Click'n Prove

Interactive Proofs Within Set Theory

by J.-R. Abrial and D. Cansell

Version 23

May 2003

Click'n Prove: Interactive Proofs Within Set Theory

Jean-Raymond Abrial[1] and Dominique Cansell[2]

[1] Consultant Marseille France
jr@abrial.org

[2] LORIA, INRIA Lorraine France
Dominique.Cansell@loria.fr
Partly supported by PRST IL/QSL/ADHOC project

Abstract. In this article, we first briefly present a proof assistant called the Predicate Prover, which essentially offers two functionalities: (1) an automatic semi-decision procedure for First Order Predicate Calculus, and (2) a systematic translation of statements written within Set Theory into equivalent ones in First Order Predicate Calculus. We then show that the automatic usage of this proof assistant is limited by several factors. We finally present (and this is the main part of this article) the principles that we have used in the construction of a *proactive* interface aiming at circumventing these limitations. Such principles are based on our practical experience in doing many interactive proofs (within Set Theory).

1 Introduction

We believe that the *modeling* of software systems and more generally that of complex systems is an important phase in their rigorous development. This is certainly very common in other engineering disciplines, where models are sometimes called *blueprints*. We also believe that the writing of such models can only be done by *stepwise refinements*. In other words, a model is built by successive enrichment of an original simple "sketch" and then by some careful transformations into more concrete representations. As an analogy, the first sketchy blueprint of an architect is gradually zoomed in order to eventually represent all the fine details of the intended building and then some decisions are made concerning the way it can be constructed, thus obtaining the final complete set of blueprints. We believe that the usage of some formal language is indispensable in such a modeling activity. It gives you the necessary framework within which such models can be built. This is, in a sense, equivalent to the formal conventions to be used in the drawings of blueprints.

We also strongly believe that using such an approach without *validating* the formal texts which constitute the initial model and its successive refinements, is an error because the texts in question can then just be "read" (which is already difficult and rather boring) without any possibilities for reasoning about them. In fact, as the validation of such models cannot be done by testing (since there is no "execution" for a model: the blueprint of a car cannot be driven!) the only (and far more serious) possibility is to validate them by *proving* them.

Of course there exist many "formal" methods on the market which are not using either refinement nor any kind of proving system for validation. These are said to be already sufficient to help the designers. It is our opinion however that it is not sufficient nowadays, in the same way as it is not sufficient to draw a bridge and just observe the drawing in question to be persuaded that it will be strong enough.

The construction of formal proofs has certainly not yet entered the collection of standard techniques used by (software) engineers, although some of them have already used such techniques with great success, showing that it is quite feasible. One of the reasons, which is argued against the generalization of such techniques, is that it is too difficult to master (as an aside, we believe that what is really difficult to master is the technique of modeling rather than that of proving). In case where the proving system at hand performs an automatic proof, this is not a problem of course, but this is clearly not always the case. What seems difficult to master is then the technique of *performing a proof interactively* with the computer. To a certain extent, we agree with that opinion. But we believe that, besides a possible intrinsic (but rare) mathematical difficulty, it is most of the time difficult essentially because the interfaces that are usually proposed to users are not adequate. This is the point where we arrive in this introduction at the purpose of this paper: its aim is to report on an experiment in building such an *interface* for a proving system which is possibly, but not exclusively, used in industry.

Being familiar with the B Method [2] and its industrial tool called **Atelier B** [6], our project consisted of completely reshaping the external aspect of this tool in order to give it a more adequate user interface (we are aware however that the adequacy in question is certainly very subjective). We have chosen to develop this interface under **Emacs** for various obvious reasons, which we are not going to develop here.

Among other things, **Atelier B** is essentially made of two tools: the Proof Obligation Generator and the Prover. The former is able to analyze a formal text and to extract from it a number of mathematical statements expressing what is to be proved in order to make sure that the text in question is "correct". Once this is done, these lemmas are passed to the Prover, which first scans them and tries to prove them automatically as much as it can. But there always remains a number of lemmas that cannot be proven automatically (some of them of course because it is not possible to prove them at all, hence detecting a bug in the corresponding model). This is where interactive proofs may enter into the scene.

The prover of **Atelier B** is made of two independent pieces: the first one is called **pb**[1] (for "prover of **Atelier B**"), whereas the second one is called **pp** (for "predicate prover"). The prover **pb** works directly at the level of Set Theory (the mathematical language used in the B Method), it is essentially made of a large number of inference rules able to discharge a large number of "simple" mathematical statements. These rules have been constructed in a rather experimental way. Historically, **pb** was the only prover of **Atelier B**. The prover **pp** had been developed as an independent project in order to validate the many rules of the other, and it was only later fully incorporated within **Atelier B**. The prover **pp** is more elaborate than **pb**, it is based on a firmer theoretical background: it is essentially a prover for First Order Predicate Calculus with Equality (and some Arithmetic) together with a Translator from Set Theory to First Order Predicate Calculus.

The organization of this paper is as follows. In section 2, we briefly describe the various parts of **pp**. In section 3 we explain why **pp** may fail to automatically prove some statements and hence requires the presence of an interactive interface. In section 4 (which is quite large) we describe this interface in a progressive way. In section 5 we open a general discussion about the principles we have used in the development of this interface and we also perform a short comparison with other proof assistants. In section 6 we conclude the paper.

[1] We shall only briefly mention **pb** in this paper, but we must say that most of the features described in section 4 have been borrowed from those existing already in **pb**. So that we have often just make them easy to use.

2 Set-theoretic Proofs with the Predicate Prover

2.1 Some Underlying Principles Behind the Construction of pp

In this section, we present some ideas and concepts that have driven us in the construction of the Predicate Prover.

To begin with, **pp** has been developed *incrementally* on the basis of a *hierarchy of provers*. Although important, this strategy is, after all, nothing else but a good design practice.

The most important idea, we think, behind the construction of **pp**, lies in the fact that it has been designed around a fixed *wired-in* logical system, which is the most classical of all, namely First-Order Predicate Calculus with Equality (used as the *internal engine*), and Set Theory (used as the *external vehicle*). The prover **pp** also contains some treatment of Arithmetic, which we shall not mention further here.

In no way is **pp** constructed from a meta-prover able to be parameterized by a variety of distinct logics. This contrasts with what can be seen in academic circles where extremely powerful general purpose *Proof Systems* are usually offered: HOL [8], Isabelle [12], PVS [11], Coq [5], etc. Our approach is quite different. It is rather similar to that used in the development of some industrial programs handling symbolic data. For instance, a good C compiler is not a meta-compiler specialized to C; likewise, a good chess-playing program is not a general purpose game-playing program specialized by the rules and strategies of the game of chess.

In our case, we have internalized classical logic because it is clearly that logic which is used to handle the usually simple lemmas which validate software and more generally system developments. This is *not* to say, however, that classical logic is the logic of software and system development. Our view is that the logic of software development is whatever logic one wants (Hoare-logic, wp-logic, temporal logic, etc). Such logics, we think, are not the ones concerned by the "how-to-prove", they are the ones used to generate the "what-to-prove".

We think that it is important to completely separate these two functions in two distinct tools: again, this is what is done in **Atelier B** where you have the, so-called, Proof Obligation Generator based on a weakest pre-condition analysis of formal texts and which delivers the statements to be proved (it tells you the "what-to-prove"), and, as a distinct tool, the Prover which is supposed to discharge the previous statements (it tells you thus the "how-to-prove"). Our view is that, whatever the logic of system development you decide to use, it will generate some final lemmas that are, inevitably, to be proved within classical logic.

Moreover, we do not think at all that the usage of Set Theory restricts you to the expression of elementary mathematical facts only: it is quite possible to encode within Set Theory mathematical statements that are clearly belonging to what is called higher order logic [1] (another example is given in **Appendix 2**). A mathematical analogy which can be used at this point, is the well known fact that some non-Euclidean geometry can be "encoded" (on a sphere) within the Euclidean one.

2.2 A Propositional Calculus Decision Procedure

The prover **pp** essentially first contains a decision procedure for Propositional Calculus. This procedure is very close to what is elsewhere proposed under the technical name of Semantic Tableaux [7].

More precisely, the proof procedure consists of applying a number of simple rules (not detailed here), which *gradually* transform an original sequent with no hypotheses into one or several sequents with hypotheses made of *literals* only (that is, either simple propositions containing no logical connectives or the negation of such simple propositions). Such sequents are discharged as soon as their collection of hypotheses contains a certain atomic proposition together with its negation (such literals are boxed in the example that follows). Next is an example proof of a propositional statement with this procedure:

$$\vdash\ (A \vee B) \Rightarrow (A \vee (B \vee C))$$

$$\vdash\ A \Rightarrow (A \vee (B \vee C)) \qquad\qquad \vdash\ B \Rightarrow (A \vee (B \vee C))$$

$$A\ \vdash\ A \vee (B \vee C) \qquad\qquad B\ \vdash\ A \vee (B \vee C)$$

$$A\ \vdash\ \neg A \Rightarrow (B \vee C) \qquad\qquad B\ \vdash\ \neg A \Rightarrow (B \vee C)$$

$$\boxed{A},\ \boxed{\neg A}\ \vdash\ B \vee C \qquad\qquad B,\ \neg A\ \vdash\ B \vee C$$

$$B,\ \neg A\ \vdash\ \neg B \Rightarrow C$$

$$\boxed{B},\ \neg A,\ \boxed{\neg B}\ \vdash\ C$$

2.3 A Semi-Decision Procedure for First-Order Predicate Calculus

The just presented simplified form of **pp** has then been extended to handle First-Order Predicate Calculus statements. Within this new framework, the prover again starts with a sequent with no hypotheses. It then loops through two distinct phases:

1. Phase 1: The prover first proceeds by applying some rules as above with the only difference that the collection of hypotheses associated with a sequent in formation not only contains some atomic propositions as before, but also some *universally quantified predicates* that are normalized in a certain systematic fashion. The prover thus gradually transforms the original sequent until the right-hand part of the remaining sequent is reduced to F (for FALSE) unless, of course, it has been discharged in the meantime by discovering two contradicting literals. If the contradiction is not reached trivially, one has then to prove that the simple hypotheses (the literals) are contradictory with the universally quantified ones: this is the purpose of the second phase.

2. Phase 2: This search for contradiction is attempted by means of some *instantiations* of the universally quantified hypotheses. Some "interesting" instantiations are discovered in a systematic fashion by means of the atomic hypotheses. This technique is a special case of the, so-called, "set of support" technique of Otter [10], where the set in question is represented by the atomic hypotheses. The conjunct of the retained instantiations is then moved into the right-hand part of the sequent, where it implies F, thus forming a new sequent. At this point, the second phase is completed, and the prover proceeds again with the first phase, and so on.

Here is a little example of using this "ping-pong" technique to prove a simple statement:

$$\vdash\ \forall x \cdot (\, (\exists y \cdot R_{xy}) \Rightarrow \forall z \cdot R_{zx}\,) \Rightarrow \forall\,(x,y) \cdot (\, R_{xy} \Rightarrow R_{yx}\,)$$

Phase 1

$$\forall\,(x,y,z) \cdot \neg\,(R_{xy} \wedge \neg\, R_{zx}\,),\ R_{xy},\ \neg\, R_{yx}\ \ \vdash\ \mathsf{F}$$

Phase 2

$$\forall\,(x,y,z) \cdot \neg\,(R_{xy} \wedge \neg\, R_{zx}\,),\ R_{xy},\ \neg\, R_{yx}\ \ \vdash\ \neg\,(R_{xy} \wedge \neg\, R_{yx}) \Rightarrow \mathsf{F}$$

Phase 1

$$\forall\,(x,y,z) \cdot \neg\,(R_{xy} \wedge \neg\, R_{zx}\,),\ \boxed{R_{xy}},\ \neg\, R_{yx},\ \boxed{\neg\, R_{xy}} \vdash \mathsf{F} \qquad \forall\,(x,y,z) \cdot \neg\,(R_{xy} \wedge \neg\, R_{zx}\,),\ R_{xy},\ \boxed{\neg\, R_{yx}},\ \boxed{R_{yx}} \vdash \mathsf{F}$$

In the example above the effect of Phase 2 is to instantiate x, y, and z in the universal hypothesis $\forall\,(x, y, z) \cdot \neg\,(R_{xy} \wedge \neg\, R_{zx}\,)$ with x, y, and y respectively. This leads to $\neg\,(R_{xy} \wedge \neg\, R_{yx})$, which is moved to the right hand side of $\vdash$ in order to resume Phase 1.

2.4 The Translation of Set-Theoretic Statements

The next part of **pp** is the Set Translator. It is built very much in accordance with the spirit of the set-theoretic construction presented in the B-Book [2], where Set Theory just appears as an extension of First-Order Logic. The goal of this extension essentially consists of formalizing the abstract concept of set membership.

Statements involving the membership operator are reduced as much as possible by the translator by means of a number of rewriting rules. It results in predicate calculus statements, where *complex* set memberships have disappeared, the remaining set membership operators being left uninterpreted. For instance, a set-theoretic predicate such as $s \in \mathbb{P}(t)$ is transformed into $\forall x \cdot (x \in s \Rightarrow x \in t\,)$. The translator then just performs the translation of the various instances of set membership. They correspond to the classical set operators (∪, ∩, etc), to the generalization of such operators, to the binary relation operators, to the functional operators (including functional abstraction and functional application), and so on. Next is such a set-theoretic statement expressing that intersection distributes over the inverse image of a partial function:

$$f \in s \mathrel{+\!\!\!\to} t \;\;\wedge\;\; a \subseteq t \;\;\wedge\;\; b \subseteq t \;\;\Rightarrow\;\; f^{-1}[a \cap b] \;=\; f^{-1}[a] \cap f^{-1}[b]$$

Here is the translation of this statement. It is easily discharged by **pp** using the procedure described in the previous section:

$$
\begin{array}{l}
\;\forall\,(x, y) \cdot (\,(x, y) \in f \;\Rightarrow\; x \in s \;\wedge\; y \in t\,) \;\;\wedge \\
\;\forall\,(x, y, z) \cdot (\,(x, y) \in f \;\wedge\; (x, z) \in f \;\Rightarrow\; y = z\,) \;\;\wedge \\
\;\forall x \cdot (\,x \in a \;\Rightarrow\; x \in t\,) \;\;\wedge \\
\;\forall x \cdot (\,x \in b \;\Rightarrow\; x \in t\,) \\
\Rightarrow \\
\;\forall x \cdot (\,\exists y \cdot (\,x \in a \;\wedge\; x \in b \;\wedge\; (x, y) \in f) \;\Rightarrow\; \exists y \cdot (\,y \in a \wedge (x, y) \in f\,) \;\wedge\; \exists y \cdot (\,y \in b \wedge (x, y) \in f\,)\,) \;\;\wedge \\
\;\forall x \cdot (\,\exists y \cdot (\,y \in a \wedge (x, y) \in f\,) \;\wedge\; \exists y \cdot (\,y \in b \wedge (x, y) \in f\,) \;\Rightarrow\; \exists y \cdot (\,x \in a \;\wedge\; x \in b \;\wedge\; (x, y) \in f)\,)
\end{array}
$$

As can be seen, membership predicates such as $(x, y) \in f$, $x \in s$, $y \in t$, etc, no longer play a rôle in this statement. They could have been replaced by $F_{x,y}$, S_x, T_x, etc.

3 Deficiencies of the Automatic Mode of the Predicate Prover

Although **pp** is able to automatically prove some non trivial statements (see **Appendix 1**), it is nevertheless limited in various ways. The purpose of this short section is to analyze such deficiencies. We can detect four kinds of deficiencies.

The first limitation of **pp** is certainly its *sensitivity to useless hypotheses*. In other words, **pp** may easily prove a certain mathematical statement, and fail (or take a very long time) to prove the same statement once prefixed with a large collection of useless hypotheses. The prover **pp** is certainly not the only prover to suffer from such a “disease”. Note that this problem of useless hypotheses seems rather strange since, clearly, a mathematician never tries to prove a statement containing such hypotheses, his understanding of the problem usually telling him the relevant assumptions. But in formal developments, as we have mentioned above, the statements to prove are not determined by a

human being (this is prone to many errors), they are rather *automatically generated* by some tools. And in this case it is quite frequent to have a large number of irrelevant hypotheses.

A second limitation is due to the fact that **pp** is not good (so far) at *abstracting* the statements to prove. A mathematician can almost immediately detect that a certain statement to prove is in fact a special case of a simpler one, which could have been obtained by replacing some common sub-expressions by various dummy variables. And, again, the mathematician can do so because he has a good understanding of the problem at hand. As above, **pp** can take quite a long time to prove a statement that it could have proved easily once abstracted.

A third factor to take into account is the fact that **pp** is not always very good at *managing equalities*. In other words, it sometimes fails to take advantage of the fact that two (set-)expressions are equal or two predicates are equivalent.

The fourth factor is not so much a limitation of **pp** itself but rather one due to the fact that it is *automatic* . It is the recognition that certain statements to prove require some non-trivial *creative strategies*. Examples of such creative strategies are the following:

- Attempt to prove a lemma which, once proved, will become a new hypothesis,
- Perform a proof by cases,
- Decide for a proof by contradiction,
- Attempt to prove the negation of a certain hypothesis (another form of contradiction),
- Propose a witness for an existential statement to prove,
- Similarly propose a specific instantiation for a universally quantified hypothesis,
- Decide for a proof by induction,
- etc .

As a result, it seems advisable to construct a *layer*, with which a person can intervene in a proof by *preparing* the work to be given to **pp**. This is the purpose of next section.

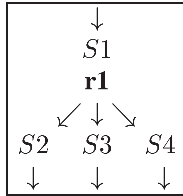
4 The “Click’n Prove” Proactive Interface

4.1 Enlarging the usage of pp within an interactive Proof Assistant

Our proof assistant deals with the proofs of *sequents* of the following form:

$$hypotheses \quad \vdash \quad conclusion$$

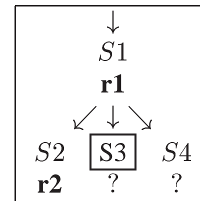
where $hypotheses$ denotes a possibly empty list of predicates and where $conclusion$ denotes a predicate which is not conjunctive and usually implicative. Conducting a proof within this framework leads to the application of some pre-defined *inference rules* able to transform such a sequent into zero, one or more *successor sequents*. Once such a rule is applied to it, a sequent is said to be *discharged* and the successor sequents (if any) have then to be discharged themselves. At this point, the procedure resumes by independently applying some inference rules on each of the remaining sequents, and so on. When a successor-less inference rule is applied on the last remaining non-discharged sequent then we are done. Note that at some point in this procedure, we may have a sequent which **pp** or **pb** could successfully discharge. In this case, **pp** or **pb** could then be considered as “special” and powerful inference rules with no successor sequent.



The *trace* of such a procedure, which records the various *states* of a proof (that is, the sequents to be considered at each step), constitutes what is commonly known as a *proof tree*. A node in such a tree is a sequent and the relationship between a node and its set of "offspring nodes" denotes the inference rule which has been applied to this node. In the following diagram, you can see part of such a proof tree with a node $S1$ together with his three "offspring nodes" $S2$, $S3$, and $S4$. Rule **r1** has been applied to $S1$ and produced $S2$, $S3$, and $S4$.

An *automatic* proof is one where the proof tree is constructed entirely automatically from an initial sequent and a set of pre-defined inference rules. In this case the prover is able to choose a correct inference rule at each step of the proof.

An *interactive* proof is one where the prover just correctly applies the rule, which a user tells it to apply at each step. During the proof process, the proof tree may contain some leafs that are "final" because a successor-less inference rule has been applied to them, and several other leafs which are "pending" because some inference rules have yet to be discovered for each of them. In this figure, we have a situation where leaf $S2$ is "final" because the successor-less rule **r2** has been applied to it, whereas leafs $S3$ and $S4$ are still "pending".



In this situation, the user has the choice to concentrate either on node $S3$ or on node $S4$ in order to discover which inference rule to apply in each case. In the figure, we have "boxed" node $S3$ in order to indicate that the user is now interested in discovering an inference rule for that node.

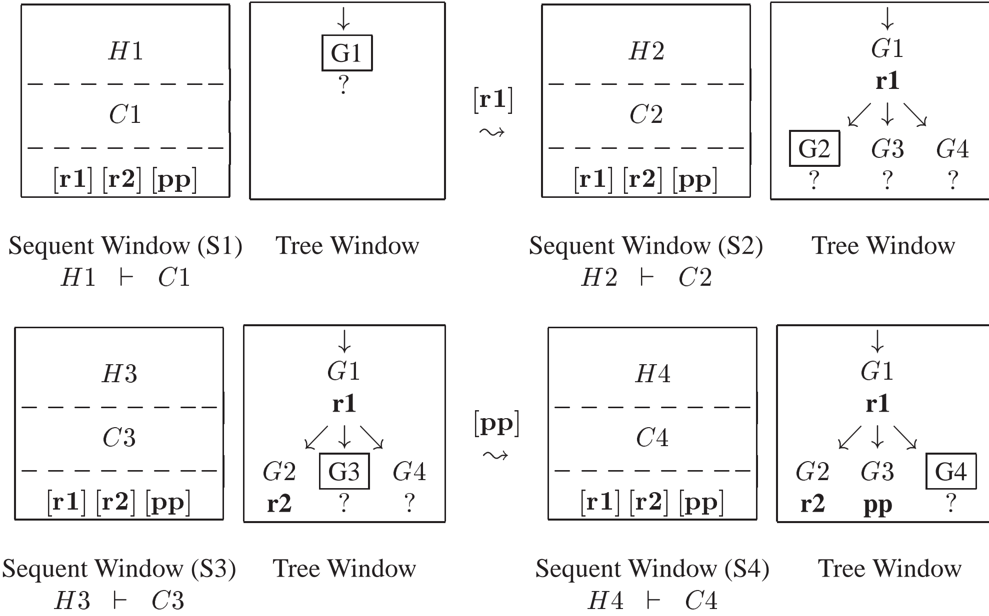
The proof tree thus appears to be the "object" with which the user has to play during the proof. Unfortunately, such an object is not at all tractable because it is *far too big*. The sequents forming the nodes may contain many large hypotheses and the conclusion may be quite large too. Finally the tree itself can be quite deep. In other words, the full tree does not fit any physical screen (and even the user brain).

In order to help solve this problem we have no choice but to *simplify the proof tree*. This can be done in various ways. First, we may not record the full sequent in each node, only some "significant part" of it. In the figures below we show only simplified sequents[2], namely $G1$, $G2$, $G3$, and $G4$ corresponding respectively to the full sequents $S1$, $S2$, $S3$, and $S4$. This simplified tree is stored in one window: let us call it the *Tree Window*. We also introduce a second window containing the "boxed" full sequent we are interested in at a given moment: this is the *Sequent Window*. In this window, beside the sequent, we may have various buttons (here buttons **[r1] [r2] [pp]**) that are able, when pressed, to activate the corresponding inference rule on the sequent which is currently there. These two windows can be pictured as indicated. We have also shown the dynamics of the proof. You can see how both windows evolves when the user successively presses **[r1]**, **[r2]** (not shown in the figures) and **[pp]**.

What we have presented so far is clearly very sketchy. In fact, these two windows are certainly *not* organized as indicated: what we have shown are just *abstractions* of the real windows at work in the real interface. The rôle of this quick overview was just to set up the scene. It remains now for us to develop the way each of the two windows is organized and how both of them are able to work in cooperation. In sections 4.2 to 4.10 we first study the Sequent Window, and in sections 4.11 to

[2] The exact nature of the simplified sequents with respect to the full ones will be described in section 4.11.

4.13 we study the Tree Window. Notice that in section 5, we shall present some general principles on which we have based our construction of the interface presented in section 4.



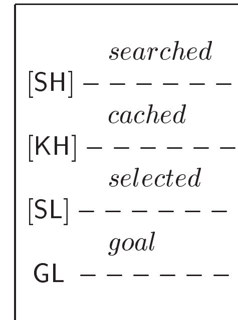
4.2 Managing the Hypotheses of the Sequent Window: [sh] [sg] [sl] [ds] [dl]

The most natural organization for the Sequent Window would be to split it up in two parts as shown in the previous section: one part for the *hypotheses* and another one for the *conclusion*. But the fact that there are *many hypotheses* forces us to structure the sequent in a richer fashion as follows, where *goal* is a non-conjunctive and non-implicative predicate and where the other "fields" are possibly empty conjunctive lists of predicates:

$$\underbrace{hidden \;\; ; \;\; searched \;\; ; \;\; cached}_{hypotheses} \quad \vdash \quad \underbrace{selected \;\; \Rightarrow \;\; goal}_{conclusion}$$

Next are the informal "window" properties of these various kinds of hypotheses:

hidden	Not visible on the window
searched	Visible after some search in *hidden*
cached	Visible but not part of the conclusion any more
selected	Visible and part of the conclusion



The layout of the window, with which the user can interact with the prover, reflects the structure of the sequent. It is thus divided into various areas where each hypothesis is written on a separate line. This layout evolves as the proof proceeds since each area can contain a variable number of hypotheses including no hypotheses at all, in which case the area simply does not exist.

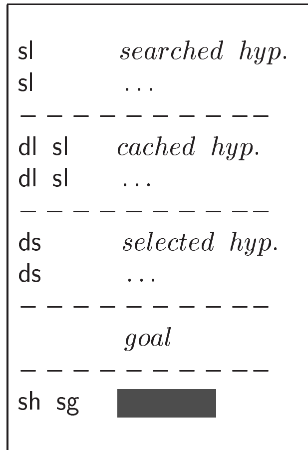
This is the reason why each area is indicated on the window with a specific label as shown in the figure. The first three labels (those surrounded by square brackets) are "buttons" which, when pressed, make the hypotheses of the area being temporarily hidden until one presses the button again. Other

buttons (not shown here) allows one to show more (or less) hypotheses in each area.

The idea behind this splitting of the hypotheses into various areas has been borrowed from the notion of *hierarchy of memories* which is usual in computer systems. We may roughly consider the following analogy between our areas and various memory media.

$hidden$	$disk$
$searched$	$core$
$cached$	$cache$
$selected$	$register$

A number of commands, namely [sh], [sg], [sl], [ds], and [dl], allows one to move an hypothesis from one area to another. A button dedicated to the relevant command is thus placed *on each line* in front of the corresponding hypothesis as shown in the figure. The window is thus divided horizontally in a "button area" on the left, and an hypotheses (or goal) area on the right.



Here are the various functions of these commands:

sh, sg	searching
sl	selecting

ds	deselecting
dl	deleting

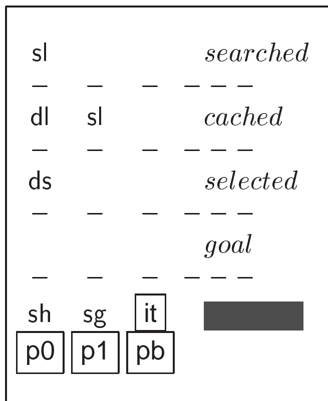
Next is the way each command may move an hypothesis from one area to another.

$hidden$	sh	$searched$
$searched$	sg	$searched$
$searched$	sl	$selected$
$selected$	ds	$cached$
$cached$	sl	$selected$
$cached$	dl	$hidden$

The button [sh] requires a search pattern, which is (in first approximation) a character string on which the search is performed. It must be entered before pressing the button. This pattern is either defined by pointing with the mouse to a piece of text anywhere on the window, or by writing it in the editing area (shown in black on the figure). After pressing the button, $searched$ (if existing) is first cleared, and then all hypotheses containing an occurrence of the given pattern are moved from $hidden$ to $searched$. The button [sg] allows one to search further within the $searched$ area itself.

4.3 Launching pp and pb: [p0] [p1] [pb] [it] *(Appendix 3 contains a list of all commands)*

A proof is started in the following situation where the $searched$ and $cached$ areas are empty:



$$hidden \;\vdash\; \underbrace{selected \;\Rightarrow\; goal}_{conclusion}$$

Note that **pp** never uses the $hidden$ hypotheses nor the $searched$ and $cached$ ones, it only works with the $conclusion$ (but it might be launched even if the $searched$ and $cached$ areas are present). If, in this initial situation, we feel that **pp** has enough hypotheses (in $selected$), we may launch it by pressing the button [p0] shown at the bottom of the window. The prover **pp** may succeed, in which case we are done, but it may also fail because it needs more hypotheses. In this case, we search for some hypotheses in $hidden$ and select those which seems to be most relevant.

This is done by first using the button [sh] together with a certain pattern as explained above: the corresponding hypotheses then appear in the $searched$ area. We then use some buttons [sl] of $searched$ to move certain hypotheses to $selected$ and we press [p0] again.

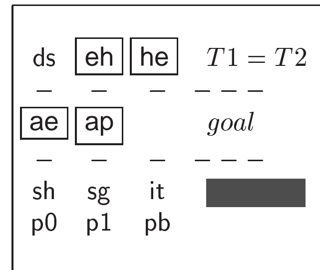
On the contrary, **pp** may fail because it has too many hypotheses. In this case, we may "Deselect" some of them by pressing the buttons [ds] situated in front of them in the $selected$ area. Note that in case we remove an hypothesis by mistake, it is easy to select it again as it is still in the $cached$ area.

Another common situation is one where the number of needed hypotheses is quite large. In this case, we may follow another technique: rather than searching and selecting some hypotheses, we directly launch **pp** with the button [p1]. This has the effect of automatically enlarging the $selected$ area (but for this session of **pp** only) with those hypotheses having some identifier in common with the identifiers occurring in the $selected$ and $goal$ areas. This is clearly a very straightforward technique, but it may quite often have very spectacular effects. The drawback of this technique is of course that it may select useless hypotheses, and **pp** may then be slowed down (a powerful machine is a good help here!).

In all these cases, instead of launching **pp**, we may do so with **pb** by pressing the corresponding button. Notice that **pp**, as well as **pb**, are not decision procedures. As a consequence, they may sometimes run for ever. In order to circumvent this "difficulty", they are automatically stopped after a certain pre-defined time. But they can also be stopped manually by pressing the button [it] (for "interruption") situated in the bottom part of the window. During a run of **pp** or **pb**, all other buttons are locked except [it], which is highlighted in order to possibly capture the attention of the user: he is then aware that either **pp** or **pb** are working.

4.4 Commands for Reshaping the Conclusion: [ae] [ap] [eh] [he]

As explained in section 3, **pp** may be helped by *abstracting* a common sub-term T in the $conclusion$. For this, we have a button called [ae] (for "abstract expression"). The sub-term T is defined by pointing with the mouse on some piece of text in the window. After pressing [ae], the prover automatically chooses a *free identifier*, say e, and replaces all occurrences of T by e in the $selected$ and $goal$ areas. It also puts the predicate $e = T$ in the $cached$ area. A similar button, [ap], is used to abstract a predicate. These permanent buttons are put next to the $goal$ as shown in this figure.

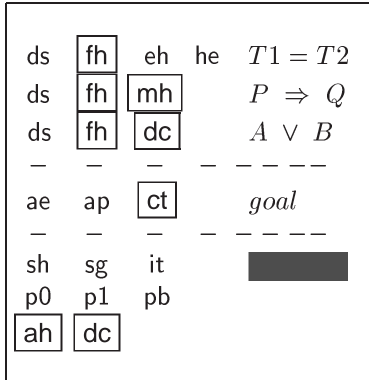


In order to lighten the figure we shall, from now on, only show the $selected$ area in the schematic Sequent Window. But the reader should be aware that the other areas may be non-empty.

Another case where **pp** can be helped is by using some hypothesis of the form $T1 = T2$ and either replacing occurrences of $T1$ by $T2$ or vice-versa. This can be done with two commands called [eh] (for "apply equality in hypothesis from left to right") and [he] (for "apply equality in hypothesis from right to left") both placed next to an hypothesis of the required form. The hypothesis in question can be in $searched$, $cached$ or $selected$, but the replacement only occurs in $conclusion$, that is in $selected$ and $goal$. In the above figure we have shown such an hypothesis in the $selected$ area.

4.5 General Purpose Proof Commands: [**ah**] [**dc**] [**ct**] [**mh**] [**fh**]

In this section, we present a number of general commands, which help organizing a proof. The first one, called [ah] (for "add hypothesis"), is used to prove a *local lemma*, which will become, once proved, a new $selected$ hypothesis for proving the original goal (this is also called the "cut rule"). This permanent button [ah] is situated in the bottom part of the window. The predicate, say L, to prove is defined by pointing to it with the mouse in the window, or alternatively by editing it in the black field. Notice L will replace the current $goal$ so that the collection of hypotheses remains "the same" during its proof.



In case L has an implicative form, say $U \Rightarrow V$, then U is automatically moved to the $selected$ area whereas $goal$ becomes V.

Another general command is called [dc] (for "do case"). It requires a parameter which is a certain predicate C. As for the previous command, this parameter can be defined by pointing to it with the mouse, or alternatively by editing it in the black field. Applying this command results in the successive proofs of $goal$ with either C or $\neg\, C$ as extra $selected$ hypotheses. As for the previous button, this permanent button is situated in the bottom part of the window.

Our next command, called [ct] (for "proof by contradiction"), allows one to assume the negation of $goal$ as an extra $selected$ hypothesis, $goal$ itself being replaced by FALSE. This permanent button is situated near $goal$ as shown.

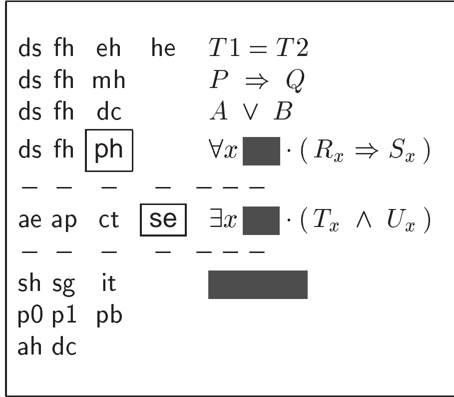
We now have two more general commands related to hypotheses with certain forms. When an hypothesis has the form $P \Rightarrow Q$, the command [mh] (for "modus ponens on hypothesis") is proposed. After pressing that button, the $goal$ is replaced by P. When P is proved then the new hypothesis Q is $selected$ and the original $goal$ is put back in place.

When an hypothesis has the form $A \vee B$, another kind of [dc] command (for "do case") is proposed. Pressing that button results in $goal$ being successively proved under the extra $selected$ hypothesis A and then B. As you can see in the figure, these buttons are dynamically constructed next to corresponding hypotheses.

A last general command is associated with all hypotheses. It is called [fh] (for "false hypothesis"). When pressing that button on an hypothesis H, the goal is simply replaced by $\neg H$. In case of success, this reveals a contradiction in the collection of hypotheses.

4.6 Commands for Instantiating Quantified Hypotheses or Goals: [**ph**] [**se**]

Another case where the user can help **pp** is one where he may propose some "interesting" instantiations to be performed on a universally quantified hypothesis. A button [ph] (for "particularize universal hypothesis") is situated next to an hypothesis with the corresponding form. Note that a small editing field is provided close to the quantified variable definition. Suppose we have an hypothesis of the form $\forall x \cdot (R_x \Rightarrow S_x)$. When pressing the button after filling the field with an expression E, then the lemma R_E is proposed (it replaces the current goal with the same hypotheses). Once R_E is proved, the original goal is put back in place together with the new hypothesis S_E.

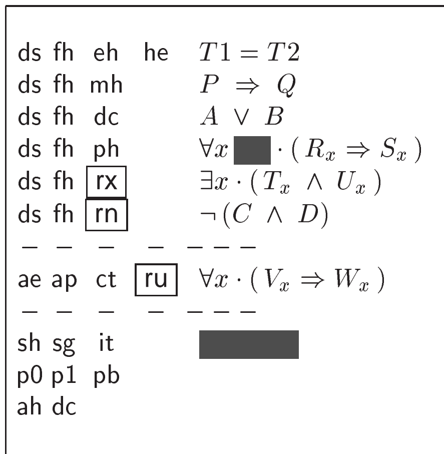


When the universal hypothesis contains several quantified variables, a little editing field is provided next to each variable. By filling some of these fields with a “$\star$”, this command results in a *partial instantiation* of the hypothesis. In other words, the resulting new hypothesis is still universally quantified under the “starred variables”, whereas the other variables are instantiated within the predicate.

A similar button, called [se] (for “suggest a witness for an existential goal”), is provided in the $goal$ field when it has an existential form as shown in the figure. Suppose we have a goal of the form $\exists x \cdot (T_x \wedge U_x)$. If the proposed witness is the expression E then the two goals T_E and U_E are presented one after the other.

Notice that these two commands are indeed the same. For instance, we could “simulate” the command [se] by means of the command [ph] as follows. It is always possible to apply [ct] to the goal $\exists x \cdot (T_x \wedge U_x)$, thus getting the new goal FALSE together with the hypothesis $\neg \exists x \cdot (T_x \wedge U_x)$, which can be transformed into $\forall x \cdot (T_x \Rightarrow \neg U_x)$ after pressing [rn] (for “remove negation”, see next section). We can then press [ph] with the expression E on this hypothesis. This requires to prove T_E. After this proof, we obtain the new hypothesis $\neg U_E$, still with the Goal FALSE to be proved. This hypothesis $\neg U_E$ can then be transformed into the goal U_E by pressing [fh] (for “false hypothesis”) !

4.7 Logical Simplification Commands: [ru] [rx] [rn] [rd]



Although it is not necessary for **pp**, it might sometimes be important for the progress of an interactive proof to remove the quantification of a universally quantified goal or that of an existential hypothesis. Two commands called [ru] (for “remove universal”) and [rx] (for “remove existential”) are provided for this, as shown in the figure below. Of course, in case the quantified variable is not free in the sequent, an “alpha conversion” takes place when applying such commands. For instance, suppose we have a goal of the form $\forall x \cdot (V_x \Rightarrow W_x)$. By pressing [ru], the predicate V_x (when x is supposed to be free in the sequent) becomes a new $selected$ hypothesis, and the goal becomes W_x. Suppose we have an hypothesis of the form $\exists x \cdot (T_x \wedge U_x)$. By pressing [rx], the predicates T_x and U_x become new $selected$ hypotheses (when x is free).

An even simpler logical simplification command is provided, it is called [rn] (for “remove negation”). Again, this is not necessary for **pp**, but it can be useful in the progress of the interactive proof. Such negation removals may occur anywhere in the sequent (hypotheses or goal). In the figure, we have shown an hypothesis of the form $\neg (C \wedge D)$. By pressing this button, this hypothesis will be

changed to $\neg C \vee \neg D$. Notice that in this case the button [dc] (for "do case") will appear (see section 4.5) next to this new hypothesis. Similar transformations will apply on all other kinds of negated predicates: $\neg \forall x \cdot (P_x \Rightarrow Q_x)$, $\neg \exists \cdot (P_x \wedge Q_x)$, $\neg (A \vee B)$, $\neg (A \Rightarrow B)$, $\neg \neg P$.

A final logical simplification command is provided for a disjunctive $goal$ of the form, say, $I \vee J$. It is called [rd] (for "remove disjunction"). When pressing that button situated in the $goal$ area, the new hypothesis $\neg I$ is created in the $selected$ field, and the $goal$ becomes J.

4.8 Set-theoretic Simplification Commands: [rm] [ri] [rn]

We now propose a number of set-theoretic simplification commands. Again, they are not indispensable for **pp** but sometimes convenient in an interactive proof.

ds	fh	eh	he	$T1 = T2$
ds	fh	mh		$P \Rightarrow Q$
ds	fh	dc		$A \vee B$
ds	fh	ph		$\forall x$ ▇ $\cdot (R_x \Rightarrow S_x)$
ds	fh	rx		$\exists x \cdot (T_x \wedge U_x)$
ds	fh	rn		$\neg (C \wedge D)$
ds	fh	[rm]		$E \in \{x \mid F_x \wedge G_x\}$
ds	fh	[ri]		$S \subseteq T$
ds	fh	[rn]		$X \neq \emptyset$
–	–	–	–	– – – –
ae	ap	ct	ru	$\forall x \cdot (V_x \Rightarrow W_x)$
–	–	–	–	– – – –
sh	sg	it		▇▇▇
p0	p1	pb		
ah	dc			

The first one, called [rm] (for "remove membership"), works on hypotheses or goals of the form $E \in \{x \mid F_x \wedge G_x\}$ as shown in the figure. Pressing that button results in the predicates F_E and G_E. They become either successive new goals (when [rm] is applied to the goal) or distinct new $selected$ hypotheses (when [rm] is applied to an hypothesis).

Another button, called [ri] (for "remove inclusion"), works on hypotheses or goals of the form $S \subseteq T$ as shown in the figure. Pressing that button results in the the predicate $\forall x \cdot (x \in S \Rightarrow x \in T)$ replacing $S \subseteq T$. Such a predicate could be further handled by buttons [ph] (section 4.6) or [ru] (section 4.7).

The button [rn] (for "remove negation") can also be used on predicates of the form $X \neq \emptyset$ as shown in the figure. When pressing that button, this predicate is replaced by $\exists x \cdot x \in X$, which can itself be further treated by buttons [se] (section 4.6) or [rx] (section 4.7).

4.9 Set-theoretic Miscellaneous Commands: [eq] [ov]

The next two commands have proved to be quite useful. The first button, called [eq] (for "prove a set membership by an equality") appears next to an hypothesis of the form $E \in S$ when the goal itself is of the form $F \in S$. By pressing that button, the new goal $E = F$ replaces $E \in S$. This button is then just a short-circuit: it saves attempting to prove the lemma $E = F$ with [ah] (section 4.5) and then applying [eh] on the resulting hypothesis (section 4.4).

The second button called [ov] (for "overriding") generates a proof by cases that is often encountered in set-theoretic statements. This button might be defined on an hypothesis or on the goal. It is present when a predicate P contains a sub-expression of the following form:

$$(f \Leftarrow g)(E)$$

where f and g are two partial functions and E is an expression.

The overriding operator $\mathbin{<\!\!\!+}$ is an infix operator taking (in general) two relations as arguments (here it takes two functions). The result of $f \mathbin{<\!\!\!+} g$ in our case is exactly the function g extended with the function f outside the domain of g. So that its application to E is either $g(E)$ when E is in the domain of g or $f(E)$ when E is in the domain of f but not in that of g. The presence of such a sub-expression somewhere in an hypothesis or in the goal strongly suggests a *proof by cases*, where the two cases are $E \in \mathsf{dom}(g)$ (in which case the sub-expression is replaced by $g(E)$) and $E \in \mathsf{dom}(f) - \mathsf{dom}(g)$ (in which case the sub-expression is replaced by $f(E)$). This is exactly what happens when this button is pressed.

ds	fh	eh	he	$T1 = T2$
ds	fh	mh		$P \Rightarrow Q$
ds	fh	dc		$A \vee B$
ds	fh	ph		$\forall x$ ■ $\cdot (R_x \Rightarrow S_x)$
ds	fh	rx		$\exists x \cdot (T_x \wedge U_x)$
ds	fh	rn		$\neg (C \wedge D)$
ds	fh	rm		$E \in \{x \mid F_x \wedge G_x\}$
ds	fh	ri		$S \subseteq T$
ds	fh	rn		$X \neq \emptyset$
ds	fh	[eq]		$E \in S$
ds	fh	[ov]		$P((f \mathbin{<\!\!\!+} g)(E))$
–	–	–		– – – –
ae	ap	ct		$F \in S$
–	–	–		– – – –
sh	sg	it		■
p0	p1	pb		
ah	dc			

4.10 Beginner and Expert Modes, On-line Help: [bg] [xp] hp]

The interface may function according to two distinct modes: either the *expert* mode (this is the default) or the *beginner* mode. While in expert mode, a [bg] button (for "beginner") allows one to switch to the other mode. Likewise in beginner mode, a similar button, called [xp] (for "expert"), is provided.

ds	fh	eh	he	$T1 = T2$
ds	fh	mh		$P \Rightarrow Q$
ds	fh	dc		$A \vee B$
ds	fh	ph		$\forall x$ ■ $\cdot (R_x \Rightarrow S_x)$
ds	fh	rx		$\exists x \cdot (T_x \wedge U_x)$
ds	fh	rn		$\neg (C \wedge D)$
ds	fh	rm		$E \in \{x \mid F_x \wedge G_x\}$
ds	fh	ri		$S \subseteq T$
ds	fh	rn		$X \neq \emptyset$
ds	fh	eq		$E \in S$
ds	fh	ov		$P((f \mathbin{<\!\!\!+} g)(E))$
–	–	–	–	– – –
ae	ap	ct		$F \in S$
–	–	–	–	– – –
sh	sg	it	[hp]	■
p0	p1	pb	[bg]	
ah	dc			

The difference between the two modes is quite significant. In expert mode, many buttons are "pressed" automatically by the prover itself: these buttons are essentially simplification buttons such as [rm], [ru], [rn] and so on. This is quite efficient but sometimes misleading for the user. He might not understand what is going on, in particular why the *goal* and the *selected* hypotheses have been modified in an unexpected way after pressing a certain button. When this is the case the user can backtrack (see section 4.12), press [bg] and resume the previous command. The prover will then not perform any automatic action "behind the curtain". Once he has understood what was done in expert mode by looking at the behavior in beginner mode, the user may switch back to the expert mode by pressing [xp].

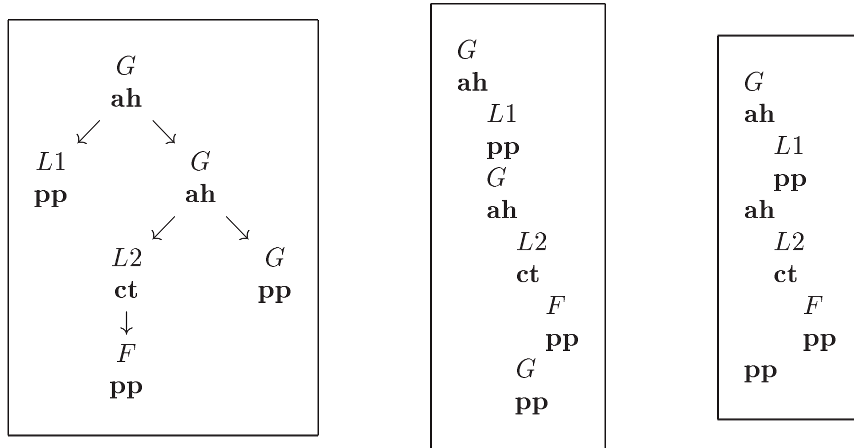
Those buttons, which are normally "pressed" automatically by the prover while in expert mode, are just *highlighted* while in beginner mode, thus suggesting to the user that such buttons might be pressed.

There exists a unique [hp] button (for "help"). After pressing that button, all other buttons when pressed do not do anything except producing a short explanation text for the corresponding command. In the mean time the [hp] button is highlighted to make the user aware that he has to press again that button in order to return to the normal mode of operation.

4.11 Representation of the Proof Tree: [zm] [mk]

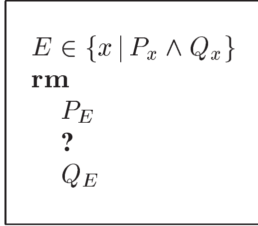
The proof tree is represented in the Tree Window according to a number of principles which are as follows:

1. Each node is represented by a simplified sequent. In fact, we shall only present the *goal* part of a sequent. The reasons for this are simple:
   (a) Usually the goal is not very large. Remember that goals are not conjunctive nor implicative, so that most of the time a goal is just a predicate defined by an equality, a set membership or a set inclusion. But we might have quantified predicates which are larger.
   (b) The number of hypotheses (even for those in *selected*) could be quite large.
   (c) The hypotheses are usually far less modified than the goal when one goes from one sequent to the next.
   (d) The absence of the hypotheses in the Tree Window is compensated by their presence in the current sequent of the Sequent Window.
2. Each simplified sequent is given together with the full command that has been applied to it: that is, the command name and its parameters.
3. In first approximation, the tree is represented vertically, the offspring nodes of a node being slightly shifted to the right (like a Table of Contents)
4. However, this shift is only performed when the simplified " offspring sequent" is different from its simplified "parent sequent". The reason for this is that we want to see at the same level all the commands that have been performed in order to prove a certain goal.
5. Moreover when a simplified "offspring sequent" is the same as its simplified "parent sequent" the former is not written again. The reason for this is to avoid copying again and again the same goal in the proof tree.



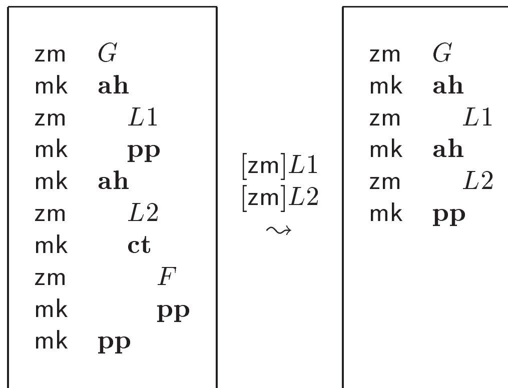
As an illustration of the previous principles, suppose that at some point in a proof, we have a certain goal G. In order to prove that goal we need two lemmas L1 and L2. The first one is proved directly by **pp**, whereas the second one is proved by contradiction and then **pp**. Once these two lemmas are proved, the proof of our original goal G follows immediately by **pp**. This results in

the exact proof tree shown on the figure above on the left. Next to it, is its vertical representation, drawn without taking into account points 4 and 5 above. In the last diagram, you may find a vertical representation which takes these two points into account. In the proof tree on the left we can "see" the progress of the proof on G, namely lemma $L1$, lemma $L2$, and **pp**. But this vision is clearly lost in the corresponding vertical representation in the center. Whereas, this structure is clearly visible again on the right-hand side diagram. This is the reason why we have adopted this representation.



When the application of a certain command results in several offspring sequents, we can see all of them in the Tree Window. We can also see which one is the current node: this is indicated by a "**?**". This is illustrated in the following example where our goal has the form $E \in \{x \mid P(x) \wedge Q(x)\}$. When applying [rm] (for "remove membership") the two goals P_E and Q_E are generated and the current goal is now P_E.

Two kinds of button are generated with each "line" in the Tree Window. At the beginning of a goal line, we have a button called [zm] (for "zoom/unzoom"). By pressing that button, the part of the proof tree having that goal at its top is hidden. Pressing [zm] again makes the sub-tree reappearing. While a part is hidden the [zm] button is highlighted. Here is an illustration of this mechanism. We have pressed [zm] twice: once for $L1$ and once for $L2$. This results in the tree shown in the right-hand side.



At the beginning of a command line, we have a button called [mk] (for "make an application of the command"). When pressing one of these button and then a second button of the same kind (the second button could be the same as the first one), this results in applying the series of commands situated between these two buttons. This is very convenient to repeat part of a proof.

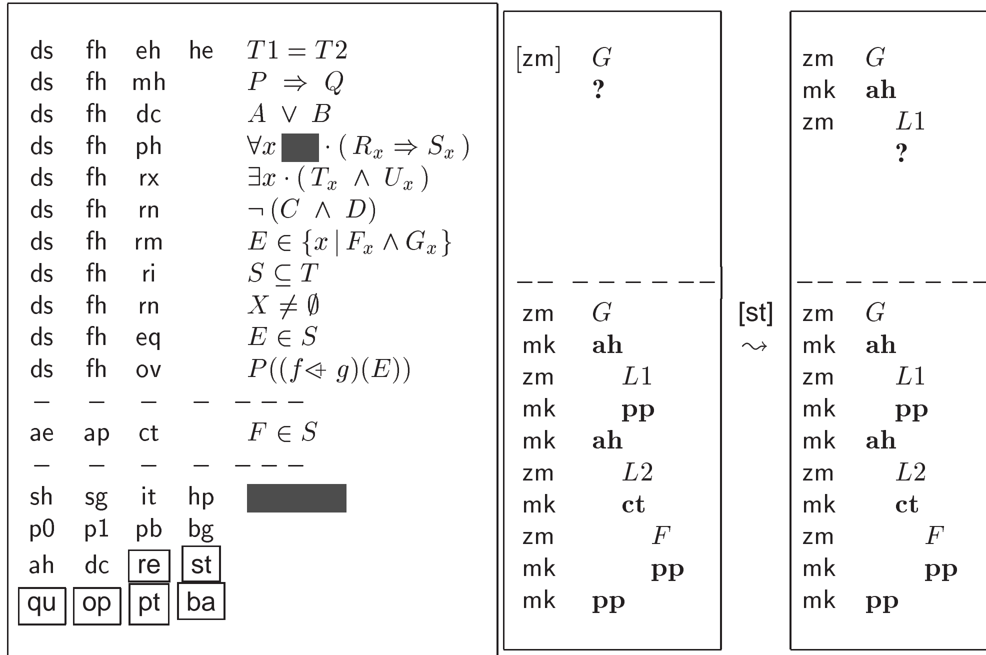
4.12 Proof Navigation: [ba] [re] [op] [pt] [st] [qu]

We have several possibilities to navigate on a proof. The basic button is called [ba] (for "backtrack"). It allows one to backtrack one step when we figure out that we have taken a wrong decision. A second navigation button, called [re] (for "reset") is provided to reset a proof. Once it is pressed, the proof done so far is not lost however. In fact, before resetting, the proof tree has been saved and it is now called the "old proof tree". So that, in general, we have always two proof trees, the "current" one and the "old" one.

When pressing the button [op] (for "old proof tree"), the Tree Window splits and we can see at the same time the current proof tree (on top) and the old proof tree (at the bottom). For instance, after doing the proof shown on the Tree Window at the end of the previous section, and then pressing [re] followed by [op], the Tree Window looks the way it is presented in the previous figure. By pressing the button [pt] (for "proof tree"), we may reinstall the current proof tree alone on the Tree Window.

Once the [re] button has been pressed, it is possible to gradually resume the proof by pressing the button called [st] (for "one step") as shown in the figure: we do one step of the old proof tree

within the new one. Another possibility in order to replay part of the old proof tree in the new one would be to use the [mk] buttons as explained in the previous section.



Finally, the button [qu] (for "quit") allows one to quit the proof. We can either quit and save the current proof tree or keep the previous proof tree. When we resume the proof in another session, then the previous proof tree automatically becomes the "old" proof tree.

4.13 Proof Refinement: [as] [r1] [r2]

The idea of a proof refinement is very important. In fact this is the way professional mathematicians usually work. The idea is very simple. If at some point in a proof we find a difficulty to prove a certain predicate P (it might be a lemma or any other goal), we would like to jump over it and consider for the time being that we have a proof for it. We then proceed with the main proof and then eventually figure out that this lemma P is what we needed. If this is the case, then we shall come back some time later to perform that part of the proof that was missing (for P) in the middle of the main proof. In doing so, we might find similar difficulties and use the same trick again, and so on.

In order to put this idea into practice, we introduce a "magic" command called [as] (for "assume"). By pressing that button the current goal is proved without any further work. Note that we can press several times [as] in the course of the same proof. If the proof eventually "succeeds" after that, we would like to resume it in order to prove the missing parts. For this, we press the button [r1] (for "refinement of the current proof tree"). When doing this, the Tree Window splits and the former proof (containing occurrences of [as]) becomes the "old" proof. The prover automatically replays the proof from the beginning to the first occurrence of [as]. At this point, it gives back the control to the user as shown in the figure. A little arrow "⇒" indicates where we are in the old proof tree as shown in the figure.

Once the missing part is proved (possibly thanks to some further [as] applications), the prover automatically performs that part of the old proof situated between the current occurrence of [as] in the

old proof tree and the next one. It then gives back control to the user as explained, and so on (unless the end of the old proof is reached).

ds	fh	eh	he	$T1 = T2$
ds	fh	mh		$P \;\Rightarrow\; Q$
ds	fh	dc		$A \;\vee\; B$
ds	fh	ph		$\forall x \cdot (R_x \Rightarrow S_x)$
ds	fh	rx		$\exists x \cdot (T_x \;\wedge\; U_x)$
ds	fh	rn		$\neg\,(C \;\wedge\; D)$
ds	fh	rm		$E \in \{x \mid F_x \wedge G_x\}$
ds	fh	ri		$S \subseteq T$
ds	fh	rn		$X \neq \emptyset$
ds	fh	eq		$E \in S$
ds	fh	ov		$P((f \lhd g)(E))$
–	–	–	–	– – –
ae	ap	ct		$F \in S$
–	–	–	–	– – –
sh	sg	it	hp	
p0	p1	pb	bg	
ah	dc	re	st	
qu	op	pt	ba	
r1	r2	as		

zm	G
mk	**ah**
zm	$L1$
mk	**pp**
mk	**ah**
zm	$L2$
zm	**?**
– – – –	– – – – –
zm	G
mk	**ah**
zm	$L1$
mk	**pp**
mk	**ah**
zm	$L2$
zm $\Rightarrow$	**as**
mk	**pp**

When reentering a proof which was left using the [qu] button (for "quit") with some pending occurrences of [as], we may first decide to restart the proof from scratch and then figure out later that it was a mistake after all. At this point, we may decide to restart our proof from the genuine "old" proof, which was in place when reentering the proof. Pressing the button [r1] then is not what we want to do. There exists for this a second refinement button called [r2] (for "refinement of the old proof tree"), which reinstalls the current proof within the initial old framework.

5 Discussion and Related Works

In this section, we first make precise the main principles which have guided us in the realization of this interface, then we shall briefly compare our work with similar efforts. One of the reasons why we have implemented this interface under **Emacs** is that we consider (with many others) that the making of an interactive proof is a sort of *specialized text editing*. In this context, **Emacs** clearly provides us with a number of features which are not very interesting to reinvent. The main principle that has guided us in the development of this interface is one of *economy*. We have tried to minimize some "gestures" as explained in what follows.

We first tried to *minimize the need for eye movement* of the user so that he can really concentrate on its main proving activity. In fact, we discovered that the obligation to frequently move eyes is a very disturbing activity for someone who is supposed to think. This is the reason why we have banned the use of menus which are very common in all sorts of interfaces. In our case, a menu is very disturbing because it hides for a moment the contents of the screen. It is also disturbing because you have to scan it in order to click on the command you want to launch. A consequence of this choice implies the permanent presence of buttons on the screen.

In order to *minimize the number of clicks* (another very frequent gesture), we have placed these buttons next to the part of the formal text to which they apply. This avoids to first select, say, a specific hypothesis and then apply a command to it. In fact, by its position on the window the button is automatically devoted to the corresponding hypothesis or to the goal. But as the number of buttons placed next to each hypothesis or the goal could then be quite large and most of the time useless, we have made the interface *dynamic-discovering which buttons are applicable* at each proof step. This has proved to be very efficient since it was discovered that *no more than four different buttons* can be used at a given moment on an hypothesis, among the fifteen different ones that could be applied.

To *minimize the moves of the right hand*, which we would like to stay almost permanently on the mouse, we have taken advantage of what is already provided by **Emacs**: the left click is used to point to a piece of text, the central click is used to move a pointed text to the editing field and also to launch a command. We have explicitly programmed the right mouse click to launch [sh] (for "searching hypotheses") so that this very frequent activity is performed in a very efficient way.

We have also managed to *minimize the moves of the mouse*. For example, when a command devoted to a specific hypothesis requires an extra text parameter, we do not need to move the text in question to the editing field: as a short circuit, it is sufficient, if the text fragment is already somewhere on the screen, to point to it with the mouse and then press the desired button. Another way to minimize the moves of the mouse is provided by the presence of specialized editing fields that are close to the quantified variables in a universal hypothesis or in an existential goal. These fields are thus closed to the corresponding buttons, namely either [ph] or [se].

As a result, it is *extremely rare to have to press any key* on the keyboard. As explained above, the right hand stays almost all the time on the mouse, it is not necessary to move it from the mouse to the keyboard and vice-versa. All such small optimizations may seem very superficial at first glance, but we think on the contrary that they are very important in that they allow the user not to be disturbed by many stupid clerical activities.

And last but not least, we want to *minimize the number of syntactic errors*. Our interface, by dynamically producing the buttons associated with a proper goal or hypothesis, guarantees that no button can be pressed wrongly. In fact the only error that may occur is the forgetting of some text parameters, in which case the corresponding button has a void action. For example, if you press [sh] (for "searching hypotheses") without pointing to a text pattern, no search is undertaken. The only visible error that may occur comes from providing a text parameter which is not syntactically correct. In this case, an error is reported.

Another general principles that we have followed in this project was to construct this interface around the concept of *proof*, not that of prover. In other words, we have always tried to discover the sort of features that would help a person doing a (manual) proof within a Sequent Calculus by applying some inference rules in backward mode. This has guided us in the development of the proposed organization with two windows (Sequent and Tree) which are directly connected to the corresponding proof practice as indicated in section 4.1.

As a matter of fact, a certain "proving style" has emerged from using this interface. It can be summarized by the following steps to be performed recurrently:

– Simplify goal and hypotheses: rd, ri, rm, rn, ru, rx, eh, he, ov, ae, ap[3].

[3] We remind the reader that in **Appendix 3** a table contains a summary of the various commands.

– Decide for an "interesting" strategy: ct, fh, dc, mh, eq.
– Propose an "interesting" instantiation: ph, se.
– Invent an "interesting" lemma: ah.
– Decide to temporary postpone the proof of a certain goal: as.
– Refine the proof: r1, r2.
– Select "interesting" hypotheses: sh, sg, sl.
– Launch automatic procedures: p0, p1, pb.

Notice that, among these points, the first one (simplification) is partly performed automatically while in expert mode, whereas the ones qualified to be "interesting" require some invention on the part of the user. Note that this "style" covers 28 commands among the 42 we have presented. In fact, the 14 remaining ones (a few more in the real interface) are used at any moment to show more or less of each window or to navigate within the proof.

Our approach could not be compared with some far more ambitious generic interface project such as *Proof General* [4]. Clearly our project is far less "general". This is coherent with what we have said in the introduction about our prover which is also far less general than those to which *Proof General* is applied (Coq [5], Phox [13], LEGO [3], Isabelle [12], etc.). It seems to us that *Proof General* is directed towards *certain kinds of proof assistants* (not proofs), which, in spite of some technical differences between them, are, in a sense, quite close in their appearance and mode of interaction (scripts). What we have in common with *Proof General* however is the fact that our interface (also written under **Emacs**) does not deal with the proof, which remains the realm of the original prover.

Another comparison could be made with the technique of *Proof by Pointing* [14]. Our dynamic handling of buttons situated next to their domain of application (hypothesis, goal) has, in a sense, a similar effect as that of "proof by pointing". Our technique of *Proof by Clicking*, however, can associate several possibilities dedicated to the same piece of formal text. The button is also clearly showing to the user the action which he is going to perform.

A final interesting comparison can be done with the KIV proof assistant [9], which also uses a proof tree window as well as a sequent window. It is not clear however from the documentation at our disposal whether the proof tree could be refined as we have explained in section 4.13 above.

6 Conclusion

In this paper, we have presented an interface to be used with **Atelier B** in order to perform formal proofs of statements written within Set Theory. This interface has been used to teach logic and formal proof to students with a limited background in this area. The result of this experiment was extremely encouraging: students were very quickly acquainted to the tool and able to perform non-trivial proofs in a short time. It seems to us that this kind of approach where you just creatively prepare the work for a rather powerful prover able to perform most of the very mechanical work is an interesting one. On the following web page, you can see more about this interface: www.loria.fr/~cansell/cnp

Acknowledgements

We would like to thank M. Sintzoff for very valuable advice concerning this work, in particular that part devoted to the proof tree. Many thanks also to S. Mertz, P. Gibson, and L. Voisin for their very careful reading of various drafts of this text.

References

1. J.-R. Abrial, D. Cansell, and G. Lafitte. "Higher-Order" Mathematics in B. In *ZB'2002 – Formal Specification and Development in Z and B*, number LNCS 2272 in Lecture Notes in Computer Science, pages 370–393, 2002.
2. J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.
3. D. Aspinall. *LEGO Manual version 1.3.1*. Edimburgh University, Edimburgh (UK), 1998. www.lama.dcs.ed.ac.uk/home/lego.
4. D. Aspinall. Proof general - a generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number LNCS 1785 in Lecture Notes in Computer Science. Springer, March 2000.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual – Version V6.3*, July 1999.
6. ClearSy, Aix-en-Provence (F). *Atelier B, Version 3.6*, 2001. www.atelierb.societe.com.
7. M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., 1996.
8. M. J. C Gordon. Hol : A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, 1988.
9. D. Hutter, B.Langenstein, C.Sengler, J.H. Siekmann, and W. Stephan. Deduction in the verification support environment (vse). In *Proceedings, International Symposium of Formal Methods Europe (FME)*, 1996. Springer-Verlag LNCS 1051.
10. J.A. Kalman. *Automated reasoning with Otter*. Rinton Press, 2001.
11. S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language. Technical report, SRI International, June 14, 1993 1993.
12. L. Paulson. *Isabelle A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
13. C. Raffalli. *The Phox proof assistant version 0.8*. Université de savoie, Chambéry (F), 2002.
14. G. Kahn Y. Bertot and L. Théry. Proof by Pointing. In *Theoretical Aspects of Computer Software*, April 1994. Springer-Verlag LNCS 789.

APPENDIX 1: A Sample of Set-Theoretic Statements Proven Automatically by pp

Elementary Set Theory:

$$a \times b \subseteq c \times d \;\wedge\; a \neq \emptyset \;\wedge\; b \neq \emptyset \;\Rightarrow\; a \subseteq c \;\wedge\; b \subseteq d$$

Generalized Set Operations:

$$s \in \mathbb{P}(\mathbb{P}(S)) \;\wedge\; t \in \mathbb{P}(\mathbb{P}(T)) \;\Rightarrow\; \bigcup_{x \in s} x \cap \bigcup_{x \in t} x \;=\; \bigcup_{(x,y) \in s \times t} x \cap y$$

Operations on Relations:

$$p \in s \leftrightarrow t \;\wedge\; q \in t \leftrightarrow u \;\wedge\; r \in u \leftrightarrow v \;\Rightarrow\; ((p \,;\, q) \,;\, r) \;=\; (p \,;\, (q \,;\, r))$$

$$r \in s \leftrightarrow t \;\wedge\; a \subseteq s \;\wedge\; b \subseteq t \;\Rightarrow\; r[a] \subseteq b \Leftrightarrow a \subseteq \overline{r^{-1}[\,\overline{b}\,]}$$

Operations on Functions:

$$f \in s \rightarrow t \;\wedge\; b \subseteq t \;\Rightarrow\; f^{-1}[\,\overline{b}\,] \;=\; \overline{f^{-1}[b]}$$

$$f \in s \nrightarrow t \;\wedge\; b \subseteq t \;\Rightarrow\; f[\,f^{-1}[\,b\,]\,] \;\subseteq\; b$$

Injections and Surjections:

$$f \in b \rightarrowtail c \;\wedge\; r \in a \leftrightarrow b \;\wedge\; s \in a \leftrightarrow b \;\wedge\; (r \,;\, f) = (s \,;\, f) \;\Rightarrow\; r = s$$

$$f \in a \twoheadrightarrow b \;\wedge\; \wedge\; r \in b \leftrightarrow c \;\wedge\; s \in b \leftrightarrow c \;\wedge\; (f \,;\, r) = (f \,;\, s) \;\Rightarrow\; r = s$$

Equivalence Relations:

$$f \in s \rightarrow t \;\wedge\; r = (f \,;\, f^{-1}) \;\Rightarrow\; \mathsf{id}\,(s) \subseteq r \;\wedge\; r = r^{-1} \;\wedge\; (r \,;\, r) \subseteq r$$

APPENDIX 2: A Higher Order Statement Proven Interactively

We first give the definition of the set of all *filters* that can be built on a set S:

$$\begin{array}{ll} filter = \{\, f \mid f \in \mathbb{P}(\mathbb{P}(S)) & \wedge \\ \qquad \forall\,(A,B) \cdot (\, A \in f \;\wedge\; B \subseteq S \;\wedge\; A \subseteq B \;\Rightarrow\; B \in f\,) & \wedge \\ \qquad \forall\,(A,B) \cdot (\, A \in f \;\wedge\; B \in f \;\Rightarrow\; A \cap B \in f\,) & \wedge \\ \qquad S \in f & \wedge \\ \qquad \emptyset \notin f\,\} & \end{array}$$

This is followed by the definition of the set of all *ultra-filters*:

$$ultraf = \{\, f \mid f \in filter \;\wedge\; \forall\, g \cdot (\, g \in filter \;\wedge\; f \subseteq g \;\Rightarrow\; f = g\,)\,\}$$

The main very classical completeness theorem that we want to prove is the following:

$$U \in ultraf \;\wedge\; C \subseteq S \;\Rightarrow\; C \in U \;\vee\; \overline{C} \in U$$

where $\overline{C}$ is a shorthand for $S - C$. For this, we first prove the following lemma:

$$U \in ultraf \;\wedge\; C \subseteq S \;\wedge\; C \notin U \;\Rightarrow\; U = \{\, X \mid X \subseteq S \;\wedge\; C \cup X \in U\,\}$$

APPENDIX 3: Summary of Commands

ae	Abstract expression	4.4	**p1**	Apply **pp** on extended conclusion	4.3
ah	Add hypothesis (lemma)	4.5	**pb**	Apply **pb** on sequent	4.3
ap	Abstract predicate	4.4	**ph**	Instantiate universal hypothesis	4.6
as	Assume without proof	4.13	**pt**	Show current proof tree	4.12
ba	Backtrack	4.11	**qu**	Quit	4.12
bg	Switch to beginner mode	4.10	**r1**	Refine current proof	4.13
ct	Prove by contradiction	4.5	**r2**	Refine old proof	4.13
dc	Prove by cases	4.5	**rd**	Remove disjunction	4.7
dl	Delete hypothesis from *cached*	4.2	**re**	Reset proof	4.12
ds	Deselect hypothesis from *selected*	4.2	**ri**	Remove inclusion	4.8
eh	Apply equality (left to right)	4.4	**rm**	Remove membership	4.8
eq	Try set equality	4.9	**rn**	Remove negation	4.7
fh	Prove false hypothesis	4.5	**ru**	Remove universal quantification	4.7
he	Apply equality (right to left)	4.4	**rx**	Remove existential quantification	4.7
hp	Help	4.10	**se**	Instantiate existential goal	4.6
it	Interuption	4.3	**sh**	Search hypotheses in *hidden*	4.2
mh	Modus ponens on hypothesis	4.5	**sg**	Search hypotheses in *searched*	4.2
mk	Perform a series of commands	4.11	**sl**	Select hypothesis from *searched*	4.2
op	Show old proof tree	4.12	**st**	Do one step of old proof	4.11
ov	Treat overriding (by cases)	4.9	**xp**	Switch to expert mode	4.10
p0	Apply **pp** on conclusion	4.3	**zm**	Zoom/unzoom on proof tree	4.11