initialization, we want the book to be empty. (The symbol for partial functions can be written by typing in \mapsto . Section 8 of the Manual contains all on how to write Event-B symbols in ASCII.)

¹The Z Notation Reference Manual can be found at <http://spivey.oriel.ox.ac.uk/mike/zrm/index.html>

7. Last, we create the event. Open the New Event Wizard (as seen in Section 3.5.1 of the Manual) on the top tool bar. Name the new Event `AddBirthday`. It has two parameters, `p` and `d`, where `p` is a `PERSON` and `d` is a `DATE` (enter these as guards). As action, write $birthday := birthday \cup \{p \mapsto d\}$ (Remember, Section 8 of the manual explains how to write Event-B symbols). Now, save the machine. If you have done everything correctly, the type checker should not return anything to the problems window at the bottom of the screen. It appears that we have successfully created a model of the birthday book. But have we?

1.2 The Proving Perspective

Now, we switch to the proving perspective. You see several windows on the screen:

- The Proof Obligation Explorer (Section 5 of the Manual): Here you can browse through proof obligations. As you can see, `BirthdayBook_C0` has no proof obligation and `BirthdayBook_0` has two. The A in the icons of the obligations means that the automatic prover attempted both obligations. The green icon next to the first proof obligation indicates that it already has been proved, and the red icon next to the second proof obligation tells that the proof is not yet completed. Proof obligations can have three colors, red, green and blue. The blue color means that the proof has been reviewed, meaning that it has been discharged without proof by the user. Click on the second prove obligation to display the proof. This fills a couple of other windows.
- The Proof Tree (Section 6.2 of the Manual): Here you see a tree of the proof that you have done so far and your current position in it. By clicking in the tree, you can navigate inside the proof. Currently, you have not started with the proof yet, so there is no new place to move to.
- The Proof Control (Manual, section 6.4): This is where you perform interactive proofs.
- The Selected Hypothesis window (Manual, section 6.7): The hypothesis that are currently being used for the proof are displayed here. You can add hypothesis into it from the Search Hypothesis window and from the Cached Hypothesis window (See section 6.4 in the Manual on how to open these windows).
- The goal window (Manual, section 6.3): This window shows what needs to be proved at the current position inside the proof tree. Currently we need to show that birthday is still a partial function from `PERSON` to `DATE` if it is extended by an entry.

There is no way to prove the goal if a birthday is already entered into the book for a certain person. So, our event needs an additional guard that restricts `p` to people for whom there is no entry yet in the book. This can be done in the proving perspective. Just switch to the machine and add the additional guard ($p \notin \text{dom}(\text{birthday})$) using

the editor (On adding guards: Manual, section 3.5.2). If you save the document now, you will see that the auto-prover can conclude.

Congratulations! You have built your first model with RODIN.

2 The Celebrity Problem

The next model that you will work with is the so-called celebrity problem. In the setting for this problem, we have a “knows” relation. This relation is defined so that

- no one knows himself,
- the celebrity knows nobody, but
- everybody knows the celebrity.

The goal is to find the celebrity. The provided development, once completed, yields a linear-time algorithm for the problem.

2.1 Modeling

1. Make sure that you have no existing Project named **Celebrity**. If you do, then rename it (Section 1.7 in the Manual). Then import the archive file **Celebrity.zip**. For this, choose “Import...” in the File menu, and then select “Existing Projects into Workspace”. Then, select the according archive file. (Read more on importing projects in section 1.6 of the Manual)
2. The tool takes a few seconds to extract and load all the files. Once it is done, it shows that there are a few problems with this project. In the first part of this section, our goal is to fix these problems and conclude the proofs.
3. First of all, we take a look at the error. It states that an event called “celebrity” is not refined. Double-click on the error in the Problem Window to open the **Celebrity_1** machine. If you look at its events (by pressing the “Events” tab), you can see that it actually does have a “celebrity” event. The problem is that it is not declared as a refinement. In order to do so, right-click on the event and select “New Refine Event”. This declares that the event is a refinement of an event with the same name in the abstract machine. As this is the case here, we can now save the project and the error disappears. (If the event in the abstract machine had a different name, you would have to edit it. For more on this issue, see Section 3.9.1 of the Manual).
4. Next, we deal with the warnings. The three remaining warnings state that witnesses are missing. In any abstract event that uses parameters, if the concrete event has no parameter with the same name, the tool needs a witness so that it notices what value the parameter should take. Witnesses are also needed for variables

that have a nondeterministic assignment in an abstract event and do not appear in the concrete model. (See also Section 3.9.4 of the Manual) To create the witness, double-click on the warning to open the concrete model (here **Celebrity_2**). Then, right-click on the “celebrity” event and select “New Witness”.

5. An empty witness has been created, which we need to fill. Its name will have to be x if we want it to be a witness for the parameter x . Next, for the content. If we switch between the two machines (either by pressing Ctrl+F6 or by clicking on their tabs), we see that the abstract event has the assignment $r := x$, while the concrete one has the assignment $r := b$. So, $x = b$ is the witness. Edit the Details and save the file. One warning will disappear, two to go.
6. Try completing the other two witnesses on your own. A hint: Both witnesses are simple equalities, and both can be found by comparing the third guard of the abstract event with the second guard of the concrete one. Remember to give the witness the name of the variable it stands for. If you completed this step correctly, there should be no warning, info or error left on the Problems window.

2.2 Proving

1. All we have to do now is prove. Switch to Proving Perspective. Browsing around in the Obligation Explorer (Section 5 of the Manual shows you how), you can see that the auto-prover did quite a good job. If you have chosen the witnesses correctly, all except for five proofs already should be completed. Except for the last one of them, all of them could be proved with a different external prover, but in order to learn a few new techniques, we will prove them with the p0 prover.
2. Let’s start with the proof in **Celebrity_0**. Select the proof by clicking on it. What you need to prove is that P is not empty. Enter P in the proof control and open the Search Hypothesis Window. Like this, you get all hypothesis that have P in them. We need to find one that works toward the goal. $c \in P$ does that, so add it to the selected hypothesis (Section 6.7 of the manual explains how) and click on p0 in the Proof Control. The proof succeeds.
3. Next, we look at **celebrity/act1/SIM** of **Celebrity_1**. Here we need to prove $x = c$. You have no hypothesis about c selected, so look for them, like you did for hypothesis with P in the last proof. This time, you can see that $c \in Q$ is all you need to conclude the proof. So add it to the selected hypothesis and the proof will succeed using p0.
4. **remove_1/inv2/INV**, is a little more complex. In order to prove the statement, it suffices to prove $x \neq c$. so type this into the proof control and press the Add Hypothesis button (ah). Now, press p0 until it does not get you any further. Now, try selecting the right hypothesis by yourself in order to complete the proof. If you cannot find it, you may also just select all hypotheses.

5. In order to move to the next proof obligation, you may also use the Next Undischarged PO button of the Proof Control. The next proof can be solved the same way as the last one. You just need to add at least two hypotheses this time.
6. In the proof in `Celebrity_2`, you have to fill in an existential quantifier. First, look in the list of hypothesis if you find any variable that is in P , and select that hypothesis. Then, instantiate b' and R' correctly (For instance, if you want to instantiate b' with c , then $P \setminus \{c\}$ is a good choice for R') by typing the instantiations in the Goal Window and then clicking on the red existential quantifier. Now, all open branches of the proof tree can be proved with $p0$. After this, we have completed all the proofs, and the model is ready for use.

3 Customizing a perspective suitable for RODIN

So far, you needed two different perspectives to work with RODIN. But really, it is possible to work with only one perspective. In this section, we try to customize a perspective so that we do not need any other. If you have experience with customizing Eclipse perspectives, you may only want to read the next paragraph which contains a few thoughts about a good perspective for RODIN.

As a start, we should think about what we want the perspective to look like. The proving perspective already is pretty nice. We just could use little bit more editing space and the windows of the Event-B perspective. To create more space, we could move all windows that currently are on both sides of the editing area onto one side, as they never really need to be used simultaneously. For even a bit more space, we could dock all of these windows onto the so-called Fast View Bar, so that they disappear when they are not needed. Like that, there should be enough space to even work split-screen with different components, for example, we could have an abstract machine on one side of the editing surface, and the concrete machine on the other.

Most of the perspective editing is simply drag and drop. First of all, you need to find the Fast View Bar. Usually, it is at the bottom end of the Eclipse window. But it also can be on the side or hidden inside the Shortcut Bar. For our purposes, it probably is best to have it on the right side of the screen. Place it there by dragging it with the mouse. Now, add some items to it. To do that, press the New Fast View button on the bar. It might be useful to leave the Goal, the Problems and the Proof Control window at the bottom of the screen, as you may want them to stay open while editing. A good choice for the Fast view may be:

- Project Explorer,
- Obligation Explorer,
- Search Hypothesis,

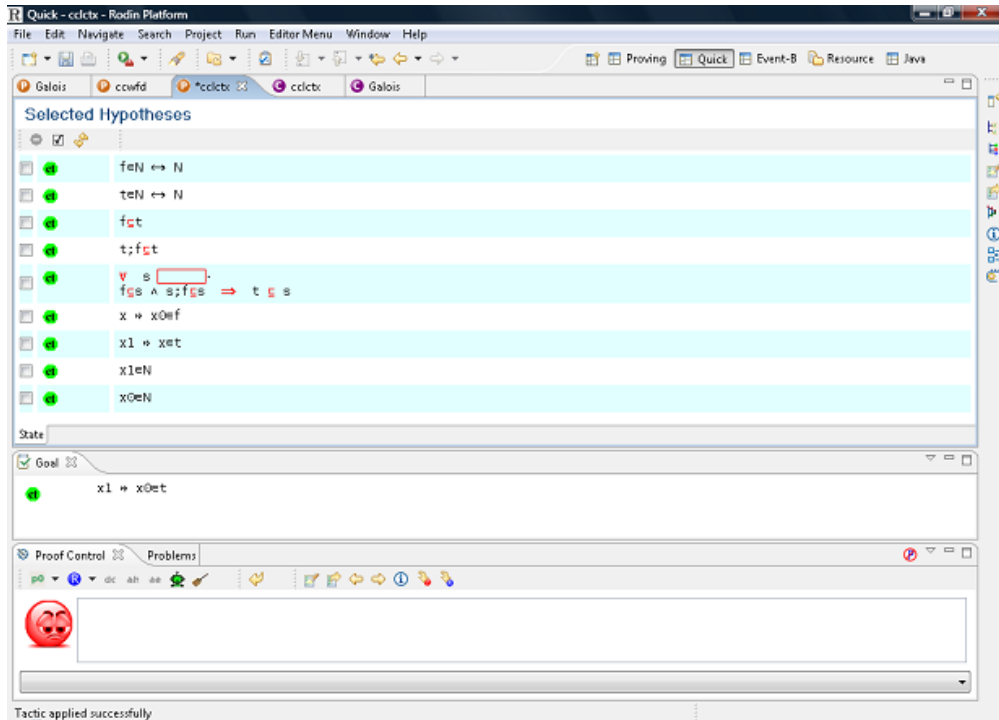


Figure 1: Our self-made Quick perspective

- Cache Hypothesis,
- Proof Tree,
- Proof Information
- Progress Window.

All of the windows that you cannot create directly when clicking on the New Fast View Bar can be found in Others/General. Once you are done, the window should look like in Figure 1. Click on “Save Perspective As...” in the Window menu to save the perspective.

4 Location Access Controller

In this section, we will take a closer look at a few more complex proofs. For this, we will use the model of the Location Access Controller, which is saved in the `doors.zip` project archive. Explanations on the model can be found in the chapter entitled “Location Access Controller” of the Event-B book. Our task is to develop the proofs for deadlock freeness of the initial model and of the first refinement.

Import `doors.zip`. Looking through it, you will see that everything already has been proved. This is true, however, RODIN does not do any deadlock freeness proof yet, so you will have to add them yourself.

4.1 Initial Model

First, let us look at the initial model, consisting of `doors_ctx1` and `doors_0`. There are two carrier sets in the model, one for people and one for locations. Then, there is a location called *outside* and a relation *aut* which reflects where people are allowed to go. Everyone is allowed to *outside*. The model only has one event, which changes the location of a person. So, proof of deadlock freeness means proving, that someone can always change room.

1. Add a new theorem to `doors_0` called “DLF” and change the predicate so that it is the disjunction of all guards (since there is only one guard here, it would be $\exists p, l \cdot (p \mapsto l \in \text{aut} \wedge \text{sit}(p) \neq l)$).
2. Save the machine. You will see that the autoprover fails to prove the theorem (DLF/THM). Let us analyze whether this is an inconsistency in the model. In order to succeed with the proof, we need a tuple $p \mapsto l$ that is in *aut*, but not in *sit*. Searching the hypotheses, we find AXM4 of `doors_ctx1`, which states that there is a room l , where everyone is permitted to. So, for every person p in P , $p \mapsto l$ and $p \mapsto \text{outside}$ is in *aut*. Since these are different, at least one of them cannot be in the function *sit*. Now, all we would need to prove is that P is nonempty. This holds, as carrier sets always are nonempty, but is a bit hard to derive. Add the hypothesis $P \neq \emptyset$ using the `ah` button. Then, click on the Auto Prover button (The button with a robot on it). Other provers do not work here. After successfully adding the hypothesis, we can conclude the proof as follows:
3. Add AXM4 (the one with the existential quantifier) to the selected hypothesis. You see that it automatically is instantiated as l . Next, click on the “ \neg ” of $\neg P = \emptyset$ in the search hypothesis window. This creates a selected hypothesis $x \in P$. We can now instantiate p in the goal with x .
4. In order to instantiate l , we need a case distinction. Type $\text{sit}(x) = l$ this into the proof control and click on Case Distinction (dc) to look at the two cases $\text{sit}(x) = l$ and $\text{sit}(x) \neq l$. Before starting with the cases, the prover now wants you to prove that $x \in \text{dom}(\text{sit})$. This can be done with `p0`, as *sit* is a total function. In the first case x is situated in l , so it cannot be in *outside*. So, you can instantiate l with *outside*. In order to prove that x is allowed to *outside*, you will need to select the hypothesis $P \times \text{outside} \subseteq \text{aut}$. Then you can finish this case with `p0`. If you look at the proof tree, you see that you now have reached the other branch of the case distinction. In this case, you can simply instantiate l with l , as x is not situated there. Finally, click on `p0` to finish the proof.

4.2 First Refinement

Now we get to the a bit harder proof: The deadlock freeness proof for the first refinement. There is not much that has changed. The constant *com* has been added in order to describe which rooms are connected. Additionally, we have a constant *exit*, which will be explained later.

1. Open `door_1` and add a theorem called DLF stating $\exists q, m. (q \mapsto m \in aut \wedge sit(q) \mapsto m \in com)$ to it, then save the file. Once again, the prover fails to prove deadlock freeness automatically.
2. At the beginning of the proof, there are no selected hypothesis at all. So we need to select a few. The old deadlock freeness theorem will be useful, AXM7 of `doors_ctx2` too. To begin with, we try to avoid using *exit*, as we want to keep things as simple as possible. But since *sit* and *aut* are inside the theorem, we also are likely to need $sit \subseteq aut$, $sit \in P \rightarrow L$ and $aut \in P \leftrightarrow L$.
3. Once again, the prover automatically rewrites the existential quantifiers in the hypotheses. We now look at the proof. There is an easy case if $sit(p) = outside$. Solve it.
4. For the other case, we will need the notion of *exit*. The axioms about *exit* state that
 - (AXM 3) Every room except the outside has exactly one exit.
 - (AXM 4) An exit must be a room that communicates with the current one.
 - (AXM 5) A chain of exits leads to the outside without any cycles or infinite paths.
 - (AXM 6) Everyone allowed in a room is allowed to go through its exit.

In our proof, we still need to show that anyone who is not outside can walk through a door. For this, AXM 5 is useless, so we add all hypothesis containing *exit* except for AXM 5. Now we only need to instantiate *q* and *m* correctly and concluding the proof should not be too hard. For *q*, the choice *p* is obvious. But it is not quite as easy for *m*. Expressed in language, *m* must be the room behind the exit door of the room that *p* is currently in. Try translating this into set theory. But do not worry if you get it wrong. You can still go back in the proof by right-clicking at the desired point in the proof tree and choosing prune.

5 Mathematical proofs

By now, you should know how to create and edit models, and how to do simple proofs with help of the predicate prover. In this section, we will look at more complex proofs in mathematical settings. For this, it is advisable for you to have some knowledge of the theory, but you can also see this as a pure proving exercise. First, we try to perform a

proof without the predicate prover in order to get familiar with all the tools. Then, you can try proving three further theorems by yourself.

5.1 First proof on Transitive Closure

1. Open the “Closure” project. it is a simple mathematical model, in which f is a binary relation. The axiom defines t so that it is the transitive closure of f . You will have to prove that $t; t \subseteq t$.
2. For that, we have to instantiate the s of $\forall s \cdot (f \subseteq s \wedge s; f \subseteq s \Rightarrow t \subseteq s)$ with $(N \times N) \setminus (t \sim; ((N \times N) \setminus t))$. This instantiation is a Galois transform. The project **Galois** shows the properties of galois transforms. Simply said, these transforms are a sort of inverse for the subset relation between relations. In the proof, this is useful, as we can easily derive the goal from the implication (Because of the main property of the Galois transform, $t \subseteq s$ is equivalent to $t; t \subseteq t$). The first part of the condition of the axiom also becomes easy to prove, as $f \subseteq s$ is equivalent to $t; f \subseteq t$. However, the second part of the condition, $s; f \subseteq s$ does not have such an easily provable equivalence. We will just have to believe that the instantiation also works here.
3. If you want to check that the instantiation really does work, you can use the predicate prover now and it will succeed. Afterwards, prune the predicate prover step so that you can do the proof yourself.
4. If the instantiation succeeded, you now should have a new, quite lengthy new hypothesis which has the form $p1 \wedge p2 \Rightarrow p3$, where $p1$, $p2$ and $p3$ are predicates. This hypothesis will help us to split the proof into three parts. First, we will prove $p1$, then $p2$, and last, we will derive our goal out of $p3$. To split the proof, click on the red arrow of the implication and choose “Apply Modus Ponens using this hypothesis”. If you now look at the pending subgoals by using the “Next Pending Subgoal” button in the Proof Control, you will see that the proof has been split into the three described parts above.
5. We will start off proving the first subgoal $f \subseteq (N \times N) \setminus (t \sim; ((N \times N) \setminus t))$. First of all, we need to translate the statement a bit more towards predicate logic. Click on the subset symbol in the goal and choose the first option (Remove inclusion in goal). As described, this translates the inclusion into predicate logic. If you look at the proof tree, you will see that the prover automatically has done another two steps, removing the quantifier and then the implication in the goal.
6. Now, you will need to completely translate the goal into predicate logic. For this, click on any red symbol that appears in the proof goal until there is none left. Should you have any tilde operators or cross products in the selected hypothesis, then it will not hurt to translate them, too. To do that, click on the red “ \in ” next to then. In the end, your proof should look like in Figure 2.

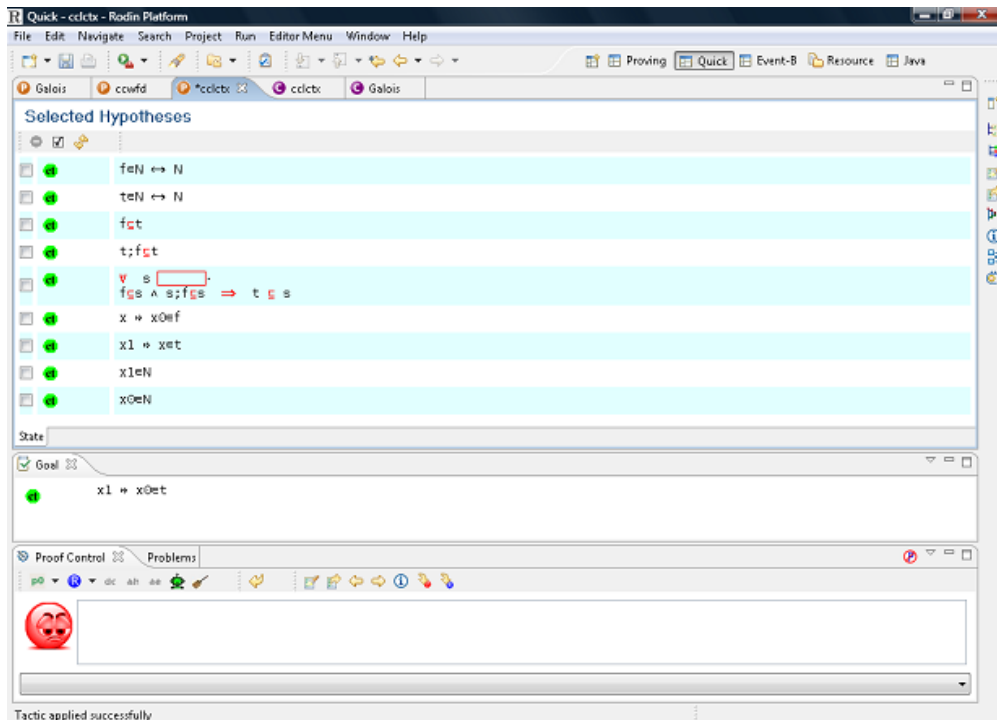


Figure 2: The proof after step 6

7. As the goal appears to be provable from $t; f \subseteq t$, $x \mapsto x0 \in f$ and $x1 \mapsto x \in t$, remove all the other hypotheses, if you like to keep your workbench tidy. This can be done by choosing the three hypotheses and then clicking the “Inverse selection” button first and then the “Remove hypotheses” button.
8. To prove the current goal, we first prove that $x1 \mapsto x0 \in (t; f)$. From that, $x1 \mapsto x0 \in t$ follows, because of $t; f \subseteq t$. Add the hypothesis $x1 \mapsto x0 \in (t; f)$ by entering it in the proof control and pressing the (ah) button. Once more, the proof will branch. For the first branch, click on the red “ \in ” symbol in the Proof Goal window to translate the goal into predicate calculus. You will then need to instantiate a variable, but the choice should be obvious if you look at the selected hypotheses. The second part of the proof can be solved by removing the “ \subseteq ” in $t; f \subseteq t$ and then instantiating correctly.
9. For the time being, we do not yet want to prove the second subgoal, but proceed with the last one. So review this subgoal by pressing the blue button in the proof control. The prover automatically skips to the next subgoal.
10. The proof of the third subgoal will only require the hypothesis $t \subseteq (N \times N) \setminus (t \sim ; ((N \times N) \setminus t))$. So you can remove all the others. Remove the inclusion in the hypothesis and in the goal in order to transform them towards predicate calculus. Then, click on the “ \in ” of the hypothesis $x \mapsto x0 \in t; t$ in order to introduce $x1$.

You can now instantiate x with $x1$ and $x0$ with $x0$ in the hypothesis with the quantifiers, as is done in the proof of the theorem in the **Galois** project.

11. After translating the hypothesis into predicate logic by pressing all “ \in ”, you get a rule that states $\neg(\exists x \cdot x1 \mapsto x \in t \sim \wedge x \mapsto x0 \in (N \times N) \setminus t)$. Click on the “ \neg ” to get a hypothesis with a universal quantifier.
12. Now, you will want to get this hypothesis into a better applicable form. By using rewrites (clicking on the red symbols), you should be able to transform it into $\forall x \cdot x \mapsto x1 \in t \wedge x \in N \wedge x1 \in N \Rightarrow x \mapsto x0 \in t$. Try to get the hypothesis into the right form yourself. Should you completely mess something up, you can still go a few steps back in the proof tree by clicking on the “Backtrack from the current node” button in the proof control.
13. The instantiation for x should now be obvious. After the instantiation, you get a new hypothesis. If you Apply the Modus Ponens on this, the smiley in the Proof Control will turn blue. This means that all the non-reviewed goals have been proven. So, you will now have to prove the second subgoal. Click on “Select the next review subgoal” to get there.
14. The proof of the second subgoal begins similar to the others. Remove the inclusion in the goal and simplify the goal as far as possible. Next, translate the new selected hypotheses into predicate logic by clicking on the “ \in ” symbols in the selected hypothesis until there is none left. There should be one selected hypothesis that starts with a negation. Remove the negation in it. Now we get a hypothesis beginning with a universal quantifier. It should look like this, just the variable names might differ: $\forall x0 \cdot \neg (x \mapsto x0 \in t \sim \wedge x0 \mapsto x2 \in (N \times N) \setminus t)$.
15. The hypothesis stated above can also be brought into a more easily comprehensible and directly applicable form. In the end, it should state something like $\forall x0 \cdot (x0 \mapsto x \in t \wedge x0 \in N \wedge x2 \in N \Rightarrow x0 \mapsto x2 \in t)$, where names may vary. Get the hypothesis into the right form yourself using rewrites (similar to step 12). Now, the instantiation for $x0$ should be obvious, as there is only one variable that maps to x (In Figure 3, $x1$).
16. The rest of the subproof is the same to steps 7 and 8, except for the variable names. So you can either repeat these steps or try to copy the proof. To copy the proof, right-click on the part that you need on the proof tree where you added the hypothesis and choose copy. Then right-click on your current location and select paste. By all chance, the proof did not work due to the different names. Look in the proof tree for where the wrong name has been used and prune that part. You will have to redo that bit of the proof again.

5.2 Exercise: Theorem 2

In Theorem 2, you have to prove that $t = f \text{ union}(t; f)$. For this, you have to prove both directions ($t \subseteq f \cup (t; f)$ and $f \cup (t; f) \subseteq f$). Start with the second part, as it is much

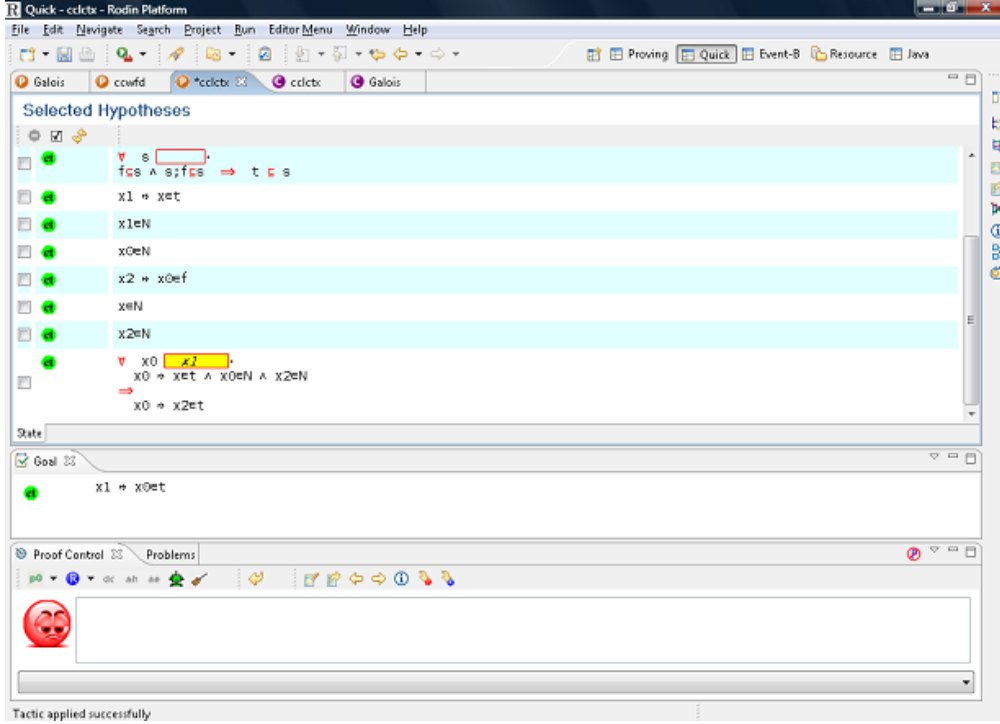


Figure 3: The proof at step 15

easier. The first part also should not be too hard, if you instantiate the hypothesis with the quantifier as early as possible.

5.3 Exercise: Theorem 3

Theorem 3 is very similar to Theorem 2. Proving also works in a very similar fashion, except that it takes a few steps more, as we do not have any axioms on $(f; t)$. Then, Theorem 1 usually is useful.

5.4 Exercise: Theorem 4

Theorem 4 states that $\forall b. f[b] \subseteq b \Rightarrow t[b] \subseteq b$. Once again, you will need the right instantiation of a hypothesis to succeed with the proof at some stage. Here, $((N \setminus b) \times N) \cup (b \times b)$ will be a good instantiation. This proof is quite lengthy, as there are many proofs by cases that you will need to perform. A lot of the cases can be solved by successfully adding the negation of a selected hypothesis and thus creating a contradiction.