A PROOF EDITOR FOR PROPOSITIONAL AND PREDICATE CALCULUS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Nikolaos Papaspyrou

January 1995

ABSTRACT

In many universities, lower-level math courses on propositional and predicate calculus usually avoid using formalism in definitions and proofs, at least in the beginning. When formalization is later introduced, the students are left with the impression that formalism is not essential or worth the effort. On the contrary, the Department of Computer Science of Cornell University adopts formalism in these courses from the very beginning. Students learn not to fear formalism and develop their reasoning abilities by constructing rigorous proofs.

However, the construction of formal proofs is not always easy. When the magnitude of the problem increases, it is easy to forget or neglect something and it is equally easy to loosen the formalism, either by carelessness or on purpose. A computer-based proof editor that checks each step of a proof can help solve these problems.

The subject of this thesis is the design and implementation of a computer program that facilitates the construction of formal proofs. The proof editor that has been developed can be used to familiarize students with formalism and help them construct rigorous and sound proofs. Special attention has been given to make this tool user-friendly and easily extendable to calculi other than propositional or predicate.

The program has been developed for the Apple Macintosh. It is written in C++ and is completely portable, except for the part implementing the user interface.

Biographical Sketch

Nikos Papaspyrou was born on January 26, 1971 in Athens, Greece. He attended the 7th High School of Athens and graduated in June 1988. In September 1988 he was admitted to the Department of Electrical Engineering and Computer Science, National Technical University of Athens (NTUA). He graduated in July 1993 with a GPA of 9.58 out of 10. He joined the Graduate School at Cornell University in August 1993 and completed the requirements for the M.Sc. Degree in September 1994.

While a student at the NTUA, Nikos Papaspyrou participated in three European Community Contests for Young Scientists (Copenhagen 1990, Zurich 1991 and Sevilla 1992) with three computer science projects which took second, first and first place in the greek national contest respectively. He speaks three foreign languages (English, French and —a little— German). Since 1985, he has played volleyball on the team of Pangrati, Athens, which is currently in the third division in Greece.

Acknowledgements

First and foremost, I would like to express my respect and thanks to Professor David Gries, who supervised my thesis and was always accessible and willing to help with the numerous problems that came up.

I would also like to thank Professor Thorsten von Eicken and Professor Steve Vavasis for serving on my special committee.

My sincere thanks also go to my first advisor, Professor Sam Toueg, and the support staff of the Department of Computer Science at Cornell University for their outstanding help while I was a graduate student.

Finally, I would like to thank my friends in Ithaca for the good times that we shared. Last but not least, my wholehearted thanks go to my family and friends in Greece for their love, encouragement and support.

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis presents an equational proof editor, following the style used in the text "A Logical Approach to Discrete Math" [GS93]. Several tools have been developed recently that can serve as formal proof editors. Among them, one should mention the following.

- the *Proxac* editor for proof and program transformation [vdS94a] [vdS94b], developed in California Institute of Technology. It tool is closest to a usable editor for proofs in the style of [GS93]. It has been implemented in Modula-3, which is the main reason why it cannot be used (until now, there is no reliable Modula-3 compiler for the Apple Macintosh). Other reasons are its inefficient user interface, the slightly different format it uses for proofs and the fact that it cannot be easily extended.
- the *NUPRL* proof development system [C$^+$86], developed in Cornell University. It focuses on automatic proof development using natural deduction. However,

the proof format that it uses and the complexity of this system make it inappropriate for lower-level math courses.

- the *MathPad* mathematical report writer [BVW94], developed in Eindhoven University of Technology. This tool is an editor, allowing the preparation of documents of mathematical nature. Although it can be used for the construction of formal proofs in propositional and predicate calculus, it is more oriented towards text rather than formulae manipulation and does not provide any checking of validity.

Chapter 2

Specifications

2.1 Introduction

This section describes the basics of the proof editor. We first define some of the terms that will be used frequently. The terms that are not defined have their usual meaning (e.g. type, identifier, constant, variable, operator, expression).

- *Fact*: an expression whose result belongs to the special type `boolean`. Not all boolean expressions are facts; only the ones that are valid (true in all states).
- *Axiom*: a fact that is considered to be *true* with no evidence.
- *Theorem*: a fact that is considered to be *true* and its proof is given.
- *Step*: the result of the transformation of an expression to another expression. The two expressions become related with a *connective* operator.
- *Proof*: a sequence of steps or other constructs of the proof editor that is considered to be the evidence that supports a theorem.

- *Rule*: a mechanism for deriving steps in a proof. An *inference rule*, as defined in [GS93], is a syntactic mechanism for deriving “truths”. Inference rules consist of a list of expressions, called *premises*, and an expression, called the *conclusion*. Such rules assert that, if the premises are assumed to be truths, then the conclusion is a truth also. Our proof editor has some predefined inference rules (e.g. *Leibniz's* rule) and provides a small programming language for creating user-defined rules.

The following paragraphs present a brief overview of the proof editor.

2.1.1 Facts, proofs and scopes

There are two basic types of facts: *axioms* and *theorems*. The difference between the two is that axioms become facts immediately after they are typed, whereas theorems become facts only when they are proved. Each fact can have two user-defined identifiers: a *title* and a *number*, e.g. :

Axiom $[p, q :$ boolean$]$, title “Symmetry of $\equiv$”, number “3.2”

States: $p \equiv q \equiv q \equiv p$

A proof is usually a list of steps. Each step consists of a *connective* operator and a *hint*. The hint is an indication of the fact (or the predefined property) that is used for performing the step. There are three types of proofs:

- *to fact*: Starting from the expression to be proved, it is transformed into a fact. An example of such a proof is the following:

Theorem $[p, q :$ boolean], title “Absorption”, number “3.43a”

Prove: $p \wedge (p \vee q) \equiv p$

Proof by transforming to a fact:

$p \wedge (p \vee q) \equiv p$

$= \langle$ Golden rule [3.35], with $q := p \vee q$ $\rangle$

$p \vee q \equiv p \vee p \vee q$

$= \langle$ Idempotency of $\vee$ [3.26] $\rangle$

$p \vee q \equiv p \vee q$

$\Box$ $\langle$ Reflexivity of $\equiv$ [3.5], with $p := p \vee q$ $\rangle$

The symbol “$\Box$” in the last line means that the proof has been completed. Note that a hint must be given after “$\Box$”, indicating to which fact the initial expression has been transformed.

- *from fact*: Starting from a fact, the fact is transformed into the expression to be proved. The same example, expressed as a proof of this type is:

Theorem $[p, q :$ boolean], title “Absorption”, number “3.43a”

Prove: $p \wedge (p \vee q) \equiv p$

Proof by starting from a fact:

$\langle$ Reflexivity of $\equiv$ [3.5], with $p := p \vee q$ $\rangle$

$p \vee q \equiv p \vee q$

$= \langle$ Idempotency of $\vee$ [3.26] $\rangle$

$p \vee q \equiv p \vee p \vee q$

$=$ ⟨ Golden rule [3.35], with $q := p \vee q$ ⟩

$p \wedge (p \vee q) \equiv p$

$\square$

Note that now there is no need for a hint after the "$\square$" symbol but before the first expression. This hint indicates which fact we are transforming.

- *by transformation*: Starting from some part of the expression to be proved, that part is transformed to the remaining part. The same example, expressed as a proof of this type is:

  Theorem $[p, q :$ boolean$]$, title "Absorption", number "3.43a"

  Prove: $p \wedge (p \vee q) \equiv p$

  Proof by transformation:

  $p \wedge (p \vee q)$

  $=$ ⟨ Golden rule [3.35], with $q := p \vee q$ ⟩

  $p \equiv p \vee q \equiv p \vee p \vee q$

  $=$ ⟨ Idempotency of $\vee$ [3.26] ⟩

  $p \equiv p \vee q \equiv p \vee q$

  $=$ ⟨ Symmetry of $\equiv$ [3.2], with $q := p \vee q$ ⟩

  p

  $\square$

  Note that there is no need for a hint after the "$\square$" symbol. The connective operator now becomes part of the expression that we are proving.

To fact proofs have the characteristic that, at each step, the whole proof depends on proving the current expression. Because of this characteristic, such proofs allow arbitrary dependencies that the user can specify by using rules. For instance, consider the following proof:

Theorem $[p, q : \text{boolean}]$

Prove: $p \equiv q$

Proof by mutual implication:

Prove: $p \Rightarrow q$

...

□

Prove: $q \Rightarrow p$

...

□

In this case, the proof of "$p \equiv q$" depends on the two proofs "$p \Rightarrow q$" and "$q \Rightarrow p$". These two proofs will be called *auxiliary* proofs.

We now define the notion of *scope*. A scope is a part of a proof *to a fact*, containing additional assumptions and auxiliary proofs. No auxiliary proof can span across scopes. Scopes can be nested, that is, they can include other scopes. *Assumptions* are facts that are considered valid without evidence. However, an assumption can have no free variables: all variables in an assumption are considered fixed and cannot be substituted by anything. Assumptions are only visible within the scope in which they are defined, or scopes that are nested in this. They can only appear in the beginning of a scope, before any auxiliary proofs. An example of a proof containing two nested scopes is the following. Scopes are shown by indentation.

```
Theorem [$p, q$ : boolean], number "4.1"
Prove: $p \Rightarrow (q \Rightarrow p)$

  Assume: $p$ (A1)
  Prove: $q \Rightarrow p$

    Assume: $q$ (A2)
    Prove: $p$

    Proof by transforming to a fact:
      $p$
    $\Box$ ⟨ Assumption A1 ⟩
```

2.1.2 Types and expressions

Types are generally sets of values. In our proof editor, though, types are just names. The set of values that inhabit a type cannot be defined. Nevertheless, all types are considered non-empty. All types are simple; our proof editor does not allow types to be built from other types.

The simplest expressions consist of single variables or constants. Variables and constants must be defined before they can be used. The definition of a variable or constant assigns to it a name and a type (however, to simplify the type system, we assign a user-defined default type to untyped variables).

There are three kinds of operators:

- *prefix* operators, i.e. unary operators of the form $\diamond : T \rightarrow T_r$. A prefix operator can be applied to operand x by writing "$\diamond x$".
- *postfix* operators, i.e. unary operators of the form $\diamond : T \rightarrow T_r$. A postfix operator can be applied to operand x by writing "$x \diamond$".
- *infix* operators, i.e. binary operators of the form $\diamond : T_1 \times T_2 \rightarrow T_r$. An infix operator can be applied to operands x, y by writing "$x \diamond y$".

The user can specify operator precedence and other properties. Parentheses can be used in order to enforce the order of operations that the user wants. In case of ambiguity in the application of operators, the proof editor must report an error (e.g. "$x \diamond_1 y \diamond_2 z$" is ambiguous if it is not known that one of the two operators has greater precedence than the other or that $\diamond_1$ and $\diamond_2$ are mutually associative).

Operators can be declared *associative* (see §2.2.5). This allows expressions of the form "$x \diamond y \diamond z$". Moreover, *mutual* associativity allows expressions of the form "$x \diamond_1 y \diamond_2 z$". In general, consider an expression of the form:

$$E_1 \diamond_1 E_2 \diamond_2 \ldots \diamond_{n-1} E_n$$

where $n \geq 2$ and operator precedence does not resolve the ambiguity. This expression is not ambiguous if:

```
for all $1 \leq i \leq j < n$:
    if $\diamond_i = \diamond_j$ , $\diamond_i$ is associative,
    if $\diamond_i \neq \diamond_j$ , $\diamond_i$ and $\diamond_j$ are mutually associative.
```

General functions can be defined, by giving the types of the operands and the result type. A function can in general be of the form $f : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$, where $n \geq 1$, and can be applied to its operands $x_1, x_2, \ldots, x_n$ by writing "$f(x_1, x_2, \ldots, x_n)$". Nevertheless, for our proof editor functions are just names.

Quantifications are expressions of the form $(\diamond\ \bar{x} \mid R : E)$, where $\diamond$ is an infix operator[1], $\bar{x}$ is a list of dummy variables, R is an optional boolean expression (the *range*; if it is omitted, it is considered to be the default *true* value) and E is an expression (the *body*). The scope of a dummy is the range and body.

2.1.3 Textual substitution

Textual substitution is the most important operation in our proof editor. The notation that we will use is "$E\ [x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$", where $n \geq 1$, E is the expression in which the substitution will take place, $x_1, x_2, \ldots, x_n$ are the *distinct* variables[2] that will be *simultaneously* substituted and $e_1, e_2, \ldots, e_n$ are the expressions that will replace them.

We first define the predicate *occurs*("$\bar{x}$", "$\bar{e}$"), where $\bar{x}$ is a list of variables and $\bar{e}$ is a list of expressions. It is *true* if any of the variables in $\bar{x}$ occurs free in any of the expressions in $\bar{e}$ (the occurrence of a variable in an expression is *free* if it is not in the scope of any quantification that has this variable as a dummy). The formal definition of textual substitution is given in Fig.2.1.

[1]The infix operator used in a quantification must be associative and symmetric. It must also have an identity.

[2]We allow only substitution of variables.

$$v\ [\bar{x} := \bar{e}] \ = \ \begin{cases} e_i & \text{, if variable } v \text{ is same as } x_i \\ v & \text{, otherwise} \end{cases}$$

$$c\ [\bar{x} := \bar{e}] \ = \ c$$

$$(\diamond E)\ [\bar{x} := \bar{e}] \ = \ \diamond(E\ [\bar{x} := \bar{e}])$$

$$(E\diamond)\ [\bar{x} := \bar{e}] \ = \ (E\ [\bar{x} := \bar{e}])\diamond$$

$$(E_1 \diamond_1 \ldots \diamond_{m-1} E_m)\ [\bar{x} := \bar{e}] \ = \ (E_1\ [\bar{x} := \bar{e}]) \diamond_1 \ldots \diamond_{m-1} (E_m\ [\bar{x} := \bar{e}])$$

$$f(E_1, E_2, \ldots, E_m)\ [\bar{x} := \bar{e}] \ = \ f(E_1\ [\bar{x} := \bar{e}], E_2\ [\bar{x} := \bar{e}], \ldots, E_m\ [\bar{x} := \bar{e}])$$

$$(E)\ [\bar{x} := \bar{e}] \ = \ (E\ [\bar{x} := \bar{e}])$$

Provided not $occurs(``\bar{y}", ``\bar{x}, \bar{e}")$:

$$(\diamond\ \bar{y} \mid R : E)\ [\bar{x} := \bar{e}] \ = \ (\diamond\ \bar{y} \mid R\ [\bar{x} := \bar{e}] : E\ [\bar{x} := \bar{e}])$$

Memorandum:

$\bar{x}$	:	the list of variables $x_1, x_2, \ldots, x_n$.
$\bar{e}$	:	the list of expressions $e_1, e_2, \ldots, e_n$.
v	:	a variable.
c	:	a constant.
$\diamond, \diamond_i$	:	operators.
f	:	a function.
$\bar{y}$	:	a list of dummy variables.
E, E_i, R	:	expressions.

Figure 2.1: Textual substitution.

2.1.4 Interface

Regardless of the appearance of the interface (windows, menus, etc.) some things are bound to be needed. We will make an effort not to put any unnecessary restrictions on the interface. What will certainly be needed is:

- The current *expression*. This is the whole expression that is being edited. When a proof is being edited, the current expression is the last expression appearing in the proof.
- The current *subexpression*. This is the subexpression of the current expression that has been selected for transformation. If nothing has been selected, the whole current expression is taken.
- The current *fact*. This is the fact that will be applied. The user can specify the appropriate variable substitutions, if necessary.
- The current *rule*. This is the user-defined rule that will be applied.

2.2 Predefined items

The following paragraphs present the predefined items of the proof editor.

2.2.1 Types

There are four predefined types:

- `boolean`: the type of all facts.

- `ANY`: a wild card type. Useful in definitions of axioms and theorems that contain equality and/or textual substitution, or for dummy variables in substitutions.
- `VARIABLE`: a type for variables of any type. Useful in definitions of axioms and theorems that contain textual substitution.
- `LIST`: a type for lists of expressions. Useful in definitions of axioms and theorems that contain textual substitution, as well as in quantifiers.

The first is a normal type, whereas the other three are only used for special purposes.

2.2.2 Operators

An equality operator is predefined for every type `T`:

```
define operator = infix : T x T -> boolean
```

The precedence of all equality operators is the same. All equality operators have the following properties (see also §2.2.5):

```
rule symmetric(=)

rule connective(=)

rule transitive(=, =, =)

rule conjunctional(=)

rule prove(=)

rule leibniz(=, =)
```

2.2.3 Accessors

The following accessors are predefined (see also §2.2.5). They can be used only when defining rules (see also §2.3.9).

- `TYPEOP(op)`: the type of the operand of prefix or postfix operator `op` (a type name).
- `TYPEOP1(op)` and `TYPEOP2(op)`: the types of the two operands (left and right respectively) of infix operator `op` (a type name).
- `TYPERES(op)`: the result type of operator `op` (a type name).
- `TYPEEXP(exp)`: the type of expression `exp` (a type name).
- `LIDENT(op)`: the left identity of infix operator `op`, if it exists (a constant).
- `RIDENT(op)`: the right identity of infix operator `op`, if it exists (a constant).
- `IDENTITY(op)`: the identity of infix operator `op`, if it exists (a constant).
- `LZERO(op)`: the left zero of infix operator `op`, if it exists (a constant).
- `RZERO(op)`: the right zero of infix operator `op`, if it exists (a constant).
- `ZERO(op)`: the zero of infix operator `op`, if it exists (a constant).
- `DUAL(op)`: the operator that has been declared dual to operator `op` by a rule. If operator `op` is symmetric, then it returns `op`.
- `DEF_TRUE`: the boolean constant that has been declared to be the default *true* value.

- `DEF_FALSE`: the boolean constant that has been declared to be the default *false* value.
- `DEF_AND`: the operator that has been declared to be the default conjunction operator.
- `DEF_OR`: the operator that has been declared to be the default disjunction operator.
- `QUANTOPER(op)`: the name that will be used when operator `op` is used in a quantification. If no `quantifier` rule has been given, it returns `op`.
- `CURR_EXPR`: the current expression.
- `CURR_FACT`: the current fact (if a hint is specified by the user, after all substitutions have been made).
- `CURR_SUBEXP`: the current subexpression expression.

2.2.4 Predicates

The following predicates are predefined. They can be used only in the definition of rules (see also §2.2.5 and §2.3.9).

- `IS_CONSTANT(exp)`: expression `exp` is the name of a constant.
- `IS_SYMMETRIC(op)`: operator `op` has been declared symmetric.
- `IS_ASSOCIATIVE(op)`: operator `op` has been declared associative.
- `IS_IDEMPOTENT(op)`: operator `op` has been declared idempotent.

- `HAS_LIDENT(op)`: operator `op` has been declared to have a left identity.
- `HAS_RIDENT(op)`: operator `op` has been declared to have a right identity.
- `HAS_IDENTITY(op)`: operator `op` has been declared to have both a left and a right identity and these are equal.
- `HAS_LZERO(op)`: operator `op` has been declared to have a left zero.
- `HAS_RZERO(op)`: operator `op` has been declared to have a right zero.
- `HAS_ZERO(op)`: operator `op` has been declared to have both a left and a right zero and these are equal.
- `IS_CONNECTIVE(op)`: operator `op` has been declared connective.
- `HAS_DUAL(op)`: operator `op` has been declared dual to another operator, or `op` is symmetric.
- `IS_DUAL(op1, op2)`: operators `op1` and `op2` have been declared dual.
- `IS_MUTUAL_ASSOC(op1, op2)`: operators `op1` and `op2` have been declared mutually associative.
- `IS_PROVE_FF(op)`: operator `op` has been declared `proveFF`.
- `IS_PROVE_TF(op)`: operator `op` has been declared `proveTF`.
- `IS_PROVE(op)`: operator `op` has been declared `prove`.
- `IS_RELATED(op1, op2)`: operators `op1` and `op2` have been declared related.

- `IS_CONJUNCTIONAL(op)`: operator `op` has been declared conjunctional.
- `IS_EQUIVALENT(op1, op2)`: operators `op1` and `op2` have been declared equivalent.
- `ISDEF_TRUE`: a default *true* value has been defined.
- `ISDEF_FALSE`: a default *false* value has been defined.
- `ISDEF_AND`: a default conjunction operator has been defined.
- `ISDEF_OR`: a default disjunction operator has been defined.

2.2.5 Rules for operator properties

Rules for operator properties are used internally by the editor. They are defined with the statement `rule`. Some of these properties can be applied to the current expression *on the fly*, that is without any intermediate steps and hints[3]. Some others affect the behavior of operators and the way in which expressions are matched.

Operator properties that are used in transformations are:

- `symmetric(op)` specifies that operator `op` is symmetric. This means that a subexpression of the form "`x op y`" can be replaced on the fly by "`y op x`". The following conditions must be satisfied:

  ```
  TYPEOP1(op) = TYPEOP2(op)
  not HAS_DUAL(op)
  ```

[3]However, the user can specify that the intermediate steps should not be omitted. In that case, a descriptive hint will be given and the equality operator will be used.

- `associative(op)` specifies that operator `op` is associative. This means that there is no ambiguity in expressions of the form "`x op y op z`" and no parentheses are needed. Such an expression has more than one outermost operator. This property can be used to transform on the fly a subexpression of the form "`(x op y) op z`" to "`x op (y op z)`" and vice versa. The following condition must be satisfied:

  ```
  TYPEOP1(op) = TYPEOP2(op) = TYPERES(op)
  ```

- `idempotent(op)` specifies that operator `op` is idempotent. This means that a subexpression of the form "`x op x`" can be replaced on the fly by "`x`" and vice versa. The following conditions must be satisfied:

  ```
  TYPEOP1(op) = TYPEOP2(op) = TYPERES(op)
  ```

- `dual(op1, op2)` specifies that operators `op1` and `op2` are *dual*. This means that a subexpression of the form "`x op1 y`" can be replaced on the fly by "`y op2 x`" and vice versa. The following conditions must be satisfied:

  ```
  not IS_SYMMETRIC(op1)
  not IS_SYMMETRIC(op2)
  ```

- `mutualAssoc(op1, op2)` specifies that operators `op1` and `op2` are *mutually associative*. This means that there is no ambiguity in expressions of the form "`x op1 y op2 z`" and no parentheses are needed. Such an expression has more than one outermost operator. This property can be used to transform on the fly a subexpression of the form "`(x op1 y) op2 z`" to "`x op1 (y op2 z)`" and vice versa. The following condition must be satisfied:

```
TYPEOP2(op1) = TYPEOP1(op2) = TYPERES(op1) = TYPERES(op2)
```

- `leftIdent(op, c)` specifies that `c` is the left identity of operator `op`. This means that a subexpression of the form "`c op x`" can be replaced on the fly by "`x`" and vice versa. The following conditions must be satisfied:

  ```
  IS_CONSTANT(c)
  TYPEEXP(c) = TYPEOP1(op) = TYPEOP2(op) = TYPERES(op)
  ```

- `rightIdent(op, c)` specifies that `c` is the right identity of operator `op`. This means that a subexpression of the form "`x op c`" can be replaced on the fly by "`x`" and vice versa. The same conditions as for `leftIdent` must be satisfied.

- `identity(op, c)` is the same as giving both:

  ```
  leftIdent(op, c)
  rightIdent(op, c)
  ```

- `leftZero(op, c)` specifies that `c` is the left zero of operator `op`. This means that a subexpression of the form "`c op x`" can be replaced on the fly by "`c`" and vice versa. The same conditions as for `leftIdent` must be satisfied.

- `rightZero(op, c)` specifies that `c` is the right zero of operator `op`. This means that a subexpression of the form "`x op c`" can be replaced on the fly by "`c`" and vice versa. The same conditions as for `leftIdent` must be satisfied.

- `zero(op, c)` is the same as giving both:

  ```
  leftZero(op, c)

  rightZero(op, c)
  ```

The following properties affect the behavior of operators:

- `connective(op)` specifies that operator `op` is a *connective* operator. This means that it can be used to connect expressions along the steps of a proof. The following conditions must be satisfied:

  ```
  TYPEOP1(op) = TYPEOP2(op)

  TYPERES(op) = boolean
  ```

- `transitive(op1, op2, op3)` specifies that the sequence of steps:

  ```
      exp1

  op1

      exp2

  op2

      exp3
  ```

  is equivalent to a step:

  ```
      exp1

  op3

      exp3
  ```

(omitting the hints). The following conditions must be satisfied:

```
IS_CONNECTIVE(op1)
IS_CONNECTIVE(op2)
IS_CONNECTIVE(op3)
```

- `proveFF(op)` specifies that if we have a sequence of steps equivalent to:

  ```
      exp1
  op
      exp2
  ```

  and `exp1` is a fact, then by ending the proof there `exp2` becomes a fact. The following conditions must be satisfied:

  ```
  IS_CONNECTIVE(op)
  TYPEOP1(op) = TYPEOP2(op) = boolean
  ```

- `proveTF(op)` specifies that if we have a sequence of steps equivalent to:

  ```
      exp1
  op
      exp2
  ```

  and `exp2` is a fact, then by ending the proof there `exp1` becomes a fact. The following conditions must be satisfied:

  ```
  IS_CONNECTIVE(op)
  TYPEOP1(op) = TYPEOP2(op) = boolean
  ```

- `prove(op)` is the same as giving both `proveFF` and `proveTF`. The following conditions must be satisfied:

  ```
  IS_CONNECTIVE(op)
  TYPEOP1(op) = TYPEOP2(op) = boolean
  ```

- `related(op1, op2)` specifies that if a sequence of steps equivalent to:

  ```
      exp1
  op1
      exp2
  ```

  is a valid proof for the expression "`exp1 op2 exp2`". Every operator is considered to be related to itself, therefore no rules of the form `related(op, op)` need to be given. The following conditions must be satisfied:

  ```
  IS_CONNECTIVE(op1)
  TYPEOP1(op1) = TYPEOP2(op1) = TYPEOP1(op2) = TYPEOP2(op2)
  ```

- `conjunctional(op)` specifies that a subexpression of the form:

  ```
  x1 op x2 op x3 op ... xn-1 op xn
  ```

  can be replaced on the fly by:

  ```
  (x1 op x2) DEF_AND (x2 op x3) DEF_AND ... DEF_AND (xn-1 op xn)
  ```

  and vice versa. Before this can be applied, though, a default conjunction operator must be defined. The following conditions must be satisfied:

```
TYPEOP1(op) = TYPEOP2(op)

TYPERES(op) = boolean
```

- `equivalent(op1, op2)` specifies that operators `op1` and `op2` are equivalent. This means that a subexpression of the form "`x op1 y`" can be replaced on the fly by "`x op2 y`" and vice versa[4]. The following conditions must be satisfied:

  ```
  TYPEOP1(op1) = TYPEOP1(op2)

  TYPEOP2(op1) = TYPEOP2(op2)

  TYPERES(op1) = TYPERES(op2)
  ```

  It should be noted that operator equivalence is not transitive, i.e. if it has been defined that "`equivalent(op1, op2)`" and "`equivalent(op2, op3)`", this does not mean that "`equivalent(op2, op3)`".

- `quantifier(new-op, op)` specifies that when operator `op` is used in a quantification, the new operator `new-op` should be used instead. Note that `new-op` is only a second name for `op`. However, it can be used only as a quantifier. The following conditions must be satisfied:

  ```
  IS_ASSOCIATIVE(op)

  IS_SYMMETRIC(op)

  HAS_IDENTITY(op)
  ```

Finally, the following rules can be used to specify the default *true* and *false* values and the default conjunction and disjunction operators.

[4]Provided that the replacement does not create any syntactic or semantic errors.

- `defTrue(c)` defines the default *true* value to be `c`. The following conditions must be satisfied:

  ```
  not ISDEF_TRUE

  IS_CONSTANT(c)

  TYPEEXP(c) = boolean
  ```

- `defFalse(c)` defines the default *false* value to be `c`. The following conditions must be satisfied:

  ```
  not ISDEF_FALSE

  IS_CONSTANT(c)

  TYPEEXP(c) = boolean
  ```

- `defAnd(op)` defines the default conjunction operator to be `op`. The following conditions must be satisfied:

  ```
  not ISDEF_AND

  TYPEOP1(op) = TYPEOP2(op) = TYPERES(op) = boolean

  IS_ASSOCIATIVE(op)
  ```

- `defOr(c)` defines the default disjunction operator to be `op`. The following conditions must be satisfied:

  ```
  not ISDEF_OR

  TYPEOP1(op) = TYPEOP2(op) = TYPERES(op) = boolean

  IS_ASSOCIATIVE(op)
  ```

2.2.6 Rules for transformation steps

The following rules are used internally by the editor in order to create transformation steps.

- `substitute(op1, op2)` specifies that if there is a fact of the form "`x op1 y`" and the current expression is "x", then a step can be made to "y", using operator `op2` and the fact as a hint.

  If `IS_SYMMETRIC(op1)`, then the step can be made even if the fact is of the form "`y op1 x`".

  If `IS_DUAL(op1, op1')` and `IS_DUAL(op2, op2')` and the fact is of the form "`y op1' x`", then the step can be made using operator `op2'` and the fact as a hint.

  The following conditions must be satisfied:

  ```
  TYPEOP1(op1) = TYPEOP2(op1)
  IS_CONNECTIVE(op2)
  ```

- `leibniz(op1, op2)` specifies that if there is a fact of the form "`x op1 y`" and the current expression contains "x" as a subexpression, then a step can be made by replacing "x" by "y" in the current expression, using operator `op2` and the fact as a hint. This property is an extension of the previous property (`substitute`).

  If `IS_SYMMETRIC(op1)`, then the step can be made even if the fact is of the form "`y op1 x`".

If `IS_DUAL(op1, op1')` and `IS_DUAL(op2, op2')` and the fact is of the form "`y op1' x`", then the step can be made using operator `op2'` and the fact as a hint.

The following conditions must be satisfied:

```
TYPEOP1(op1) = TYPEOP2(op1)
IS_CONNECTIVE(op2)
```

2.2.7 Proof techniques

Three proof techniques are predefined:

- `PROVE_FF`: Starting with a fact, transform it to the expression that needs to be proved. If the sequence of steps is equivalent to:

```
    exp1
op
    exp2
```

  and `exp1` is a fact, then `exp2` becomes a fact. The following condition has to be satisfied:

```
IS_PROVE_FF(op)
```

- `PROVE_TF`: Starting with the expression that needs to be proved, transform it to a fact. If the sequence of steps is equivalent to:

```
    exp1
```

```
op
    exp2
```

and `exp2` is a fact, then `exp1` becomes a fact. The following condition has to be satisfied:

```
IS_PROVE_TF(op)
```

- `PROVE_TR`: Starting with an expression, transform it to another expression. If the sequence of steps is equivalent to:

  ```
      exp1
  op1
      exp2
  ```

  and the expression that needs to be proved is of the form "`exp1 op2 exp2`", then it is considered proved if the following condition is also satisfied:

  ```
  IS_RELATED(op1, op2)
  ```

2.2.8 Quantifier properties

The following quantifier properties have been predefined. They can be applied on the fly in the same way as operator properties defined by rules. The user can specify whether the intermediate steps will be omitted. If intermediate steps are not omitted, a descriptive hint will be given and the equality operator will be used.

- *Missing range*:

```
(op x |: E) = (op x | DEF_TRUE : E)
```

- *Empty range*:

```
(op x | DEF_FALSE : E) = IDENTITY(op)
```

- *One-point rule*: Provided not *occurs*(“x”, “E”):

```
(op x | x = E : F) = F[x:=E]
```

- *Distributivity*:

```
(op x | R : P) op (op x | R : Q) = (op x | R : P op Q)
```

- *Range split*:

```
(op x | R DEF_OR S : P) op (op x | R DEF_AND S : P) =
   (op x | R : P) op (op x | S : P)
```

- *Range split*: Provided (`R DEF_AND S = DEF_FALSE`) or `IS_IDEMPOTENT(op)`:

```
(op x | R DEF_OR S : P) = (op x | R : P) op (op x | S : P)
```

- *Interchange of dummies*: Provided not *occurs*("y", "R") and not *occurs*("x", "Q"):

```
(op x | R : (op y | Q : P)) = (op y | Q : (op x | R : P))
```

- *Nesting*: Provided not *occurs*("y", "R"):

```
(op x, y | R DEF_AND Q : P) = (op x | R : (op y | Q : P))
```

- *Dummy renaming*: Provided not *occurs*("y", "R, P"):

```
(op x | R : P) = (op y | R[x:=y] : P[x:=y])
```

- *Dummy reordering*:

```
(op x, y | R : P) = (op y, x | R : P)
```

2.3 The proof editor's internal language

Although the syntax of this language may change, we think it is necessary to define its basic characteristics and semantics[5]. Each file that is processed by the proof editor is of the form:

<file> ::= (*<statement>*)*

The following paragraphs describe what statements are.

[5]This is the internal language; users need not write in it or read it.

2.3.1 Using existing files

The `include` statement is used in order to include the contents of an existing file in the current file. Its syntax is:

> *<statement>* ::= “`include`” *<string>*

It should be noted that the included file cannot be changed while editing the file from which it was included. For example, the proof of a theorem that was defined in the included file cannot be edited.

2.3.2 Definition of types

Types are defined by statements of the form:

> *<statement>* ::= “`define`” “`type`” *<id:type>* (“`,`” *<id:type>*)*

When used in some declarations, a type is defined as:

> *<type>* ::= *<id:type>* | “`ANY`” | “`VARIABLE`” | “`LIST`”

2.3.3 Definition of constants

Constants are defined by statements of the form:

> *<statement>* ::= “`define`” “`constant`” *<id:constant>*
> (“`,`” *<id:constant>*)* “`:`” *<id:type>*

2.3.4 Definition of operators

Operators are defined by statements of the form:

<statement> ::= "`define`" "`operator`" *<id:operator>*
("`prefix`" | "`postfix`") "`:`" *<id:type>* "`->`" *<id:type>*
| "`define`" "`operator`" *<id:operator>* "`infix`" "`:`"
<id:type> "`x`" *<id:type>* "`->`" *<id:type>*

The precedence of operators can be defined by statements of the form:

<statement> ::= "`define`" "`precedence`" "`PREC`" "`(`" *<id:operator>* "`)`"
(("`<`" | "`=`") "`PREC`" "`(`" *<id:operator>* "`)`")$^+$

Several such statements for several operators create a *partial* order of precedence[6]. The definition "`PREC(op1) < PREC(op2)`" means that `op2` binds stronger than `op1`, whereas the definition "`PREC(op1) = PREC(op2)`" means that `op1` has the same precedence as `op2`.

2.3.5 Definition of functions

Functions are defined by statements of the form:

<statement> ::= "`define`" "`function`" *<id:function>* "`:`"
<id:type> ("`x`" *<id:type>*)* "`->`" *<id:type>*

2.3.6 Expressions

Expressions are the most basic part of the internal language. They are defined by the following rules:

<expression> ::= *<term>* ["`[`" *<seq-subst>* "`]`"]

[6]A total order is imposed by the proof editor's interface.

| *<id:operator>* *<expression>*

| *<expression>* *<id:operator>*

| *<expression>* *<id:operator>* *<expression>*

<term> ::= *<id:constant>*

| *<id:variable>*

| *<id:function>* "(" *<expr-list>* ")"

| "(" *<expression>* ")"

| "(" *<id:operator>* *<dummy-decl>* "`|`"

[*<expression>*] ":" *<expression>* ")"

<seq-subst> ::= *<var-list>* "`:=`" *<expr-list>*

<var-list> ::= *<id:variable>* ("," *<id:variable>*)*

<expr-list> ::= *<expression>* ("," *<expression>*)*

<dummy-decl> ::= *<part-dummy-decl>* (";" *<part-dummy-decl>*)*

<part-dummy-decl> ::= *<var-list>* ":" (*<id:type>* | "`LIST`")

2.3.7 Definition of axioms

Axioms are defined by statements of the following form:

<statement> ::= "`axiom`" ["`[`" *<type-decl>* "`]`"] *<fact-name>*

(*<fact-condition>*)* "`states`" "{" *<expression>* "}"

where:

<type-decl> ::= *<part-type-decl>* (";" *<part-type-decl>*)*

<part-type-decl> ::= *<var-list>* "`:`" *<type>*

<fact-name> ::= ["`title`" *<string>*] ["`number`" *<string>*]

<fact-condition> ::= "`condition`" ["`not`"]
"`occurs`" "`(`" *<var-list>* "`;`" *<expr-list>* "`)`"

2.3.8 Definition of theorems

Theorems are defined by statements of the form:

<statement> ::= "`theorem`" ["`[`" *<type-decl>* "`]`"] *<fact-name>*
(*<fact-condition>*)* *<proof>*

<proof> ::= "`prove`" "{" *<expression>* "}" *<proof-tail>*

In order for a theorem to be considered a fact, the proof that follows it must be completed and valid. These properties, though, cannot easily be specified syntactically and are ignored for the time being. A proof is defined as follows:

<proof-tail> ::= ϵ
| "`fromFact`" *<hint>* "{" *<expression>* "}" *<seq-of-steps>*
["`qed`"]
| "`toFact`" *<seq-of-steps>*
["`qed`" *<hint>* | *<depends-clause>*]
| "`transform`" "{" *<expression>* "}" *<seq-of-steps>*
["`qed`"]

<seq-of-steps> ::= (*<step>* "{" *<expression>* "}")*

<step> ::= "`step`" *<id:operator>* *<hint>*

<hint> ::= "`hint`" (*<fact-hint>* | *<property-hint>*)

<fact-hint> ::= ["`substFact`"] *<string>* ["`[`" *<seq-subst>* "`]`"]

<property-hint> ::= "`insparen`"
| "`rmvparen`"
| "`perfsub`"
| "`symmetric`" "`(`" *<id:operator>* "`)`"
| "`dual`" "`(`" *<id:operator>* "`,`" *<id:operator>* "`)`"
| "`conjunctional`" "`(`" *<id:operator>* "`)`"
| "`equivalent`" "`(`" *<id:operator>* "`,`" *<id:operator>* "`)`"

<depends-clause> ::= "`depends`" *<proof-scope>* "`enddep`"

<proof-scope> ::= (*<assumption>*)* (*<proof>*)$^+$

<assumption> ::= "`assume`" "{" *<expression>* "}" *<fact-name>*

2.3.9 Definition of rules

Rules are divided in four categories:

- Predefined rules for transforming the current subexpression on the fly.
- Predefined rules for transforming the current subexpression by using facts.
- Predefined rules for defining the default *true* and *false* values and also the default conjunction and disjunction operators.

- User-defined rules.

Rules can be defined with statements of the form:

```
<statement> ::= "rule" <pred-oper-property>
              | "rule" <pred-transf-rule>
              | "rule" <pred-default-val>
              | "rule" [ "title" <string> ] ( <rule-statement> )*

<pred-oper-property> ::= "symmetric" "(" <id:operator> ")"
                       | "associative" "(" <id:operator> ")"
                       | "idempotent" "(" <id:operator> ")"
                       | "dual" "(" <id:operator> "," <id:operator> ")"
                       | "mutualAssoc" "(" <id:operator> ","
                           <id:operator> ")"
                       | "leftIdent" "(" <id:operator> ","
                           <id:constant> ")"
                       | "rightIdent" "(" <id:operator> ","
                           <id:constant> ")"
                       | "identity" "(" <id:operator> ","
                           <id:constant> ")"
                       | "leftZero" "(" <id:operator> ","
                           <id:constant> ")"
                       | "rightZero" "(" <id:operator> ","
                           <id:constant> ")"
```

```
                      | "zero" "(" <id:operator> "," <id:constant> ")"
                      | "connective" "(" <id:operator> ")"
                      | "transitive" "(" <id:operator> ","
                           <id:operator> "," <id:operator> ")"
                      | "proveFF" "(" <id:operator> ")"
                      | "proveTF" "(" <id:operator> ")"
                      | "prove" "(" <id:operator> ")"
                      | "related" "(" <id:operator> ","
                           <id:operator> ")"
                      | "conjunctional" "(" <id:operator> ")"
                      | "equivalent" "(" <id:operator> ","
                           <id:operator> ")"
                      | "quantifier" "(" <id:operator> ","
                           <id:operator> ")"

<pred-transf-rule> ::= "substitute" "(" <id:operator> ","
                          <id:operator> ")"
                     | "leibniz" "(" <id:operator> ","
                          <id:operator> ")"

<pred-default-val> ::= "defTrue" "(" <id:constant> ")"
                     | "defFalse" "(" <id:constant> ")"
                     | "defAnd" "(" <id:operator> ")"
                     | "defOr" "(" <id:operator> ")"
```

```
<rule-statement> ::= "fail" <string>
                   |  "scopeBegin" ( <scope-statement> )* "scopeEnd"
                   |  "if" <rule-condition> "then" ( <rule-statement> )*
                        ( "elseif" <rule-condition> "then"
                            ( <rule-statement> )* )*
                        [ "else" ( <rule-statement> )* ] "endif"

<scope-statement> ::= "assume" [ "array" "(" <id:variable> ")" ]
                          "{" <rule-expression> "}"
                    |  "prove" [ "array" "(" <id:variable> ")" ]
                          "{" <rule-expression> "}"

<rule-condition> ::= <rule-cond-term> ( "or" <rule-cond-term> )*

<rule-cond-term> ::= <rule-cond-factor> ( "and" <rule-cond-factor> )*

<rule-cond-factor> ::= "(" <rule-condition> ")"
                     |  "not" <rule-cond-factor>
                     |  "occurs" "(" <var-list> ";" <rule-expr-list> ")"
                     |  <rule-primary> "matches" <rule-primary>
                     |  "MODE" "is"
                           ( "PROVE_FF" | "PROVE_TF" | "PROVE_TR" )
                     |  <pred-predicate>

<rule-primary> ::= [ "[" <rule-type-decl> "]" ] "{" <rule-expression> "}"
                 |  [ "[" <id:variable> ":" "array" "of" ( <id:type>
```

| <*pred-access-type*>) "]"]

"EXP" "(" <*id:variable*> "," <*infix-oper*> ")"

<*infix-oper*> ::= <*id:operator*> | <*meta-infix-oper*>

<*rule-expr-list*> ::= <*rule-expression*> ("," <*rule-expression*>)*

<*rule-expression*> ::= <*rule-term*> ["[" <*rule-seq-subst*> "]"]

| <*id:operator*> <*rule-expression*>

| <*rule-expression*> <*id:operator*>

| <*rule-expression*> <*infix-oper*> <*rule-expression*>

<*rule-term*> ::= <*id:constant*>

| <*id:variable*>

| <*id:function*> "(" <*rule-expr-list*> ")"

| "(" <*rule-expression*> ")"

| "(" <*id:operator*> <*rule-dummy-decl*> "|"

[<*rule-expression*>] ":" <*rule-expression*> ")"

| <*pred-access-term*>

<*rule-seq-subst*> ::= <*var-list*> ":=" <*rule-expr-list*>

<*rule-dummy-decl*> ::= <*rule-part-dummy-decl*>

(";" <*rule-part-dummy-decl*>)*

<*rule-part-dummy-decl*> ::= <*var-list*> ":"

(<*id:type*> | "LIST" | <*pred-access-type*>)

<*rule-type-decl*> ::= <*part-rule-type-decl*> (";" <*part-rule-type-decl*>)*

```
<rule-part-type-decl> ::= <var-list> ":" ( <type> | <pred-access-type> )

<pred-predicate> ::= "IS_CONSTANT" "(" <expression> ")"
                   | "IS_SYMMETRIC" "(" <id:operator> ")"
                   | "IS_ASSOCIATIVE" "(" <id:operator> ")"
                   | "IS_IDEMPOTENT" "(" <id:operator> ")"
                   | "HAS_LIDENT" "(" <id:operator> ")"
                   | "HAS_RIDENT" "(" <id:operator> ")"
                   | "HAS_IDENTITY" "(" <id:operator> ")"
                   | "HAS_LZERO" "(" <id:operator> ")"
                   | "HAS_RZERO" "(" <id:operator> ")"
                   | "HAS_ZERO" "(" <id:operator> ")"
                   | "IS_CONNECTIVE" "(" <id:operator> ")"
                   | "HAS_DUAL" "(" <id:operator> ")"
                   | "IS_DUAL" "(" <id:operator> "," <id:operator> ")"
                   | "IS_MUTUAL_ASSOC" "(" <id:operator> ","
                       <id:operator> ")"
                   | "IS_PROVE_FF" "(" <id:operator> ")"
                   | "IS_PROVE_TF" "(" <id:operator> ")"
                   | "IS_PROVE" "(" <id:operator> ")"
                   | "IS_RELATED" "(" <id:operator> "," <id:operator> ")"
                   | "IS_CONJUNCTIONAL" "(" <id:operator> ")"
                   | "IS_EQUIVALENT" "(" <id:operator> ","
                  <id:operator> ")"
              |  "ISDEF_TRUE"
              |  "ISDEF_FALSE"
              |  "ISDEF_AND"
              |  "ISDEF_OR"

<pred-access-type> ::= "TYPEOP" "(" ( <id:operator> ) ")"
                    |  "TYPEOP1" "(" <id:operator> ")"
                    |  "TYPEOP2" "(" <id:operator> ")"
                    |  "TYPERES" "(" ( <id:operator> ) ")"
                    |  "TYPEEXP" "(" <expression> ")"

<meta-infix-oper> ::= "DUAL" "(" <id:operator> ")"
                   |  "CONJUNCT" "(" <id:operator> ")"
                   |  "QUANTOPER" "(" <id:operator> ")"
                   |  "DEF_AND" | "DEF_OR"

<pred-access-term> ::= "LIDENT" "(" <id:operator> ")"
                    |  "RIDENT" "(" <id:operator> ")"
                    |  "IDENTITY" "(" <id:operator> ")"
                    |  "LZERO" "(" <id:operator> ")"
                    |  "RZERO" "(" <id:operator> ")"
                    |  "ZERO" "(" <id:operator> ")"
                    |  "DEF_TRUE"
                    |  "DEF_FALSE"
| "CURR_EXPR"
| "CURR_FACT"
| "CURR_SUBEXP"
```

A small programming language allows the users to define their own rules. The semantics of this language is described in §2.4.4. We tried to make this language as simple as possible, so many features that may be needed have been left out. User-defined rules can be used in order to relax the style of proofs. Some features of our editor, such as scopes and auxiliary proofs, cannot be used but from user-defined rules.

2.4 More detailed specifications

In this section we give some more formal definitions of terms used in the previous sections and describe the user interface in more detail.

2.4.1 Subexpressions

We now define the notion of a subexpression. Let $\mathcal{S}[E]$ be the set of subexpressions of E. A formal definition of $\mathcal{S}[E]$ is given in Fig.2.2.

2.4.2 Structure of expressions

We now define the notion of an expression's *structure*. We have to define two auxiliary notions: *outermost operator* and *primary subexpression*. Let $\mathcal{O}[E]$ be the tuple[7]

[7]A *tuple* is an ordered set that can contain multiple occurrences of the same element. A tuple containing the elements $x_1, x_2, \ldots, x_n$ in this order is written as $\langle x_1, x_2, \ldots, x_n \rangle$. The empty tuple is written as $\langle\rangle$. If x is an element and t is a tuple, $x \triangleleft t$ is the tuple that results from prepending x to t.

E	$\mathcal{S}[E]$
v	$\{\, v \,\}$
c	$\{\, c \,\}$
$\diamond E$	$\{\, \diamond E \,\} \; \cup \; \mathcal{S}[E]$
$E \diamond$	$\{\, E \diamond \,\} \; \cup \; \mathcal{S}[E]$
$E_1 \diamond_1 E_2 \diamond_2 \ldots \diamond_{n-1} E_n$	$\bigcup_{1 \leq i < j \leq n} \{\, E_i \diamond_i \ldots \diamond_{j-1} E_j \,\} \; \cup \; \bigcup_{i=1}^{n} \; \mathcal{S}[E_i]$
$f(E_1, E_2, \ldots, E_n)$	$\{\, f(E_1, E_2, \ldots, E_n) \,\} \; \cup \; \mathcal{S}[E_1] \; \cup \; \mathcal{S}[E_2] \; \cup \; \ldots \; \cup \; \mathcal{S}[E_n]$
(E)	$\{\, (E) \,\} \; \cup \; \mathcal{S}[E]$
$(\diamond \; \bar{y} \mid R : E)$	$\{\, (\diamond \; \bar{y} \mid R : E) \,\} \; \cup \; \mathcal{S}[R] \; \cup \; \mathcal{S}[E]$
$E \; [\bar{x} := \bar{e}]$	$\{\, E \; [\bar{x} := \bar{e}] \,\} \; \cup \; \mathcal{S}[E] \; \cup \; \bigcup_{e_i \in \bar{e}} \; \mathcal{S}[E_i]$

E	$\mathcal{O}[E]$	$\mathcal{P}[E]$
v	$\langle\, \mathit{variable} \,\rangle$	$\langle\, v \,\rangle$
c	$\langle\, \mathit{constant} \,\rangle$	$\langle\, c \,\rangle$
$\diamond E$	$\langle\, \mathit{prefix}[\diamond] \,\rangle$	$\langle\, E \,\rangle$
$E \diamond$	$\langle\, \mathit{postfix}[\diamond] \,\rangle$	$\langle\, E \,\rangle$
$E_1 \diamond_1 E_2 \diamond_2 \ldots \diamond_{n-1} E_n$	$\langle\, \mathit{infix}[\diamond_1], \mathit{infix}[\diamond_2], \ldots, \mathit{infix}[\diamond_{n-1}] \,\rangle$	$\langle\, E_1, E_2, \ldots, E_n \,\rangle$
$f(E_1, E_2, \ldots, E_m)$	$\langle\, \mathit{function}[f] \,\rangle$	$\langle\, E_1, E_2, \ldots, E_n \,\rangle$
(E)	$\langle\, \mathit{parenthesis} \,\rangle$	$\langle\, E \,\rangle$
$(\diamond \; \bar{y} \mid R : E)$	$\langle\, \mathit{quantifier}[\diamond], \bar{y} \,\rangle$	$\langle\, R, E \,\rangle$
$E \; [\bar{x} := \bar{e}]$	$\langle\, \mathit{substitution}, \bar{x} \,\rangle$	$\langle\, \bar{e} \,\rangle$

<u>Memorandum:</u>

v	:	a variable.
c	:	a constant.
$\diamond$	:	an operator.
f	:	a function.
$\bar{x}, \bar{y}$	:	lists of dummy variables.
E, E_i, R	:	expressions.
$\bar{e}$	:	a list of expressions.

Figure 2.2: Subexpressions and structure.

of all outermost operators of expression E. Let $\mathcal{P}[E]$ be the tuple of all primary subexpressions of E. A definition of these two notions is given in Fig.2.2.

2.4.3 Expression matching

Expression matching is a very important operation of our proof editor. It is used internally in the transformation procedure and also in user-defined rules. However, we do not give a completely formal definition for expression matching, since this would be rather tedious.

Variables are generally divided in two categories:

- *Free* variables: variables that are declared before an expression, surrounded by brackets, e.g.

  ```
  [p, q : boolean]  p && (p => q) => q
  ```

  Such variables can be replaced by other expressions.

- *Fixed* variables: variables that are considered to be fixed and cannot be substituted by other expressions.

When a free variable becomes *bound* to an expression, every occurrence of this variable in the initial expression is substituted by this expression. A *binding* is a tuple of pairs of the form "(v, e)", where v is a free variable and e is an expression. If B is a binding and E is an expression, we define $E\ \{B\}$ to be the result of the substitution in E of all variables that are bound in B, in the order they appear in B. A recursive definition of $E\ \{B\}$ is given below:

$$\begin{aligned} E\ \{\langle\rangle\} &= E \\ E\ \{(v,e) \triangleleft B\} &= E\ [v := e]\ \{B\} \end{aligned}$$

When matching two expressions, the aim is to find a binding of the free variables that makes the expressions identical. With respect to binding B, variable v can be *free*, *bound* or *fixed*.

Bindings can be merged. The result $B_1@B_2$ of merging two bindings B_1 and B_2 is a binding that has the property that for all expressions E:

$$E\ \{B_1@B_2\} = E\ \{B_2\}\ \{B_1\}$$

Note that the order of the two arguments in $B_1@B_2$ cannot be reversed.

We now describe the matching algorithm. We define *match*$(E_1; B_1; E_2; B_2)$ to return a tuple of three elements $\langle\, f, B_1', B_2' \,\rangle$. The first element is a boolean. Its value is true, if and only if there are bindings B_1' and B_2' such that:

$$E_1\ \{B_1'@B_1\} = E_2\ \{B_2'@B_2\}$$

If f is true, the second and third elements of the result contain the two bindings respectively. If f is false, they both contain the empty tuple.

We start with a general rule about matching:

if *match*$(E_1; B_1; E_2; B_2) = \langle\, f, B_1', B_2' \,\rangle$

then *match*$(E_2; B_2; E_1; B_1) = \langle\, f, B_2', B_1' \,\rangle$.

Therefore, we do not need to handle symmetrical cases.

We now define the the matching for parenthesized expressions, variables and constants. In the following, the letter c denotes constants and the letter v denotes variables.

$$\mathit{match}((E_1); B_1; E_2; B_2) \;=\; \mathit{match}(E_1; B_1; E_2; B_2)$$

If v is free in B_1:

$$\mathit{match}(v; B_1; E_2; B_2) \;=\; \langle\, \mathit{true}, (v, E_2) \triangleleft B_1, B_2 \,\rangle$$

If v is bound in B_1:

$$\mathit{match}(v; B_1; E_2; B_2) \;=\; \mathit{match}(v\ \{B_1\}; B_1; E_2; B_2)$$

If v_1 and v_2 are fixed in B_1 and B_2 respectively, and $v_1 = v_2$:

$$\mathit{match}(v_1; B_1; v_2; B_2) \;=\; \langle\, \mathit{true}, B_1, B_2 \,\rangle$$

Matching of expressions containing operators, function applications and textual substitution is defined as:

$$\mathit{match}(\diamond E_1; B_1; \diamond E_2; B_2) \;=\; \mathit{match}(E_1; B_1; E_2; B_2)$$

$$\mathit{match}(E_1 \diamond; B_1; E_2 \diamond; B_2) \;=\; \mathit{match}(E_1; B_1; E_2; B_2)$$

```
To calculate: $\mathit{match}(E_1^1 \diamond E_1^2; B_1; E_2^1 \diamond E_2^2; B_2)$
    $\langle\, f, B_1', B_2' \,\rangle \leftarrow \mathit{match}(E_1^1; B_1; E_2^1; B_2)$
    if $f = \mathit{false}$ then
        return $\langle\, \mathit{false}, \langle\rangle, \langle\rangle \,\rangle$
    $\langle\, f, B_1'', B_2'' \,\rangle \leftarrow \mathit{match}(E_1^2; B_1'@B_1; E_2^2; B_2'@B_2)$
    if $f = \mathit{false}$ then
        return $\langle\, \mathit{false}, \langle\rangle, \langle\rangle \,\rangle$
    return $\langle\, true, B_1''@B_1', B_2''@B_2' \,\rangle$

To calculate: $match(f(E_1^1, E_1^2, \ldots, E_1^n); B_1; f(E_2^1, E_2^2, \ldots, E_2^n); B_2)$
    $B_1'' \leftarrow \langle\rangle$ , $B_2'' \leftarrow \langle\rangle$
    for $i = 1$ to $n$ do
        $\langle\, f, B_1', B_2' \,\rangle \leftarrow match(E_1^i; B_1; E_2^i; B_2)$
        if $f = false$ then
            return $\langle\, false, \langle\rangle, \langle\rangle \,\rangle$
        $B_1 \leftarrow B_1'@B_1$ , $B_2 \leftarrow B_2'@B_2$
        $B_1'' \leftarrow B_1'@B_1''$ , $B_2'' \leftarrow B_2'@B_2''$
    end do
    return $\langle\, true, B_1'', B_2'' \,\rangle$

To calculate: $match(E_1\ [\bar{x} := \bar{e}_1]; B_1; E_2\ [\bar{x} := \bar{e}_2]; B_2)$
    $\langle\, f, B_1', B_2' \,\rangle \leftarrow match(E_1; B_1; E_2; B_2)$
    if $f = false$ then
        return $\langle\, false, \langle\rangle, \langle\rangle \,\rangle$
    $B_1 \leftarrow B_1'@B_1$ , $B_2 \leftarrow B_2'@B_2$
    $B_1'' \leftarrow \langle\rangle$ , $B_2'' \leftarrow \langle\rangle$
    for $i = 1$ to $\mid \bar{e}_1 \mid$ do
        $\langle\, f, B_1', B_2' \,\rangle \leftarrow match(e_1^i; B_1; e_2^i; B_2)$
        if $f = false$ then
            return $\langle\, false, \langle\rangle, \langle\rangle \,\rangle$
        $B_1 \leftarrow B_1'@B_1$ , $B_2 \leftarrow B_2'@B_2$
    $B''_1 \leftarrow B'_1@B''_1$ , $B''_2 \leftarrow B'_2@B''_2$
  end do
  return $\langle\, true, B''_1, B''_2 \,\rangle$
```

Finally, matching quantifiers is defined as:

```
To calculate: $match((\diamond\ \bar{x}_1 \mid R_1 : E_1); B_1; (\diamond\ \bar{x}_2 \mid R_2 : E_2); B_2)$
  $B_d \leftarrow matchDummy(\bar{x}_1; \bar{x}_2)$
  $\langle\, f, B'_1, B'_2 \,\rangle \leftarrow match(R_1; B_d@B_1; R_2; B_d@B_2)$
  if $f = false$ then
    return $\langle\, false, \langle\rangle, \langle\rangle \,\rangle$
  $\langle\, f, B''_1, B''_2 \,\rangle \leftarrow match(E_1; B'_1@B_d@B_1; E_2; B'_1@B_d@B_2)$
  if $f = false$ then
    return $\langle\, false, \langle\rangle, \langle\rangle \,\rangle$
  return $\langle\, true, B''_1@B'_1, B''_2@B'_1 \,\rangle$
```

where *matchDummy*$(\bar{x}_1; \bar{x}_2)$ is a binding of dummy variables, such that the lists $\bar{x}_1$ and $\bar{x}_2$ become identical. Its definition is left out.

2.4.4 The language for rules

The simple programming language for user-defined rules supports six statements, so far. It will probably be extended in the future. It supports also a new type constructor, `array of`, and a new expression constructor, `EXP(a, op)`. These will be described together with the `if` statement.

A brief description of the semantics of rule statements is given below. The syntax of these statements is defined in §2.3.9, where the grammar is given. It is not repeated here. We explain only what is not clear from the grammar. The `if` statement is the most important and is presented first.

- `if`: What needs to be defined is the semantics of the conditions. There are four primary conditions for the `if` statement. The first is a predefined predicate. The other three are:

  - `MODE is PROVE_xxx`: This condition is true if the proof technique that is currently used is `PROVE_xxx`. If no proof technique has yet been declared (the proof has just started), then the condition is still true and the current proof technique is set to `PROOF_xxx`.

  - `x matches y`: This is the most important primary condition; it allows the user to access the structure of an expression. The expressions `x` and `y` can be either normal expressions, with free variables declared in brackets, or the special expression constructor `EXP(a, op)`, with `a` possibly a free variable declared in brackets and `op` an infix operator.

    In `EXP(a, op)`, `a` is an array of expressions. If the number of expressions in `a` is n and the expressions are $E_1, E_2, \ldots, E_n$, then the constructor `EXP(a, op)` is equivalent to the expression:

    $$E_1 \ op \ E_2 \ op \ \ldots \ op \ E_n$$

    Operator `op` has to be associative. If array `a` is free, when matching `EXP(a, op)` expressions $E_1, E_2, \ldots, E_n$ are considered free, but also their number is not determined. However, it must be $n \geq 2$.

When a condition of the form "`x matches y`" is given, the proof editor is trying to match the two expressions `x` and `y`, possibly by assigning values to free variables. If a matching is possible, the condition is true, otherwise it is false.

  - `occurs(v;e)`: The predicate that has been defined in §2.1.3, where `v` is a variable list and `e` is an expression list. Arrays can also be used as expression lists.

- `beginScope` and `endScope`: Defines a new scope. If the current proof technique is not `PROVE_TF` an error occurs. In this scope there can be the following *scope* statements:

  - `assume`: Make an assumption. Assumptions can be made only in the beginning of a new scope. If this is not the case, an error occurs. If an array is given, the statement is repeated for all elements of the array.

  - `prove`: Start an auxiliary proof. If the current proof technique is not `PROVE_TF` an error occurs. The current proof before the rule was applied will depend on the new one. If an array is given, the statement is repeated for all elements of the array.

- `fail`: Print a diagnostic message, explaining why this rule cannot be applied, and stop.

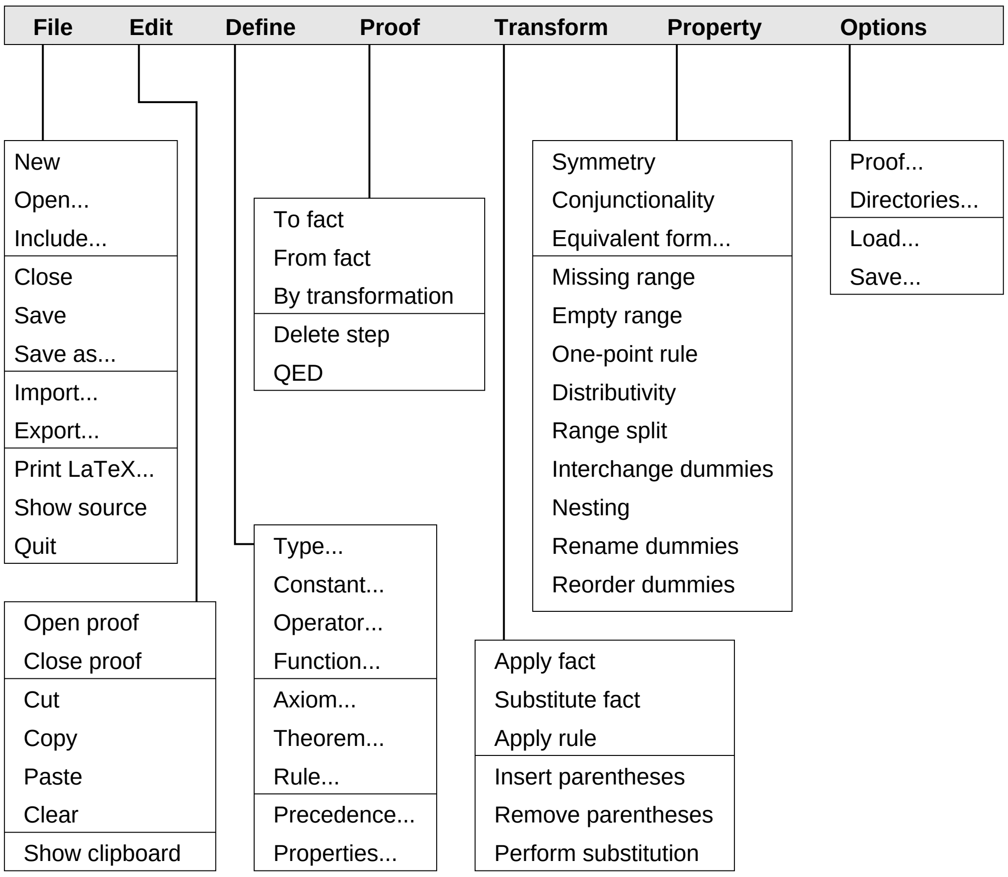


Figure 2.3: The main menu.

2.4.5 The interface

We now give more details about the user interface. We describe the functions that the user interface should provide. In Fig.2.3, we organize these functions into a menu; the main menu of our application. Some of these functions are used more often than others, so we will need more handy ways to access them, but this depends on the capabilities of the programming environment.

We now describe these functions one by one.

- **File**: Various file operations.
  - *New*: Open a new file with a dummy name. Close the current one, if any (ask for confirmation, if it has not been saved).
  - *Open*: Open an existing file (ask the user for its name). Close the current one, if any (ask for confirmation, if it has not been saved).
  - *Include*: Include an existing file (ask the user for its name) in the current one, after all other included files.
  - *Close*: Close the current file (ask for confirmation, if it has not been saved).
  - *Save*: Save the current file. If no name has been given, same as *Save as*.
  - *Save as*: Save the current file (ask the user for its name).
  - *Import*: Read a text file that contains internal language statements.
  - *Export*: Write the current file as text.
  - *Print LaTeX*: Print the current file to a LaTeX document (ask the user for its name).
  - *Show source*: Show the source of the current file as it will be saved on disk (that is, show the internal language statements).
  - *Quit*: Quit the application (ask for confirmation, if the current file has not been saved).
- **Edit**: Editing commands.
  - *Open proof*: Open the proof of the current theorem for editing.

  - *Close proof*: Close the current proof.
  - *Cut*: Cut the selected text and place it in the clipboard.
  - *Copy*: Copy the selected text in the clipboard.
  - *Paste*: Insert the contents of the clipboard at the current editing point.
  - *Clear*: Delete the selected text.
  - *Show clipboard*: Show the contents of the clipboard.

- **Define**: Various definitions.
  - *Type*: Define a new type.
  - *Constant*: Define a new constant.
  - *Operator*: Define a new operator.
  - *Function*: Define a new function.
  - *Axiom*: Define a new axiom.
  - *Theorem*: Define a new theorem.
  - *Rule*: Define a new rule.
  - *Precedence*: Define operator precedence.
  - *Property*: Define operator properties.

- **Proof**: Editing the current proof.
  - *To fact*: Start a proof to a fact.
  - *From fact*: Start a proof from the current fact, with its variables substituted as specified by the user.

  - *By transformation*: Start a proof by transformation, starting from the expression that is specified by the user.
  - *Delete step*: Delete the last step in the current proof.
  - *QED*: Complete the current proof.

- **Transform**: Transforming the current expression.
  - *Apply*: Apply the current fact to the current subexpression (if possible) and create a transformation step.
  - *Substitute fact*: Substitute the current subexpression, which should be an instance of the current fact, by the default *true* value.
  - *Apply rule*: Apply the current user-defined rule, if possible.
  - *Insert parentheses*: Insert parentheses around the current subexpression.
  - *Remove parentheses*: Remove all unnecessary parentheses in the current subexpression.
  - *Perform substitution*: Perform all possible substitutions in the current subexpression.

- **Property**: Applying operator and quantifier properties.
  - *Symmetry*: Apply symmetry or duality of the outermost operator to the current subexpression, if possible.
  - *Conjunctionality*: Apply the conjunctionality of the outermost operator to the current subexpression, if possible.

  - *Equivalent form*: Substitute the outermost operator of the current subexpression by any of its equivalent forms.
  - *Missing range*: Apply the missing range quantifier property to the current subexpression, if possible.
  - *Empty range*: Apply the empty range quantifier property to the current subexpression, if possible.
  - *One-point rule*: Apply the one-point rule quantifier property to the current subexpression, if possible.
  - *Distributivity*: Apply the distributivity quantifier property to the current subexpression, if possible.
  - *Range split*: Apply the range split quantifier property to the current subexpression, if possible.
  - *Interchange dummies*: Interchange the dummies of the quantifier that is currently selected as the current subexpression.
  - *Nesting*: Apply the nesting quantifier property to the current subexpression, if possible.
  - *Rename dummies*: Rename the dummies of the quantifier that is currently selected as the current subexpression.
  - *Reorder dummies*: Reorder the dummies of the quantifier that is currently selected as the current subexpression.

- **Options**: Various options.

– *Proof*: Options about proofs (e.g. what hints should be displayed or in what form).

– *Directories*: Options about the directories, where the system or user files are found.

– *Load*: Load options from a file.

– *Save*: Save options to a file.

Chapter 3

User's Manual

3.1 Introduction

The proof editor that will be presented was developed in the Department of Computer Science, Cornell University, as an M.Sc. thesis by *Nikos Papaspyrou* and was supervised by *David Gries*. This tool can be used to facilitate the development of proof transformations. The current version of the proof editor is 1.0 *alpha* for Apple Macintosh. This version is not complete. For more details about unimplemented features and known bugs, refer to §3.6.

For a general presentation of the proof editor and for understanding its basics, the reader is directed to §2.1 and §2.2. The present document is a user's manual for the proof editor and assumes that the reader understands the principles of proof transformations and the basic terminology of this tool as described in its specifications document. The user should also be familiar with the standard Macintosh user interface and have some experience using typical Macintosh applications.

3.2 Working with files

A *module* is the fundamental document with which the proof editor works. Each module is associated with a file, from which it is loaded and to which it is saved. Modules can include other modules, thus creating a module *hierarchy*. The proof editor recognizes one module as the current one: the one being edited. All other modules (that are contained in the current one) can be used but not altered in any way. At most one file can be edited at a time: the file containing the current module.

Each module can contain definitions of types, constants, operators, functions, axioms, theorems, properties and rules. All this information is represented in the proof editor's *internal language*, which is described in §2.3. The proof editor works with two types of files: *proof editor files* and *text files*. The former can only be edited by the proof editor. The latter can also be edited by any text editor; however, the user needs to understand the internal language before editing these files.

3.2.1 Creating a new module

When the proof editor application is launched, the dialog box in Fig.3.1 appears. In this dialog box, the user can create a new empty module, by giving a module name and clicking button *OK*. Alternatively, the user can click button *Open* and then choose to open an existing file (see §3.2.2). The same dialog box appears when the user selects command `File|New`. If another module is being edited when this happens, it will be closed. A warning message will appear if that module has not been saved.

The user can quit the proof editor by selecting command `File|Quit`. Again, a

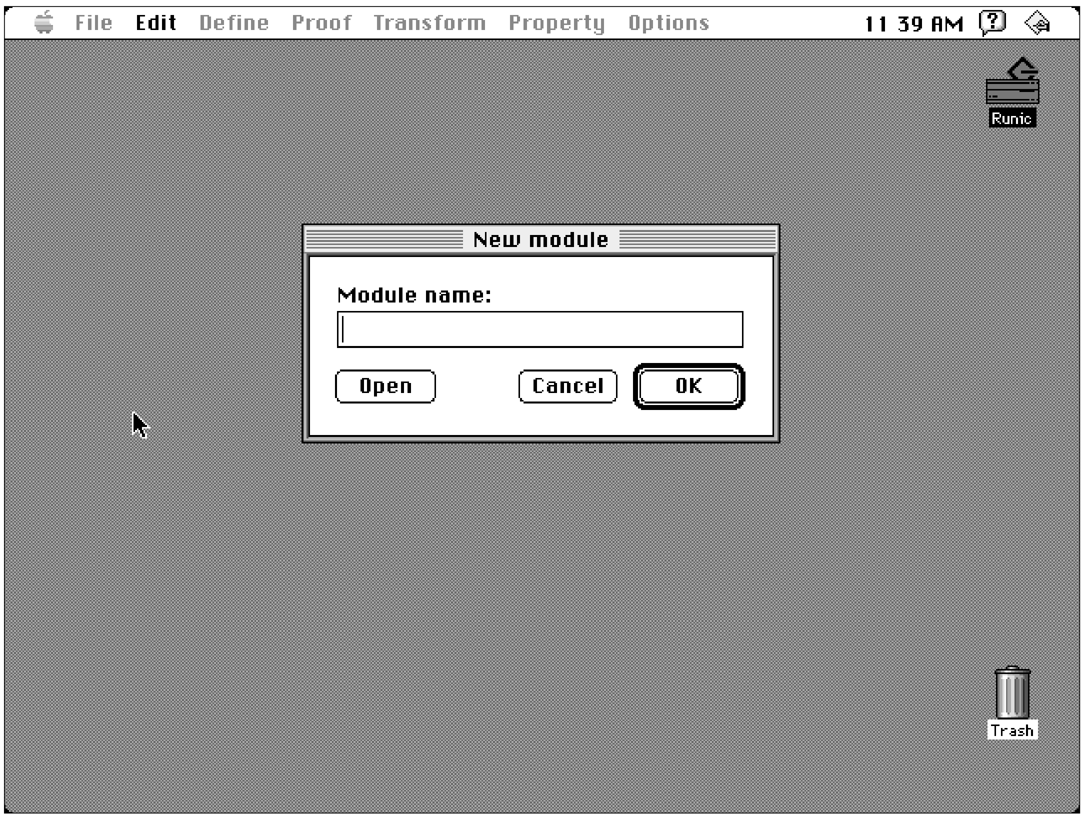


Figure 3.1: Creating a new module.

warning message will appear if the current module has not been saved.

3.2.2 Editing existing modules

By selecting command `File | Open`, the user can open an existing proof editor file. If another module is being edited at the time, it will be closed (the proof editor can work with one current module at a time). This command displays a standard dialog box for opening files, from which the user can select the file that he or she wants to open.

Command `File | Save` saves the current module. If this module was created by using command `File | New`, this is the same as selecting `File | Save as`. Otherwise, the module is saved in the file with which it has been associated. Command `File | Save as` first asks the user for the name of the file in which the current module is to be saved. A standard dialog box for saving files appears.

Finally, by selecting command `File | Close`, the user can close the current module. If this has not been saved, a warning message will appear and the user will be prompted to save it.

3.2.3 Working with text files

The proof editor lets the user work with text files. These files can be edited with any text editor and later be imported again, so that they can be used with the proof editor. Although this is not necessary and is against the philosophy of the proof editor, some people prefer working with text files. Furthermore, the proof editor does not allow all kinds of changes in a module (e.g. facts cannot be deleted once they have been defined), and sometimes the only way to change a module is by editing its internal language representation. The user must be very careful when editing these files as text and comply with the internal language's syntax, described in §2.3.

Command `File` | `Import` works the same as `File` | `Open`, only it opens text files. A standard dialog box appears when this command is selected. Command `File` | `Export` is equivalent to command `File` | `Save as` for text files.

3.2.4 Other `File` commands

The remaining two commands in the `File` menu are:

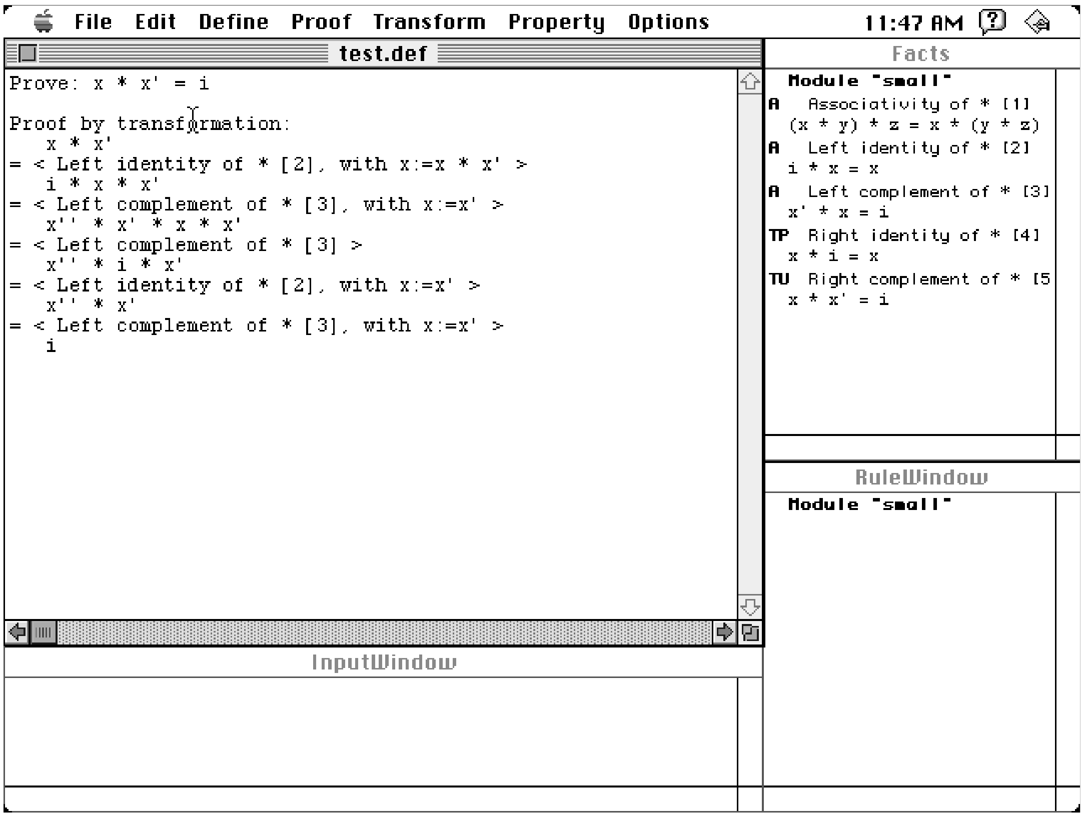
- `File` | `Print LaTeX`: Print the current module as a LaTeX file. It has not been implemented in this version of the proof editor.
- `File` | `Show source`: Display the representation of the current module in the proof editor's internal language.

3.3 Working with modules

This section describes how the user can edit a module's contents by defining new types, constants, operators, functions, axioms, theorems, properties and rules. For a description of editing proofs, see the next section.

3.3.1 Screen layout

When a module is being edited, the screen looks as in Fig.3.2. It is separated into four windows:



```
Prove: x * x' = i

Proof by transformation:
    x * x'
= < Left identity of * [2], with x:=x * x' >
    i * x * x'
= < Left complement of * [3], with x:=x' >
    x'' * x' * x * x'
= < Left complement of * [3] >
    x'' * i * x'
= < Left identity of * [2], with x:=x' >
    x'' * x'
= < Left complement of * [3], with x:=x' >
    i
```

```
  Module "small"
A   Associativity of * [1]
  (x * y) * z = x * (y * z)
A   Left identity of * [2]
  i * x = x
A   Left complement of * [3]
  x' * x = i
TP  Right identity of * [4]
  x * i = x
TU  Right complement of * [5
  x * x' = i
```

```
  Module "small"
```

Figure 3.2: Screen layout.

- *The proof window*: In this window, the biggest window on the screen, the current proof is displayed (when a proof is being edited).

- *The facts window*: This window contains a list of all facts (axioms and theorems) defined in the current module and all modules that this one includes. The facts are sorted by module and are given in the order in which they have been defined. By clicking on a module's name in the facts window, the user can hide its facts. They can be shown again with a second click.

  Each fact is displayed in two lines. The first line contains its title and number, preceded by one to three letters characterizing the fact. The first letter is one of the following:

  A: the fact is an axiom.

  T: the fact is a theorem.

  a: the fact is an assumption.

  In case the fact is a theorem, the second letter is one of the following:

  P: the theorem has been proved.

  U: the theorem has not been proved.

  Finally, the third letter is an optional **c**, meaning that the fact can be applied under some conditions.

- *The rules window*: This window contains a list of all user-defined rules defined in the current module and all modules that this includes. The rules are sorted by module and are given in the order in which they have been defined. By clicking on a module's name in the rules window, the user can hide its rules. They can be shown again with a second click. For each rule, only its name is displayed.

- *The input window*: This window is used as a means of communication between the proof editor and the user.

3.3.2 Including existing modules

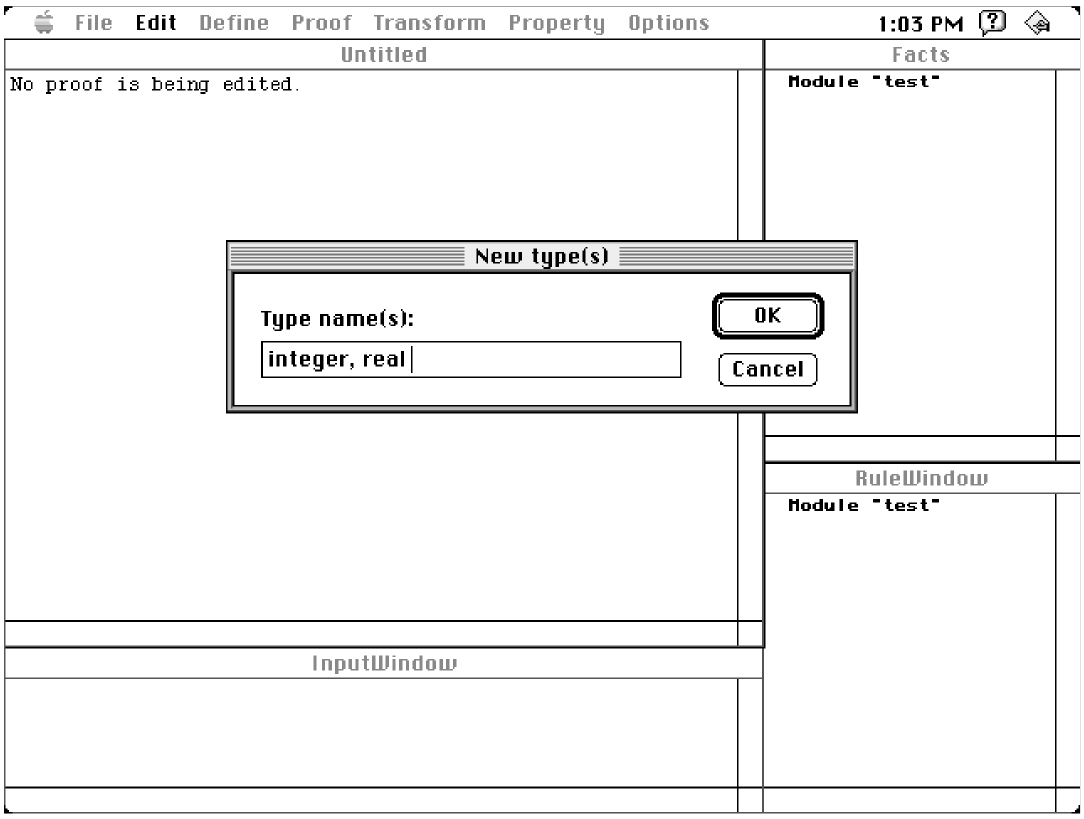
By using command `File | Include`, the user can include other existing modules in the current module. A standard dialog box appears, which enables the user to choose the proof editor file to be included.

3.3.3 Defining types

New types can be defined using command `Define | Type`. The dialog box in Fig.3.3 appears. The user must specify the names of one or more types to be defined, separated by commas or spaces.

3.3.4 Defining constants

New constants can be defined using command `Define | Constant`. The dialog box in Fig.3.4 appears. The user must specify the names of one or more constants to be defined, separated by commas or spaces, and select their type from the popup menu that appears.



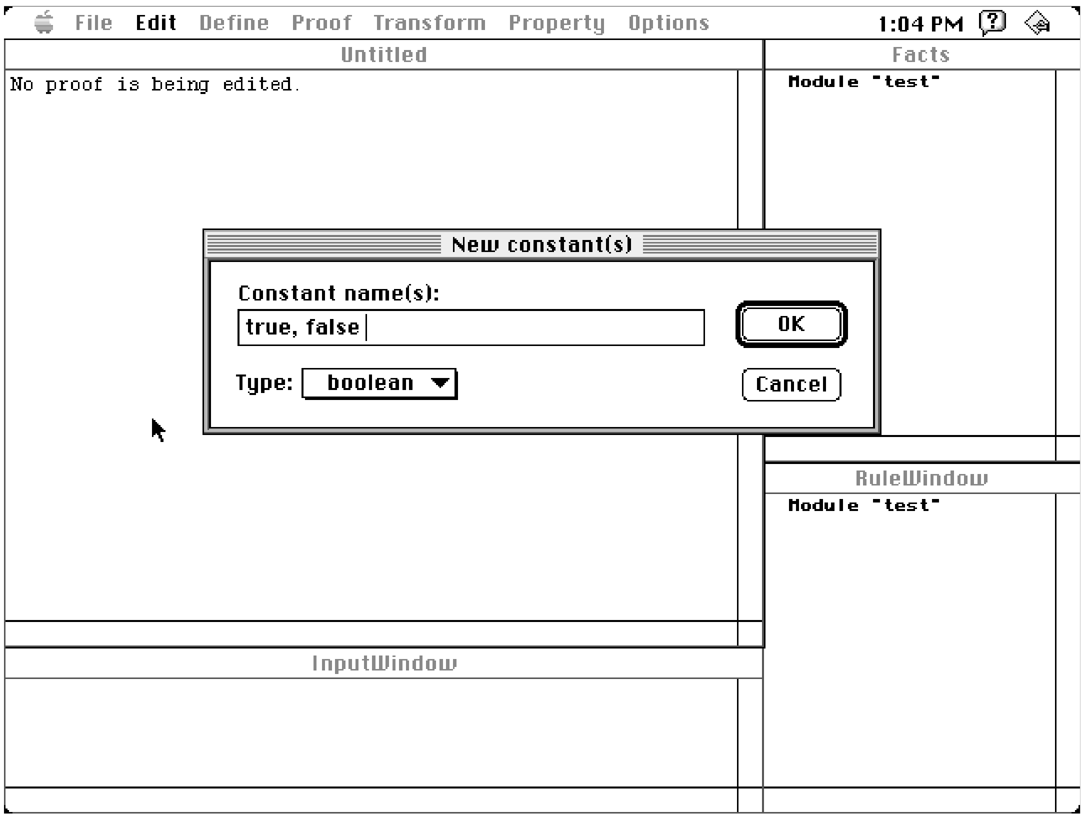
No proof is being edited.

Module "test"

integer, real

Module "test"

Figure 3.3: Defining types.



No proof is being edited.

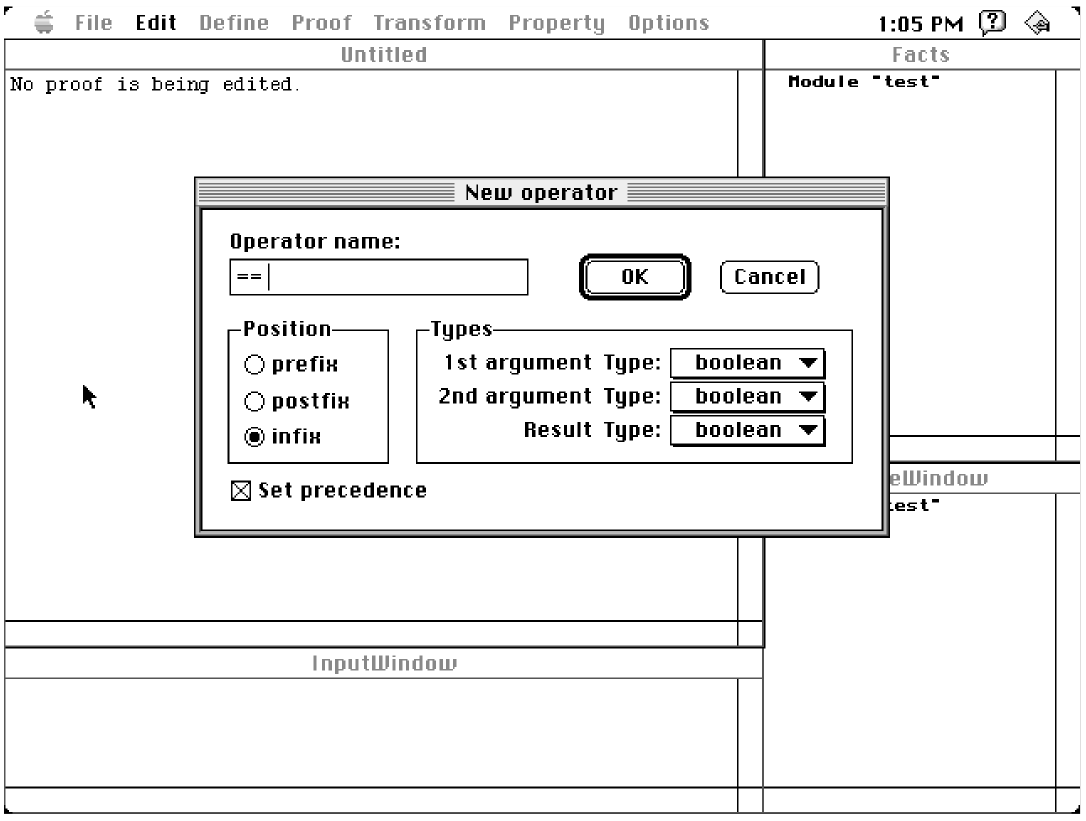
Module "test"

Module "test"

Constant name(s): true, false

Type: boolean

Figure 3.4: Defining constants.



No proof is being edited.

Module "test"

Operator name: ==

Position: prefix, postfix, [✓ infix]

Types

1st argument Type: boolean

2nd argument Type: boolean

Result Type: boolean

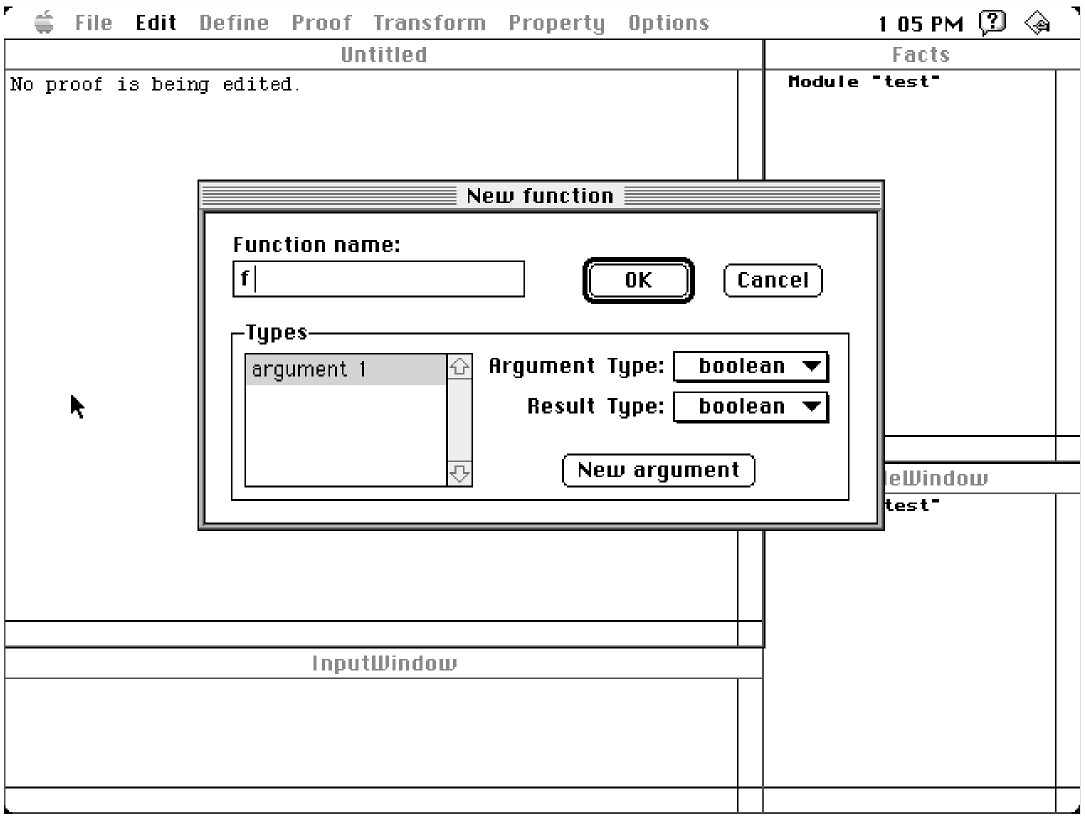
[✓ Set precedence]

Figure 3.5: Defining operators.

3.3.5 Defining operators

By using command `Define` | `Operator`, the user can define a new operator. The dialog box in Fig.3.5 appears. The user must specify the name of the operator, its position (i.e. prefix, postfix or infix), the types of its arguments (prefix and postfix operators take one argument, while infix operators take two arguments) and the result type.

If box *Set precedence* is checked, after defining an operator the user will be asked to place it in the operator precedence table.



No proof is being edited.

Module "test"

Function name: f

argument 1

Argument Type: boolean

Result Type: boolean

Figure 3.6: Defining functions.

3.3.6 Defining functions

By using command `Define|Function`, the user can define a new function. The dialog box in Fig.3.6 appears. The user must specify the name of the function, the types of its arguments and the result type. Initially, a newly-defined function has no arguments (which is not legal). By clicking on button *Add argument*, a new argument is added and the user must specify its type.

3.3.7 Defining axioms and theorems

New axioms and theorems can be defined using the two commands `Define|Axiom` and `Define|Theorem` respectively. These two will be described together, since the process is exactly the same. A dialog box similar to the one in Fig.3.7 appears. The user must specify the fact's title and number (they can be left blank, but leaving both blank is not a good idea) and the fact's expression. Then, by clicking button *Find free variables*, the fact's expression is parsed and all free variables are placed in a list. The user must then specify each free variable's type from the type popup menu.

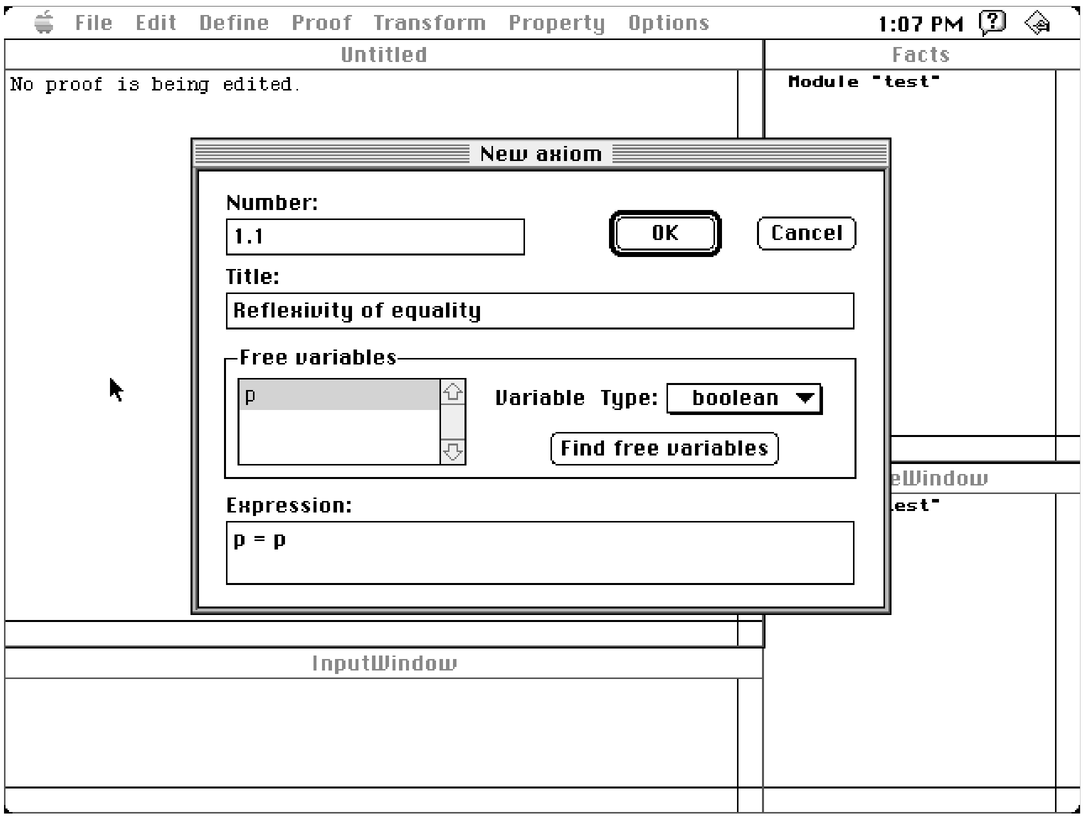
3.3.8 Defining rules

The user can define new rules using command `Define|Rule`. The dialog box in Fig.3.8 appears, where the user must specify the rule's title and its text. The text of the rule is its representation in the proof editor's internal language, omitting the initial `rule` and the title and ending with `endrule`. For more information about the syntax and semantics of rules, read §2.3.9 and §2.4.4.

3.3.9 Defining operator precedence

When a new operator is defined, its precedence with respect to other operators is undefined. This means that the operator cannot be used in expressions containing other operators, without explicit parentheses. In order to define an operator's precedence, command `Define|Precedence` must be used. The dialog box in Fig.3.9 appears.

This dialog box has two tables. The upper table is the precedence table. If



No proof is being edited.

Module "test"

Number: 1.1

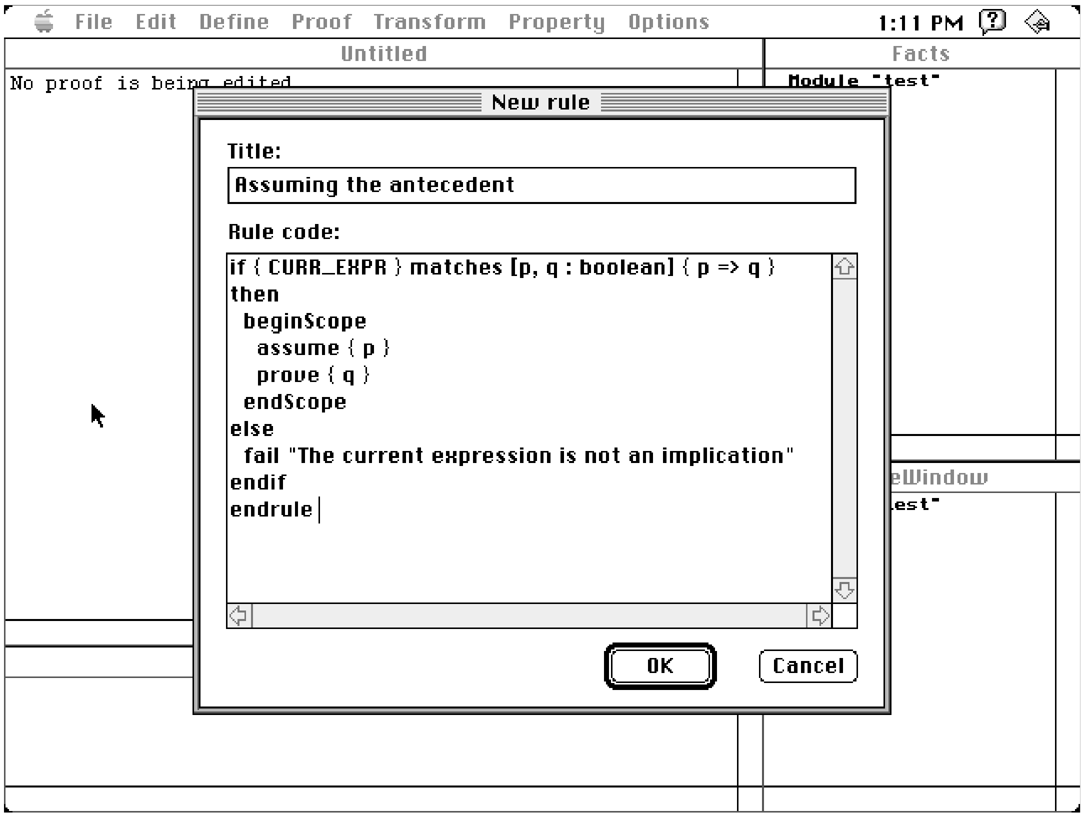
Title: Reflexivity of equality

Free variables: p

Variable Type: boolean

Expression: p = p

Figure 3.7: Defining axioms and theorems.



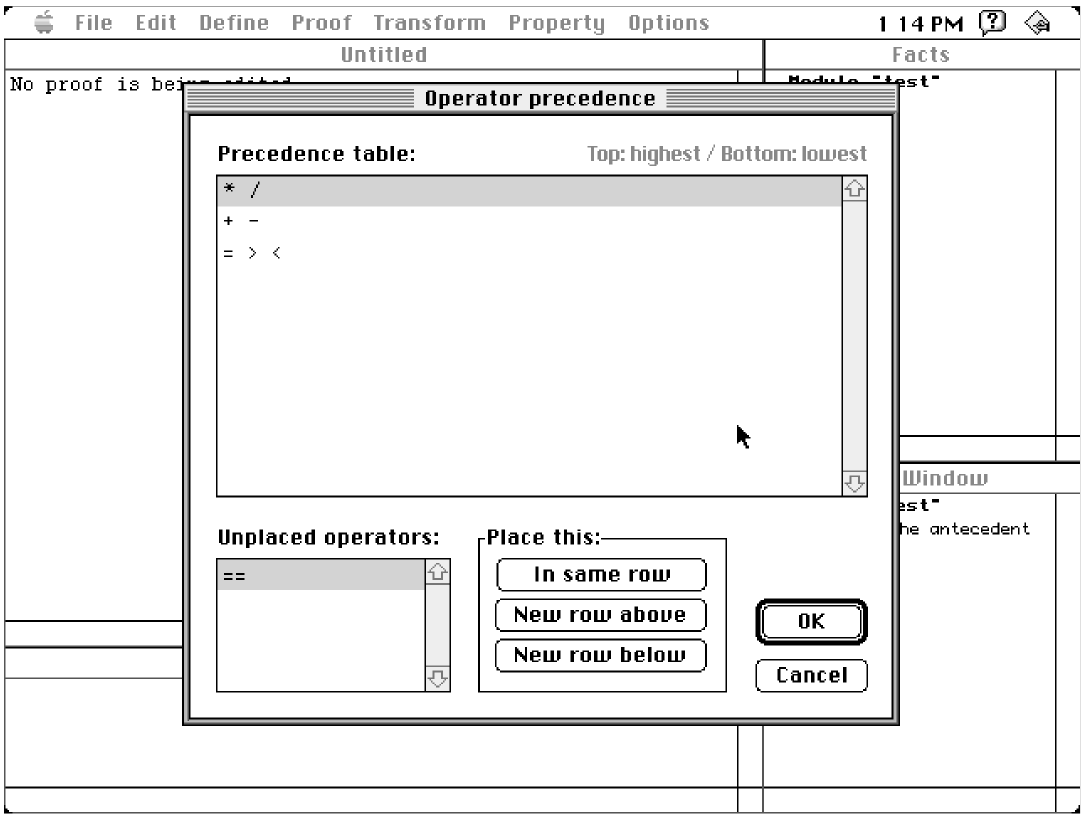
Title:

Assuming the antecedent

Rule code:

```
if { CURR_EXPR } matches [p, q : boolean] { p => q }
then
  beginScope
    assume { p }
    prove { q }
  endScope
else
  fail "The current expression is not an implication"
endif
endrule
```

Figure 3.8: Defining rules.



No proof is bei

Module "test"

est"

he antecedent

Operator precedence

Precedence table: Top: highest / Bottom: lowest

```
* /
+ -
= > <
```

Unplaced operators:

```
==
```

Place this:

In same row

New row above

New row below

Figure 3.9: Defining operator precedence.

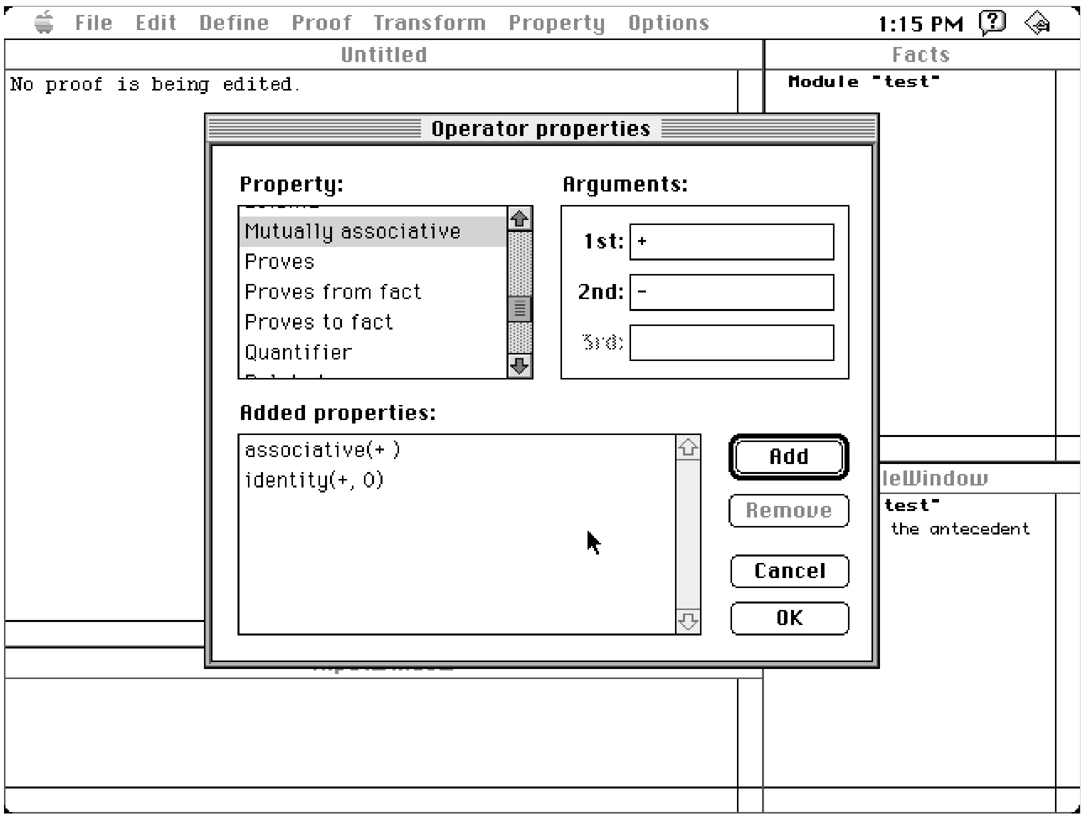
an operator has been placed in this table, its precedence with respect to all other operators has already been defined. This table consists of several rows. In each row, one or more operators can be placed. If two operators are in the same row, they have the same precedence. Operators in rows toward the top of the table bind stronger than operators toward its bottom.

The lower table contains operators that have not yet been placed in the precedence table. To place such an operator in the precedence table, select the operator, select a row in the precedence table and click at one of the three placement buttons:

- *In same row*: place the operator in the row that has been selected.
- *New row above*: create a new row just above the one that has been selected and place the operator there.
- *New row below*: create a new row just below the one that has been selected and place the operator there.

3.3.10 Defining operator properties

By using command `Define`|`Property`, the user can define operator properties (internal language predefined rules, see §2.2.5). The dialog box in Fig.3.10 appears. To define an operator property, the user must select the property's name and specify its arguments. Then, by clicking on button *Add*, the property is placed in the list of properties to be defined (it can be removed by clicking on button *Remove*).



No proof is being edited.

Module "test"

Property: Mutually associative, Proves, Proves from fact, Proves to fact, Quantifier

1st: +
2nd: -
3rd:

associative(+)
identity(+, 0)

test"
the antecedent

Figure 3.10: Defining operator properties.

3.4 Working with proofs

In order to edit a theorem's proof, the user selects the theorem from the facts window and clicks command `Edit`|`Open proof`. The proof window then displays the theorem's proof. If the proof is not completed, the user can edit it by using the commands in menus `Proof`, `Transform` and `Property`. Otherwise the proof cannot be changed. To close the proof that is currently being edited, the user clicks command `Edit`|`Close proof`.

When editing a proof, the last expression appearing in the proof window is considered to be the *current expression*. This expression is used in the next transformation step. The user can drag the mouse to select a part of this expression: the *current subexpression*. If a subexpression is selected, only this subexpression will be used in the next transformation step.

3.4.1 Starting a proof

There are three ways to start a proof, each corresponding to a predefined proof technique (see §2.2.7). The three relevant commands are:

- `Proof`|`To fact`: Start a proof to a fact.
- `Proof`|`From fact`: Start a proof from a fact. The currently selected fact (in the facts window) is used as the fact to start from. The user can type a substitution of this fact's free variables in the input window.
- `Proof`|`By transformation`: Start a proof by transformation. The user must type the starting expression in the input window.

3.4.2 Using facts

One way to add steps to a proof is by using proved facts. Such facts can be either *applied* or *substituted.* The corresponding commands are `Transform|Apply fact` and `Transform|Substitute fact`. The user has to select the fact to be used from the facts window and optionally type a substitution of its free variables in the input window.

When using `Transform|Apply fact`, the current subexpression is matched with part of the current fact (after all user-specified substitutions have taken place). The fact's free variables are matched with terms in the current subexpression. If there are still free variables after the matching, they are fixed. A transformation step is created, according to what is described in §2.2.6.

When using `Transform|Substitute fact`, if the current subexpression is the default *true* value, it is replaced by the current fact (after all user-specified substitutions have taken place and remaining free variables are fixed). Otherwise, the current subexpression is matched with the current fact. If it is an instance of this, a transformation step is created (with the equality operator) and the current expression is substituted by the default *true* value.

It should be noted here that, although the matching of the current subexpression with the current fact is always correct (that is, the transformation steps are always valid), it is not necessarily what the user expects. Free variables can usually be matched in various ways, resulting in completely different transformation steps. If more than one way is possible, the proof editor will choose one of them. If the user wanted something different, they have to delete the step and try again by specifying the substitutions manually. To delete the last transformation step in the current proof, the user can select command `Proof | Delete step`.

3.4.3 Completing proofs

A proof can be completed by selecting command `Proof | QED`. Completing a proof is not always valid; the proof's correctness is checked when the user selects this command, as described in §2.2.7. Completing a proof does not automatically close it. It can be undone by selecting `Proof | Delete step`. However, after closing a complete proof, the user cannot ever change it again (unless they change the module's internal representation).

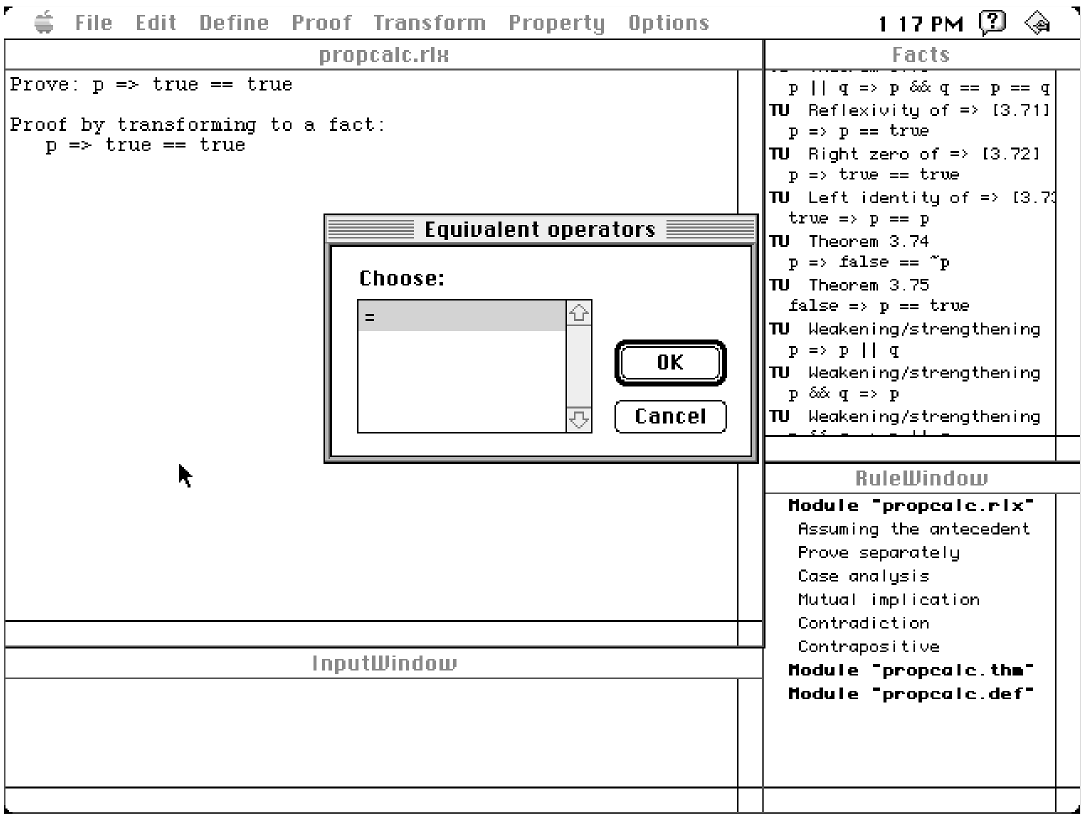
3.4.4 Applying operator/quantifier properties

Operator and quantifier properties can be applied to the current subexpression by using the commands of menu `Property`. When applying an operator property, the current subexpression's outermost operators (see §2.4.2) must be infix operators. When applying a quantifier property, the current subexpression must be an instance of one of the two sides of a quantifier predefined property (see §2.2.8). For all these properties, a transformation step with the equality operator is created.

The operator properties that can be applied are the following:

- *Symmetry*: If the outermost operator of the current subexpression is symmetric, its two operands are reversed. Otherwise, if it has a dual, its two operands are reversed and the operator is replaced by its dual.

- *Conjunctionality*: If the current subexpression is a chain of conjunctional opera-



```
Prove: p => true == true

Proof by transforming to a fact:
   p => true == true
```

```
  p || q => p && q == p == q
TU  Reflexivity of => [3.71]
  p => p == true
TU  Right zero of => [3.72]
  p => true == true
TU  Left identity of => [3.7
  true => p == p
TU  Theorem 3.74
  p => false == ~p
TU  Theorem 3.75
  false => p == true
TU  Weakening/strengthening
  p => p || q
TU  Weakening/strengthening
  p && q => p
TU  Weakening/strengthening
```

```
Module "propcalc.rlx"
  Assuming the antecedent
  Prove separately
  Case analysis
  Mutual implication
  Contradiction
  Contrapositive
Module "propcalc.thm"
Module "propcalc.def"
```

Figure 3.11: Operator equivalent form.

tors, it is transformed into a conjunction of these operators. If it is a conjunction of conjunctional operators, it is transformed into a chain.

- *Equivalent form*: The dialog box of Fig.3.11 appears and the user must choose one of the equivalent forms of the current subexpression's outermost operator.

The quantifier properties that can be applied are described in §2.2.8. These have not been implemented in the current version of the proof editor, so they will not be described in more detail now.

3.4.5 Parentheses and substitutions

There are three more commands that create transformation steps. All of them create transformation steps with the equality operator. These are:

- `Transform|Insert parentheses`: Insert parentheses around the current subexpression.
- `Transform|Remove parentheses`: Remove all unnecessary parentheses from the current subexpression.
- `Transform|Perform substitution`: Perform all possible substitutions in the current subexpression.

3.4.6 Applying user-defined rules

By using command `Transform|Apply rule`, the user can apply a user-defined rule. The syntax and semantics of user-defined rules can be found in §2.3.9 and §2.4.4 respectively. To apply a rule, the user needs only select it from the rules window. Its application may depend on the current expression or subexpression, the current fact and the contents of the input window.

3.5 Error messages

This section presents and explains briefly the various error messages that the proof editor will display when something goes wrong. For each error message, the line of code that generated it is shown (for debugging purposes) and, if the error is related to text parsing, its context is given (e.g. file name and line number). Errors are categorized as follows.

- *Internal errors*: Such errors should never happen. They mean that there is a bug in the proof editor.
- *Fatal errors*: Errors that cause the proof editor application to stop.
- *Common errors*: Errors from which the proof editor can recover.
- *Warning messages*: Usually not important possible errors.

If an internal error happens, please send the author a report containing the error message and a description of how it happened (as detailed as possible). The same if the system crashes, or if the proof editor does not behave as it should (make sure to look at §3.6, though).

Here is a list of various common error messages, in alphabetical order:

`Assertion failed`

Internal error. An assertion failed. Please report this.

`Cannot apply conjunctionality to this expression`

Operator conjunctionality cannot be applied to the current subexpression.

`Cannot apply symmetry/duality to this expression`

Operator symmetry or duality cannot be applied to the current subexpression.

`Cannot change existing proof mode`

A proof mode has already been selected for the current proof and cannot be changed. To change it, you have to delete all steps of the proof.

`Cannot open a dependent scope in this proof`

A dependent scope can be opened only in a proof to a fact.

`Cannot open file` <*filename*>

File <*filename*> cannot be opened, for reading or writing. If for reading, it is possible that the file does not exist or cannot be found. If for writing, it is possible that the file already exists and cannot be overwritten.

`Cannot read from file` <*filename*>

File <*filename*> cannot be read. Perhaps there is a problem with the disk.

`Cannot resolve operator precedence`

Operator precedence between these operators has not been defined. The expression containing them is ambiguous.

`Cannot set precedence (check all precedences)`

Operator precedence cannot be set, because it is already defined. Check all precedences to find where the problem is.

`Cannot write to file` <*filename*>

File <*filename*> cannot be written. Perhaps there is a problem with the disk, or the disk is full.

`Constant cannot be` <*property*>

This constant cannot be given this property. Possible reasons are type incompatibility, or another constant has already been given the same property. Check the conditions for this property in chapter 2.

`Different array in EXP`

A different array is given in an `EXP` expression from what was previously declared in parentheses.

`Identifier has already been defined`

This identifier has already been defined for something else. Use unique identifier names.

`Incomplete proof`

This proof cannot be completed. For details on proof correctness, read §2.2.7.

`Invalid module, expected 'module'`

This file does not contain a valid module. A text file must start with `module`.

`Misplaced identifier`

A constant or variable identifier is found in an expression in a place where an operator is expected.

`Misplaced operator`

An operator is found in an expression in a place where a term is expected.

`No current expression exists`

No current expression exists in the current proof.

`No default` <*value*> `has been defined`

A default value is needed and has not been defined.

`No proof is being edited`

Some command that requires a proof to be edited was used without a current proof.

`Operator(s) cannot be` *<property>*

This operator cannot be given this property. Possible reasons are type incompatibility, or another operator has already been given the same property. Check the conditions for this property in chapter 2.

`Operator has no` *<property>*

This property is needed and has not been defined for this operator.

`Operator precedence not totally ordered`

The operator precedence is not totally ordered. This operator will not be placed in the precedence table.

`Operator precedence over itself is forbidden`

You cannot define that an operator has higher precedence than itself.

`Out of memory`

The proof editor application has run out of memory. Try increasing the memory from Finder's command `File`|`Get Info`.

`Quantifier cannot be defined`

This quantifier cannot be defined. Check the conditions specified in chapter 2.

`Rule cannot be applied:`

This error message is the result of execution of a `fail` user-defined rule statement.

`Select a fact first`

You cannot apply a fact, because no fact is currently selected in the facts window.

`Select a rule first`

You cannot apply a rule, because no rule is currently selected in the rules window.

`Select a theorem first`

You have selected `Edit|Open proof` without a theorem being selected in the facts window.

`Syntax error in rule condition`

Check the user-defined rule text for a syntax error (see §2.3).

`The current fact cannot be applied`

The current fact cannot be applied to the current subexpression, because no matching was found or the outermost operator is not a connective.

`The current fact cannot be used (conditions not satisfied)`

The current fact's conditions are not satisfied, so the fact cannot be used.

`The current fact cannot be used (not proved)`

The current fact is an unproved theorem, so it cannot be used.

`The current fact cannot be used (out of range)`

The current fact is out of the range of theorems that can be used.

`The current subexpression is invalid`

What is currently selected in the proof window is not a valid subexpression of the current expression.

`The current subexpression is not a fact`

You have selected `Transform|Substitute fact`, but the current subexpression is neither the default *true* value nor an instance of the current fact.

`There is no equivalent form for this expression`

There are no equivalent forms for the current subexpression's outermost operator.

`This statement should never be executed`

Internal error. A statement was executed that was not supposed to. Please report this.

`Too few arguments in call to function`

A function call has fewer arguments than necessary.

`Too few expressions in substitution`

There are fewer expressions than variables in a substitution.

`Too many arguments in call to function`

A function call has more arguments than necessary.

`Too many expressions in substitution`

There are more expressions than variables in a substitution.

`Type mismatch`

A different type was found from the one expected.

`Undefined` <*object*>

An object is used but not defined.

`Unknown PED exception`

Internal error. Something wrong with the exception handler. Please report this.

`Unrecognized` <*object*>

This object is not recognized. Perhaps a spelling mistake?

`Unterminated comment`

A comment has not been terminated, and the end of the file has been reached.

`Unterminated string`

A string has not been terminated, and the end of the file has been reached.

3.6 Unimplemented features and known bugs

Here is a list of features that have not been implemented in the current version of the proof editor, although they are present in chapter 2.

1. Quantifier properties have not been implemented. Many things about quantifiers have not been thoroughly tested, so it can be said that, in this version, quantifiers do not work.

2. User-defined rules containing `EXP` terms do not work.

3. Dependent proof scopes have been implemented, but the current interface does not support them. They cannot be used in this version.

4. Facts that can be applied under conditions have been implemented but cannot be created using the current interface. Fact conditions have not been thoroughly tested.

5. There is no way to hide *obvious* hints (e.g. operator symmetry, conjunctionality, etc.). This should be in the options menu, which has not been implemented. For the same reason, all files have to be in the same directory as the application file (unless full path names are used when including files).

6. Operator idempotency, associativity, identities and zeroes cannot be applied *on the fly* in transformation steps. The author thinks that this is not necessary

(usually there is a fact doing the same thing and it is not desired to hide the hints), except for associativity.

7. There is no default type for free variables and quantifier dummies. All types have to be specified explicitly.

8. Command `File|Print LaTeX` has not been implemented.

9. It is not possible to restrict the range of facts to be used when proving a theorem.

The following features are not present in chapter 2 but are considered to be useful and are treated as "unimplemented features".

1. There is no way to write comments in a proof, especially when applying rules (e.g. "Proof by contradiction:", etc.).

2. It is not possible to see information about facts and rules, apart from command `File|Show source`. There should be something easier there.

3. It is not possible to see information about other symbols, especially operators. It would be useful to be able to see a list of all properties defined for a particular operator.

4. Before selecting something in a window, the user has to click the window to make it active. Therefore, for selecting something in a non-active window two clicks are necessary. Besides, the selected lines in a non-active window are not shown.

5. There should be an option to print a proof separately (in a LaTeX file) instead of the whole file.

Finally, here is a list of known bugs (some very annoying) of the current version:

1. It is possible to edit a proof of a theorem that is defined in an included file. By saving the file, this proof is not saved.

2. Clicking button *Cancel* in the first *New module* dialog box and then quitting the application causes a system crash.

3. The system may crash in low-memory situations (although a low-memory error handler has been implemented). This may be because exceptions are not handled properly (some flag was wrong when compiling?).

4. Some fatal errors should not be fatal.

Bibliography

[BVW94] Roland Backhouse, Richard Verhoeven, and Olaf Weber. Mathpad: User Manual. Technical report, Department of Computer Science, Eindhoven University of Technology, January 1994.

[C$^+$86] Robert Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, N.J., 1986.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, New York, N.Y., 1993.

[GS94a] David Gries and Fred B. Schneider. A New Approach to Teaching Mathematics. Technical report, Department of Computer Science, Cornell University, February 1994.

[GS94b] David Gries and Fred B. Schneider. Teaching Math More Effectively, Through the Design of Calculational Proofs. Technical report, Department of Computer Science, Cornell University, March 1994.

[vdS94a] Jan L.A. van de Snepscheut. Mechanized Support for Stepwise Refinement. Technical report, Department of Computer Science, California Institute of Technology, January 1994.

[vdS94b] Jan L.A. van de Snepscheut. Proxac: an Editor for Program Transformation. Technical report, Department of Computer Science, California Institute of Technology, February 1994.