Chapter 6

Interactive Proof Development

Whenever an object of kind `STM` is opened, Nuprl's proof editor will be invoked on it. The proof editor provides an interactive method for constructing and modifying proofs in a highly visual fashion. Users enter a goal statement and develop its proof in a *top-down* fashion: proof goals will be *refined* into "smaller" subgoals until all subgoals represent basic axioms or already proven facts.

In this chapter we will first describe the general structure of Nuprl proofs and then discuss the features and usage of the proof editor.

6.1 Proof Structure

Nuprl's inference system is based on the notion of *sequents*. These are objects of the form

$$H_1, \ldots, H_n \vdash C \ \lfloor \mathsf{ext}\ t \rfloor$$

which are read as "*under the assumptions H_i we can prove that the type C is inhabited by some member t*". C is called the *conclusion* of the sequent and the H_i the *hypotheses*.[1] A hypothesis is either an *assumption* A_i or a *type declaration* $x_i{:}T_i$. A type declaration $x_i{:}T_i$ is considered to bind free occurrences of the variable x_i in the terms to the right of it, that is the hypotheses $H_{i+1}, \ldots, H_n$ and in the conclusion C. The expression t is called the *extract term* of the sequent. It will be constructed *during* the proof and remains unknown up to its completion.

Sometimes we refer collectively to the hypotheses and the conclusion of the sequent as *clauses*. The word *goal* is used either to refer to a whole sequent or to just the conclusion. Which should be clear from context.

Nuprl's inference rules refine sequents, obtaining subgoal sequents whose proofs would suffice to validate the original goal. *Primitive rules* (see Section 6.1.3.1 below) are usually characterized by rule schemata that match placeholders for the hypotheses and conclusion against a goal sequent and instantiate subgoal sequents accordingly. They also describe how to construct the extract term of the goal sequent from extract terms of the subgoal sequents (see Section 8.1 for details). *Tactic rules* (Section 6.1.3.2) combine several primitive rules into a single inference rule.

Proofs are trees whose nodes contain a *goal* sequent and a *refinement* slot. The refinement slot usually contains an inference rule and the goal sequents of the *children* of the node are the subgoals resulting from applying this rule to the node's goal. If the refinement slot is empty, the node has no children and is considered *unrefined*.

[1] The older Nuprl literature uses `>>` instead of the turnstyle symbol $\vdash$ to separate hypotheses from the conclusion.

A proof is *complete* if it has no unrefined nodes, which means that the goals of the leaf nodes are completely proven by their inference rules. In this case, the *top goal* of the proof, i.e. the goal of the root node, is called a *theorem*. The extract term of a theorem can be constructed bottom-up, using the instructions contained in each of the inference rules occurring in the proof.

In NUPRL, sequents, rules, and proofs are abstract data structures that are accessible from ML. Like all system components, they are implemented in the form of abstract terms and in principle, all term editing features described in Chapter 5 can be applied to them. However, all modifications to a proof have to pass through the proof editor before they are saved to the library, which ensures that all the proofs in the library are correct.

In the sections below we will briefly describe the essential aspects of these data structures.

6.1.1 Sequents

The data structure of sequents consists of a list of hypotheses $H_1, \ldots, H_n$ and a conclusion C. The conclusion is a proposition of NUPRL's logic, while each hypothesis may either be an assumption, i.e. a proposition, or a type declaration. For the sake of uniformity, assumptions are considered type declarations[2] with *invisible variables*. Furthermore, a sequent may contain *hidden hypotheses*. These are hypotheses that cannot be used for constructive reasoning but become accessible in parts of the proof that do not contribute to the extract term.

Sequents must be *closed*. Free variables in the conclusion must be declared in one of the hypotheses while free variables occurring in hypothesis H_i must be declared in on of the hypotheses $H_1 \ldots, H_{i-1}$. Obviously, all variables declared in the hypotheses have to be distinct.

Sequents do not explicitly contain extract terms, since extract terms are only constructed for complete proofs.

In NUPRL, sequents occur only within the nodes of a proof and are therefore considered identical to proof nodes with empty refinement slots. Usually NUPRL displays sequents vertically and explicitly numbers the hypotheses, so a sequent $H_1, \ldots, H_n \vdash C$ is displayed as:

$$\boxed{\begin{array}{l} \texttt{1.}\ H_1 \\ \quad\vdots \\ n\texttt{.}\ H_n \\ \vdash C \end{array}}$$

Variables, whose name starts with a `%` character are considered invisible and will not be displayed.

The system provides a few ML functions for accessing the components of a sequent.

```
var_of_declaration:    assumption -> var
type_of_declaration:   assumption -> term
is_hidden_declaration: assumption -> bool
mk_declaration:         (var # term # bool) -> assumption

conclusion:             proof -> term
hypotheses:             proof -> assumption list
mk_sequent:             (var # term # bool) list -> term -> proof
```

Advanced users may take advantage of these functions when developing proof tactics that analyze the contents of a sequent in order to determine appropriate inferences.

[2] Since NUPRL's type theory incorporates the *propositions-as-types* principle, which associates logical propositions with the type of all their proofs, all the clauses of a sequent contain types.

6.1.2 Proof Objects

Proof trees are implemented in Nuprl as *recursive data structure*, consisting of either

- an *unrefined* goal sequent g, or
- a goal sequent g, an inference rule r, and a list $p_1, \ldots, p_n$ of proofs.

Thus each sub-tree of a proof is considered a proof as well, which makes it possible to reason locally. The Nuprl proof editor (see Section 6.2 below) enables users to focus on any node of a proof tree and to view the corresponding sub-proof as a full proof object.

The sequent g in a proof object is referred to as the (*root*) *goal* of the proof and the goals of the proofs $p_1; \ldots; p_n$ in a refined proof are referred to as its *subgoals*. A proof is *good* if

1. every sequent in the proof is closed,
2. in every sequent all variables declared in the hypotheses are distinct, and
3. at every refined node of the proof tree, the subgoals are the result of applying the rule r to the root goal g.

A proof is *complete* if it is good and contains no unrefined nodes. A proof is *incomplete* if it is good but does contain unrefined nodes.

Each statement object in Nuprl's library is associated with a list of proof objects with the same root goal, sometimes referred to as the *main goal* of the statement object. The main goal must be an *initial sequent*, i.e. a sequent with an empty hypotheses list. The main goal is a *theorem* if at least one of its proofs is complete.

If a statement object is linked to more than one proof object, one of them is considered the actual proof. The proof editor enables users to switch between proofs for the same statement, which allows them to formalize different approaches to solving the same problem or to work on a "better" proof for a theorem while preserving the existing ones.

The system provides a few ML functions for accessing the components of a proof.

```
var_of_hyp:     int -> proof -> var
type_of_hyp:    int -> proof -> term

conclusion:     proof -> term
hypotheses:     proof -> assumption list
refinement:     proof -> rule
children:       proof -> proof list

mk_sequent:     (var # term # bool) list -> term -> proof
refine:         rule -> tactic
```

The function `mk_sequent` is the only way to build unrefined proofs in Nuprl, while `refine` is the only way for constructing functions that modify proofs from scratch. The proof editor implicitly makes use of these functions when a user initiates and refines a proof.

6.1.3 Refinement Rules

Refinement rules in Nuprl serve two purposes. They *decompose* a goal sequent into a list of subgoal sequents and they provide a *validation*, which transforms evidence for the validity of the subgoals into evidence for the validity of the original goal. Refinement rules are therefore implemented as functions that transform a proof into a list of (unrefined) proofs and a validation v. The validation in turn is a function transforms a list of proofs into a proof.

A refinement rule is *correct*, if the validity of the generated subgoals $g_1; \ldots; g_n$ implies the validity of the root goal g, and if the validation v transforms complete proofs $p_1; \ldots; p_n$ for the subgoals into a complete proof p for the root goal. Logically, validations only prove for the correctness of the rule applications. Computationally, however, they provide evidence for the validity of the main theorem and can therefore be used for building the extract term of a theorem.

NUPRL distinguishes two kinds of rules. *Primitive refinement rules* are the basic rules of inference that constitute the formal theory on which NUPRL is founded. *Tactic rules* are the basis for automating the application of primitive rules. Tactics can only be constructed by combining (converted) primitive rules and other tactics into a new refinement rule. This guarantees that the correctness of proofs generated by tactics only depends on the correctness of the primitive inference rules, which in turn are justified semantically.

6.1.3.1 Primitive Refinement Rules

NUPRL's primitive refinement rules are all introduced by rule objects (see Section 8.1.1) in the system's library. The current system has primitive rules for a constructive *type theory* that is closely related to Martin-Löf type-theory (see Appendix A.3 for a complete list of rules). All NUPRL proofs are eventually justified by these primitive rules.[3]

The correctness of a NUPRL proof depends only on the correctness of these rules and of NUPRL's *refiner*. The *refiner* is a fixed piece of LISP that applies primitive rules to unrefined leaves of proofs.

In contrast to previous releases of NUPRL, users of NUPRL 5 cannot invoke primitive rules directly. The proof editor expects users to enter tactics when refining a proof, which means that primitive rules have to be converted into tactics before they can be applied (see Section 8.1.3). Furthermore, using primitive rules would require users to understand how mathematical concepts are coded within type-theory. Tactics operate at a higher level of reasoning and are much easier to deal with.

6.1.3.2 Tactic Rules

As explained in detail in Chapter 8, tactics are ML functions that enable one to automate application of primitive rules. If one applies a tactic to a proof and the tactic does not fail, then the tactic returns a proof built entirely from primitive rules. The NUPRL proof editor treats a tactic like a single inference rule and only displays the unrefined leaves of the generated proof tree. Users may view the generated primitive proof on demand.

More precisely, if a tactic rule is applied to a proof node, the NUPRL proof editor will perform the following steps.

1. The ML text of the tactic is interpreted by the ML system and applied to the current proof node, resulting in a proof tree p with unrefined leaves $p_1, \ldots, p_n$. Note that the root goal of p is always identical to the goal sequent of the current proof node and that p also includes validations.
2. Instead of simply replacing the proof node by the proof p, the editor stores p together with the tactic text in a *tactic rule*.
3. The tactic rule is inserted as refinement of the proof node and the leaves $p_1, \ldots, p_n$ become the new children of the node.

[3] This may change in the future, as NUPRL may also accept proofs provided by external proof engines without transforming them into type-theoretical proofs

The display of the tactic rule hides the proof tree p. When one views a tactic rule refinement, one only ever sees the text of the tactic. From a logical point of view, it is not strictly necessary to keep p around at all, after the tactic has executed. However, it is necessary to access the validation contained in the proof when constructing the extract term of the main theorem.

Running a tactic as a refinement rule makes it appear in a proof as a high level rule of inference, and consequently greatly increases the readability of proofs.

Note that applying a tactic rule to an already refined proof node overwrites the existing proof tree. The editor first discards the existing refinement of the node and then proceeds as described above. The previous refinement, however, remains stored in the library and can still be accessed and reinserted into the proof.

6.2 The Proof Editor

The proof editor is designed to support the top-down *refinement* style generation of proofs. The refinement style entails repeatedly choosing an unrefined leaf node of a proof and a rule to try on that node. If the rule applies, the NUPRL system changes the node to a refined node, and automatically generates appropriate children nodes.

The proof editor generates windows onto sections of proofs. One can have windows open on different proofs at the same time, and even view multiple proofs of the same theorem. In the latter event, one proof is the *main proof* while the other ones are *backup proofs*.

6.2.1 Proof Window Format

Each proof window is associated with a node of a proof. It shows the goal sequent at that node, the refinement rule (if any) at that node, the immediate subgoals, and the proofs (if any) of these subgoals, as long as they fit into the window. Figure 6.1 shows an example of a window onto a refined node of a proof and an example of a window onto an unrefined node of a proof.

```
- PRF: intsqrt
① # top 1
②
③ 1. x:ℕ
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
④ BY NatInd 1
⑤ # 1 1
   .....basecase.....
   ⊢ ∃y:ℕ. y²≤0 ∧ 0<(y+1)²
⑥ BY exR ⌜0⌝
   There is 1 hidden subgoal
⑤ # 1 2
   .....upcase.....
   1. x:ℤ
   2. 0<x
   3. ∃y:ℕ. y²≤x-1 ∧ x-1<(y+1)²
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
⑤ BY
```

```
- PRF: intsqrt
① # top 1 2
② .....upcase.....
③ 1. x:ℤ
   2. 0<x
   3. ∃y:ℕ. y²≤x-1 ∧ x-1<(y+1)²
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
④ BY
```

Figure 6.1: Proof window on refined and unrefined proof node

The title of each window contains the indicator `PRF`, denoting the kind of the object being viewed, and the name of the statement object associated with the proof. The numbered parts of these windows are as follows:

① The `#` indicates that this proof node is considered *incomplete.* Other symbols used here are `*` for *complete*, and `-` for *bad* proofs.
The `top 1` and `top 1 2` are tree addresses of the nodes being viewed. The left window shows the first child of the root of the proof and the right window shows the second child of that proof node.

② Some nodes are annotated to indicate the nature of the proof goal. Typical annotations are `wf` for well-formedness subgoals or `upcase` and `downcase` in inductive proofs. These annotations may be used by tactics but are mainly intended to assist the users.

③ This is the goal sequent of the proof node. Hypotheses are numbered and listed vertically. The conclusion is at the bottom after the turnstyle.

④ This is a tactic which was executed on the goal ③ above in order to generate the subgoals ⑤ below. The `BY` is part of the proof node display, and is not part of the tactic.
In an unrefined proof, there is an empty text slot after the `BY`.

⑤ These are the subgoals of the proof node. Each subgoal comes with a status (`*`, `#`, or `-`), an address, and the subgoal sequent. For brevity, only hypotheses that have changed or been added are displayed in the subgoal sequents.

⑥ If a subgoal has a proof, it is being displayed immediately below the subgoal as long as it fits into the window. If a proof is too large to be shown, the editor will only display its top-level tactic and indicate that its subgoals are hidden. In this case users have to move into the corresponding node to see further details or to continue editing the proof.

Sometimes the proof window is too short to display all the goal, rule, and subgoals. In this case the cursor motion commands described in Section 5.5 will automatically scroll the window. One can of course also resize the window.

6.2.2 Proof Motion Commands

←	move to sibling to immediate left
→	move to sibling to immediate right
⟨M-a⟩	move to left-most sibling
⟨M-e⟩	move to right-most sibling
↑	move up to parent node
↓	move down to selected subgoal
⟨M-z⟩	zoom in on current node
⟨C-↑⟩	move up to top of proof
⟨C-M-j⟩	jump to next unrefined node

The keyboard commands for navigating through a proof tree are summarized in the table below. In addition to these, most of the motion commands for terms described in Section 5.5 can be used for navigating through (the term-tree of) a proof window.[4]

[4]In contrast to previous releases, NUPRL 5 only uses arrow keys for proof-tree navigation. The emacs-like keyboard and mouse commands used in NUPRL 4 are now captured by the term editor, which has priority over the proof editor, and are thus reserved for navigating through the term tree of the proof window as described in Section 5.5.2.

The left-, right-, up-, and down-arrow keys move one node left, right, down, or up in the proof tree. If a node has many siblings, users may also use the key combinations ⟨M-a⟩ and ⟨M-e⟩ for larger jumps. In each case the proof window will *focus* on the new node, i.e. make it the root of the currently displayed proof tree. The cursor will be positioned in the refinement slot of that node. The proof window on the right of Figure 6.1, for instance, results from the window on the left by pressing ↓ first and then →. Pressing the ↑ key in the right window will again produce the window on the left.

Proof tree motion is relative to the node where the cursor is positioned. If the cursor is at the current root node, then pressing ←, →, or up ↑ will move the focus to a sibling or parent node that is currently not visible, while ↓ will focus on the first visible subgoal. If the cursor is at a subgoal of the current root node, then pressing an arrow key will move the focus relatively to that node, while pressing ⟨M-z⟩ will cause the proof window to focus on that node.

⟨C-↑⟩ causes the editor to jump to the root of the proof tree that is currently being edited, while ⟨C-M-j⟩ causes it to jump to the next unproven subgoal in that tree, using a preorder traversal of the proof tree. If there are none, ⟨C-M-j⟩ shifts the focus to the root of the proof tree.

6.3 Stating and Proving Theorems

The proof editor is invoked whenever a statement object is opened for viewing. Usually, this is done through the navigator by using either the right arrow key or the middle mouse button (Section 4.3.1.1). As proofs associated with statement objects are *not* copied when they are viewed with the proof editor, all changes made to proofs are immediately committed to the library. This is in contrast to editing objects of other kinds (abstractions, display forms, code objects, ...) where changes are only committed when one exits the object or explicitly asks for changes to be saved. Users who want to make tentative changes to a section of a proof should first create a backup proof (see Section 6.4.3) and then work on either of the two versions.

6.3.1 Editing The Main Goal

```
PRF not_over_and
# top
[goal]

BY
```

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))

BY
```

When a new proof window is opened, the window appears as depicted on the left above. A term cursor is positioned on an empty `[goal]` slot immediately below the root address. A user may now enter the main goal using the structured term editor (Chapter 5).[5] The window on the right, for instance, is the result of entering

␣all ↵ A ↵ prop ↵ i ↵ all ↵ B ↵ prop ↵ i ↵ implies ↵ or ↵ not ↵ A ↵ not ↵ B ↵ not ↵ and ↵ A ↵ B ↵␣

into the initial proof window. Pressing ↓ then moves the cursor into the text slot for proof tactics.

It should be noted that the proof goal is *not* committed to the library until the first refinement step has been executed. If a user closes the proof window after entering the main goal, the input will be discarded and the statement object remains empty. To commit a proof goal explicitly to the library, one may use the key combination ⟨C-M-g⟩. This will save the main goal into a proof object, which will be linked to the statement object that is currently being viewed.

[5] Currently, all input until the first ↵ will be ignored. This is an interface bug that will be corrected in the future.

Using ⟨C-M-g⟩ is required if one wants to change an already existing main goal. Simply editing the goal is not sufficient and an error message will be produced if a user tries to refine a modified proof goal that has not yet been committed.

Changing a proof by modifying either its main goal or one of the refinement steps will remove it from the proof window but not from the library. Previous versions of a proof may be recovered by walking through the proof editor history (see Section 6.4.2 for details).

6.3.2 Refining Proof Goals

To refine a proof goal, one uses the arrow keys or the mouse to move the cursor into the text slot for entering proof tactics and then types the name of the tactic to be applied. Tactics are ML functions that may have NUPRL terms and other parameters as arguments (see Chapter 8). The structure and special editing commands for tactics are the same as for `CODE` objects. NUPRL terms may be entered using the term editor after opening a term slot with ⟨C-o⟩ (Section 5.4.2). Other tactic arguments have to follow the syntax of the corresponding ML data types. Tactics may include ML comments (using `%` both as left and right delimiter) and newline characters, which will be inserted when a user types the ↵ key.

There are two possible modes for executing a tactic. In *synchronous mode*, initiated by pressing ⟨C-↵⟩, the tactic is sent to the refiner and the editor waits for the refinement process to be complete before allowing the user to continue. Pressing ⟨C-↵⟩ instead initiates *asynchronous refinement*, which allows the user to work on other proof goals while the proof goal is being refined. This is useful when executing complex tactics that may take a long time to complete.

Once a (synchronous or asynchronous) refinement is successfully completed, the proof is committed to the library and the proof window gets updated, showing the subgoals that were generated by applying the tactic. If the refinement fails, an error message describing the nature of the error and some debugging information will be inserted after the tactic and the proof goal will be marked as *bad*. The proof itself will not be changed.

Entering ⌞`D 0` ⟨C-M-↵⟩⌟, for instance, surrounds the tactic `D 0` by markers to indicate that it is currently being processed (see the window on the left below). Upon completion of the refinement, the editor removes the markers and inserts the resulting subgoals as shown in the window on the right below. The latter is also the result of entering ⌞`D 0` ⟨C-↵⟩⌟.

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY << D 0 >>
```

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY D 0
# 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY
# 2
.....wf.....
ℙ  ∈  𝕌'
BY
```

A refinement is always initiated for the proof goal at the current location of the cursor. Users are free to refine subgoals in any order and to modify already existing refinements. The latter will erase the proof tree at the current proof node and insert the result of the modified refinement into the proof window (see Section 6.4.1 for details). The old proof may, however, be recovered by walking through the proof editor history (see Section 6.4.2).

6.3.3 Generating Extract Terms

Proof steps are committed to the library as soon as a refinement step has been executed successfully. Users may therefore simply close the proof editor once a proof is complete. Like all other windows, the proof editor window will be closed by ⟨C-q⟩.

Pressing ⟨C-z⟩ instead will cause the proof editor to build the extract term of the proof before closing the proof window. As the extract term is constructed by assembling the validations contained in the refinements of each proof node, it can only be built if the proof is complete. Using ⟨C-z⟩ on an incomplete proof will close the proof window but pop up a window with an error message.

Term extraction is NUPRL's mechanism for supporting the *proofs-as-programs* principle. Extract terms describe the computational content of a proof and the algorithms that are synthesized while formally solving a program specification problem. These algorithms are proven to satisfy the specification and can be executed by the NUPRL term evaluator as described in Section 4.4.3.1.

Once created, extract terms and the corresponding proof extract tree can also be viewed from within the proof editor. Typing ⟨C-M-v⟩`pex` pops up the extract term of the saved proof, while ⟨C-M-v⟩`pet` shows the proof extract tree. The windows below, for instance, show a complete proof of the statement ⌞`∀A,B:ℙ. (¬A ∨ ¬B) ⇒ ¬(A ∧ B)`⌟ and the term extracted from that proof.

PRF not_over_and

```
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
* 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
* 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

extract: not_over_and

```
Refresh*  Quit*
 (get_prf_extract Obid: not_over_and)
  @ 2:15PM 9/10/2002

 λA,B,%,%1.case % of inl(%2) => Ax | inr(%3) => Ax
```

6.4 Advanced Editing Features

6.4.1 Modifying existing refinements

In previous releases of NUPRL, modifications to an existing refinement of a proof node caused the entire proof tree below that node to be lost. Often however, some of the subgoals generated by the new refinement could be solved by replaying certain parts of that proof tree. Since NUPRL 5 immediately commits all successful refinement steps to the library, it is possible to reuse these steps when a proof is modified.

The default refinement, initiated by pressing ⟨C-↵⟩ (or ⟨C-M-↵⟩) refines the goal at the current proof node and reuses the proofs of subgoals that were already refined in the previous proof. This is useful, when some of the newly created subgoals are identical to subgoals of the previous refinement of that node. In the example below, for instance, replacing the tactic ⌞`D 0`⌟ by ⌞`D 0 THENW Auto`⌟ generates the same "main" subgoal and initiating the refinement with ⟨C-↵⟩ preserves the proof of that subgoal (center window). In previous releases, that proof would have been lost and the whole proof would have become incomplete (right window).

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto

* 1 2
.....wf.....
A ∧ B ∈ ℙ

BY Auto
```

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto
```

PRF not_over_and

```
# top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

#  1 1

4. A ∧ B
⊢ false

BY
```

Instead of reusing the proof of a specific subgoal, users may also want to reuse the tactics that had been applied to the subgoals of the previous refinements. Using ⟨M-↵⟩ instead of ⟨C-↵⟩ causes NUPRL 5 to reuse the *tactic tree* of the previous proof for subsequent refinements. This is useful when the new refinement generates subgoals that are different from the previous ones but that can be solved in the same way.

If users want to discard the original proof and just execute a single new refinement step, they may enter the keyboard macro ⟨C-M-r⟩st. "step" refinement emulates the behavior of previous NUPRL releases, which is meaningful if reusing the previous proof would be time-consuming and of little use for the new proof.

Often, users want to rearrange a proof in a way that each refinement step corresponds to an argument a human would make. Usually this means assembling several refinements steps into one and adding a comment that describes the logical meaning of that step. NUPRL offers some support for this technique.[6] Pressing ⟨C-M-r⟩kr will collect *all* the inference steps of the proof tree starting at the current node into a single refinement step. This step consists of a tactic that combines the individual steps using the tacticals THEN and THENL. The window on the right below shows the result of applying this command to the proof on the left. Extensions of this command (like accumulating only a certain amount of steps or inserting comments) will be added in the future.

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto

* 1 2
.....wf.....
A ∧ B ∈ ℙ

BY Auto
```

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENL [D 3 THEN Auto; Auto]
```

Pressing ⟨C-M-r⟩dk turns a "kreitzed" proof back into its original form. It has no effect on refinements that were not previously generated by ⟨C-M-r⟩kr.

[6] At some time in the past, the name *kreitzing* was introduced for this technique. For lack of a better name, it is still called that way.

Similarly to the path stack of the navigator (Section 4.3.1.3) the proof editor enable users to jump between commonly used positions in the proof tree. Pressing ⟨C-`h`⟩ marks the current proof address and stores it in an address stack. Pressing ⟨M-`h`⟩ jumps back to that position and removes the address from the stack.

Nuprl also enables users to copy a pattern of reasoning used in one proof to another proof. Pressing ⟨M-`k`⟩ copies the proof at the current node to a proof stack. ⟨C-`y`⟩ pastes the proof on top of the proof stack into the current proof node and removes it from the stack. Pasting a proof means re-executing the tactics of its tactic tree until one of the tactics fails or the complete tactic tree has been reused.

6.4.2 Proof History

In the course of proof development, Nuprl's proof editor stores each refinement as a separate object in the library. As a consequence, users may walk backward and forward through the proof history and continue the refinement of previous versions of the proof.

Pressing ⟨C-←⟩ reverts the proof window to the previous proof in the proof history while ⟨C-→⟩ moves to the next proof, if there is one. If a user refines one of the previous proofs in the history, all subsequent proofs in that history will be discarded the new proof will be the last proof in the new history. Users who want to save an older version of a proof to the library without modifying it, can do so by pressing ⟨C-`x`⟩⟨C-`s`⟩ (just moving through the history does not change the stored proof). Pressing ⟨C-M-`e`⟩ reverts the proof window to the proof that was last stored in the library.

The proof history is only preserved through a proof editing session. Once the proof window is closed the history is discarded.

6.4.3 Backup Proofs

In addition to using a temporary proof history Nuprl 5 allows users to create and edit backup proofs that are linked permanently to a statement object. This makes it possible to elaborate and keep different proofs of the same statement and to preserve several interim versions of a proof attempt until they are not needed anymore.

Pressing ⟨C-M-`c`⟩ will create a backup copy of current proof and save it to the library. This proof will usually remain invisible. Pressing ⟨M-→⟩ pops up a proof editor window for each backup proof of the current statement.

Users may create multiple backup copies, including backups of backup proofs. The proof from which all these backups were created will remain to be the *main proof* unless the user explicitly changes that by pressing ⟨C-M-`f`⟩. This will declare the current proof to be the main proof from now on. To remove a specific proof, users may type ⟨C-M-`d`⟩ into the proof window of that proof. Typing ⟨C-`x`⟩⟨C-`b`⟩ deletes all backup proofs.

It should be noted that deleted backup proofs and the temporary proof history are still contained in the library but not linked to the statement object anymore. Expert users may still be able to retrieve them if that should be necessary.

6.4.4 Views of Proofs and Refinements

Nuprl enables users to look at proof trees in different ways. In the default view, the proof window displays the addresses, goals, and refinements of each visible node. Pressing ⟨C-M-`t`⟩ will show the tactic tree instead, ⟨C-M-`a`⟩`a` the proof structure (i.e. the address tree), and ⟨C-M-`v`⟩`v` the goal tree. ⟨C-M-d⟩ will revert to the default view. Examples of these four views are displayed in the windows below.

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto
```

PRF not_over_and

```
* top 1
BY D 0 THENW Auto

1 1
BY D 3 THEN Auto
```

PRF not_over_and

```
* top 1

* 1 1
```

PRF not_over_and

```
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

* 1 1

4. A ∧ B
⊢ false
```

Users may also want to create views on specific proof parts. Pressing ⟨C-M-p⟩ pops up a new proof window whose main proof is the current node. This window is considered a scratch window. Users may execute refinements in this window but these refinements will not affect the original proof. If a statement has several proofs, pressing ⟨C-M-i⟩ will pop up a window that contains pointers to other proofs of this same statement.

PRF not_over_and

```
# top 1 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
4. A ∧ B
⊢ false

BY D 3

#  1 1 1

3. ¬A
4. A ∧ B
⊢ false

#  1 1 2

3. ¬B
4. A ∧ B
⊢ false
```

PRF not_over_and

```
# top
1. A : ℙ
2. B : ℙ
3. (¬A) ∨ (¬B)
4. A ∧ B
⊢ False

BY !rule_instance{direct_computation_hypothesis:o}(#3 [1:(¬A) ∨ (¬B)] ())

# 1
3. ¬A + (¬B)
4. A ∧ B
⊢ False

BY !rule_instance{unionElimination:o}(#3 %2 %3 ())

# 1 1
4. ¬A
5. A ∧ B
⊢ False

BY !rule_instance{thin:o}(#3 ())

# 1 1 1
3. ¬A
4. A ∧ B
⊢ False

BY

# 1 2
4. ¬B
5. A ∧ B
⊢ False

BY !rule_instance{thin:o}(#3 ())

# 1 2 1
3. ¬B
4. A ∧ B
⊢ False

BY
```

The keyboard macros ⟨C-M-h⟩ and ⟨C-M-l⟩ are used to show internal details of a refinement step. ⟨C-M-h⟩ pops up a window that shows the the actual steps performed by the refinement tactic, while ⟨C-M-p⟩ shows the proof on the level of primitive inferences. An example of a primitive proof tree corresponding to simple refinement step is shown above.

6.4.5 Miscellaneous Features

Users may *print* proofs by typing ⟨C-M-`m`⟩ into the editor window. This will print a snapshot of the current proof to a file `nuprlprint/`*stm-name* in the user's home directory, where *stm-name* is the name of the corresponding statement object. The directory `nuprlprint` must already exist.

The generic commands for closing and saving editor windows have a slightly different meaning in the context of proof editing. As usual, ⟨C-`q`⟩ *closes* a proof window without initiating any modifications. However, since proofs are saved after each refinement step, all editing steps that were performed after opening the proof editor are already committed to the library and will not be discarded. *Saving* a proof window with ⟨C-`z`⟩ therefore has a somewhat stronger meaning than just saving its visible contents. In addition, an extract term will be created and saved, provided the proof is complete.

6.5 Customizing the Proof Editor

Table 6.5 summarizes the current commands of NUPRL's proof editor. Users may change the keyboard macros that initiate these commands by editing the file `mykeys.macro`, which is also used for modifying the key bindings of the term editor (Section 5.8). Modifications should be done carefully to avoid that they affect the navigator and term editor as well.

6.6 Troubleshooting

Most proof editing mistakes can easily be corrected by typing the *undo* command ⟨C-`_`⟩ or by walking backwards through the proof history with ⟨C-←⟩.

A common problem occurs when users try to delete a refinement rule as a whole with ⟨C-`k`⟩ or ⟨C-`c`⟩. This will delete the text slot for entering refinement tactics and leave a term slot, displayed as `[left]`. Usually the *undo* command ⟨C-`_`⟩ will bring back the original rule, which can then be modified using *text* editing commands. If the rule cannot be removed by text editing, for instance if it was created by ⟨C-M-`r`⟩`kr`, users should instead enter ⌞`itext_df`⌟ into the `[left]` slot to get an empty text slot.

Movement	
←	move to sibling to immediate left
→	move to sibling to immediate right
⟨M-a⟩	move to left-most sibling
⟨M-e⟩	move to right-most sibling
↑	move up to parent node
↓	move down to selected subgoal
⟨M-z⟩	zoom in on current node
⟨C-↑⟩	move up to top of proof
⟨C-M-j⟩	jump to next unrefined node
Inference	
⟨C-↵⟩	refine goal at current node
⟨M-↵⟩	refine goal reusing the tactic tree below
⟨C-M-↵⟩	asynchronously refine goal
⟨C-M-r⟩st	"step" refine
⟨C-M-r⟩kr	kreitz this subtree
⟨C-M-r⟩dk	de-kreitz this node
Copy/Paste:	
⟨C-h⟩	mark proof address
⟨M-h⟩	goto proof at address on stack
⟨M-k⟩	copy proof when selected
⟨C-y⟩	paste proof on stack into current proof node
Proof Editor History	
⟨C-←⟩	reverts window to previous proof in history walk
⟨C-→⟩	reverts window to next proof in history walk
⟨C-x⟩⟨C-s⟩	save proof in window to library
⟨C-M-e⟩	update the proof window with the current proof in the library
Saving and Deleting Proofs	
⟨C-M-g⟩	save goal to library
⟨C-M-c⟩	make backup copy of current proof
⟨C-M-f⟩	set current proof to be the main proof
⟨M-→⟩	bring up backup proofs
⟨C-M-d⟩	delete current proof
⟨C-x⟩⟨C-b⟩	delete all backup proofs
Alternative Views	
⟨C-M-p⟩	create scratch window containing the current sub-proof
⟨C-M-i⟩	pop up pointers to other proofs of the current statement
⟨C-M-d⟩	select default view mode
⟨C-M-t⟩	view the tactic tree
⟨C-M-a⟩a	view the address tree
⟨C-M-a⟩v	view the goal tree
Proofs Details	
⟨C-M-h⟩	show interior proof
⟨C-M-l⟩	show primitive proof
⟨C-M-v⟩pet	show extract tree
⟨C-M-v⟩pex	show extract term
Miscellaneous	
⟨C-M-m⟩	print current proof
⟨C-q⟩	close proof window
⟨C-z⟩	generate extract term and close proof window

Table 6.1: Proof Editor Keyboard Macros