PESCA — A Proof Editor for Sequent Calculus

Aarne Ranta
Aarne.Ranta@cs.chalmers.se

Second Version, March 24, 2000

Abstract. PESCA is a program that helps in the construction of proofs in sequent calculus. It works both as a proof editor and as an automatic theorem prover. Proofs constructed in PESCA can both be seen on the terminal and printed into LaTeX files. The user of PESCA can choose among different versions of classical and intuitionistic proposition and predicate calculi, and extend them by systems of nonlogical axioms. The aim of this document is to show how to use PESCA. It will also give an outline of the implementation of PESCA, which is written in the functional programming language Haskell.

1 Introduction

Sequent calculus was introduced by Gentzen (1934) as a formalization of logic suitable for metamathematical considerations, such as consistency proofs. Gentzen also realized that sequent calculus gives a method of automatic proof search for intuitionistic proposition calculus. Both of these aspects were built upon in later works on proof theory, such as in Kleene (1952) and Schütte (1977), and sequent calculus is still a living subject. At the same time, it was already realized by Gentzen that sequent calculus is not very natural for humans actually to write proofs. It carries around a lot of information that humans tend to keep in their heads rather than to put on paper. While greatly improving the performance of machines operating on proofs, this information easily obscures the human inspection of them, and actually writing sequent calculus proofs in full detail is tedious and error-prone. Thus it is obviously a task where a machine can be helpful.

This document will not give an introduction to sequent calculus, let alone to logic. In developing the system PESCA, we have followed the book manuscript Negri & von Plato (1999), but the sufficient background can also be gained from Troelstra & Schwichtenberg (1996). The system PESCA is not, however, fully compatible with the "older generation" of sequent calculi *à la* Gentzen and Schütte, for reasons that will become clear in the following.

The domain of sequent calculi allows for indefinitely many variations, which are not due to disagreements on what should be provable but to different decisions on the fine structure of proofs. In terms of provability, it is usually enough to tell whether a calculus is **intuitionistic** or **classical**. In the properties of proof structure, there are many more choices. The very implementation of PESCA precludes most of them, but it still leaves

room for different calculi, only some of which are included in the basic distribution. These calculi can be characterized as having

shared multiset contexts, no structural rules.

But calculi can have single as well as multiple formulae in the succedents of their sequents.

The fundamental common property of the calculi treated by PESCA is **top-down determinacy**:

given a conclusion and a rule, the premises are determined.

This property is essential for our method of **top-down proof search**. The user of PESCA starts with a conclusion, and then tries to **refine** it by suggesting a rule. If the rule is applicable, the proof of the conclusion is completed by the derived premises, and the proof search can continue from them. A branch in a proof is successfully terminated when a rule gives an empty list of premises. A branch fails if no rule applies to it. This simple procedure, which we have adopted from the proof editor ALF (Magnusson 1994) for the much richer formalism of constructive type theory, is not applicable to calculi that are not top-down determinate.

(The term top-down runs counter to the standard typographical layout of proof trees, where premises appear above conclusions. The term is frequent in computer science, where it may come from the standard layout of syntax trees, where the root is above the trees. In proof theory, the confusion is usually avoided by saying "root first" instead, as do Negri & von Plato (1999).)

2 Two example sessions

2.1 A proof in proposition calculus

Let us first construct a proof of the law of commutativity for disjunction, in the sequent form,

$$A \vee B \Rightarrow \ B \vee A.$$

Having started PESCA (see Section 5 below to find out how), we see its prompt `|-`. We then enter a **new goal** by the command `n` followed by the sequent written in ASCII notation:

```
|- n A v B => B v A
```

The reply of PESCA is a new **proof tree**, consisting of the conclusion alone, which is the only **subgoal** of the tree. Subgoals are identified by sequences of digits starting from the root, as in the following example:

$$\dfrac{\dfrac{111 \quad 112}{11} \quad \dfrac{121}{12} \quad 13}{1}$$

The command `s` shows the current subgoals, of which we still have just one, numbered 1. More interestingly, the command `a` shows the **applicable rules** for a given subgoal. In the situation where we are in our proof, we have

```
|- a 1

r 1 A1 S1 Lv  -- A v B => B v A
r 1 A1 S1 Rv1 -- A v B => B v A
r 1 A1 S1 Rv2 -- A v B => B v A
```

Any of the displayed `r` commands (line segments preceding `--`) can be cut and pasted to a command line, and it gives rise to a **refinement** of the subgoal, an extension of the tree that is determined by the chosen rule. For instance, choosing the first alternative takes us to a proof by the left disjunction rule from two premises,

```
|- r 1 A1 S1 Lv

A => B v A   B => B v A
-----------------------
A v B => B v A
```

As will be explained below, the part `A1 S1` specifies the active formulae in the antecedent and the succedent; when the first formulae are chosen, like here, this part could be omitted, like in the next refinement,

```
|- r 11 Rv2

A => A
----------
A => B v A   B => B v A
-----------------------
A v B => B v A
```

The left branch `111` can now be refined by the `ax` rule, which does not generate any more subgoals:

```
|- r 111 ax
```

The right branch `12` can be refined analogously with `11`, by `Rv1` followed by `ax`. When the proof is ready, its ASCII appearance is usually not of very good quality. Now, at the latest, it is rewarding to use the `l` command to write the current proof into a LaTeX document, which looks as follows when processed:

$$\dfrac{\dfrac{A \Rightarrow A^{ax}}{A \Rightarrow B \vee A}\,{\scriptstyle R\vee 2} \qquad \dfrac{B \Rightarrow B^{ax}}{B \Rightarrow B \vee A}\,{\scriptstyle R\vee 1}}{A \vee B \Rightarrow B \vee A}\,{\scriptstyle L\vee}$$

2.2 A proof in predicate calculus

In predicate calculus, one more command is usually needed than in propositional calculus: the command `i` for **instantiating** parametres. Logically, a parametre such as `t` in the existential introduction rule $R\exists$

$$\dfrac{\Gamma \Rightarrow A(t/x)}{\Gamma \Rightarrow (\exists x)A}$$

is just like another premise, which calls for a construction to be complete. This logic is made explicit in the Σ introduction rule of Martin-Löf's type theory, and simplifies greatly the implementation of inference rules. Here, staying faithful to the syntax of predicate calculus, we have to treat such parametres as hidden premises in the rules. Thus a proof that has uninstantiated parametres is incomplete in the same way as a proof that has open subgoals.

Let us prove the quantifier switch law,

$$(\exists y)(\forall x)C(x,y) \Rightarrow (\forall x)(\exists y)C(x,y).$$

After introducing the goal, we make a couple of ordinary refinements:

```
|- n (/Ey)(/Ax)C(x,y) => (/Ax)(/Ey)C(x,y)
|- r 1 A1 S1 R/A
|- r 11 L/E
|- r 111 A1 S1 R/E
```

The last refinement introduces a parametre `t`, which we instantiate by `y`,

```
|- i t y
```

Continuing by `L/A`, `x`, and `ax`, we obtain a complete proof, in which we have afterwards marked the two instantiations made:

$$\dfrac{\dfrac{\dfrac{\dfrac{C(x,y), (\forall x)C(x,y) \Rightarrow C(x,y)^{ax}}{(\forall x)C(x,y) \Rightarrow C(x,y)}\, L\forall x}{(\forall x)C(x,y) \Rightarrow (\exists y)C(x,y)}\, R\exists y}{(\exists y)(\forall x)C(x,y) \Rightarrow (\exists y)C(x,y)}\, L\exists}{(\exists y)(\forall x)C(x,y) \Rightarrow (\forall x)(\exists y)C(x,y)}\, R\forall$$

3 The command language

This section gives a systematic description of all PESCA commands. Some of them were already exemplified in the two sessions of the previous section.

Each command consist of one character followed by zero or more arguments, some of which may be optional. In the following, as in the on-line help file of PESCA, the arguments are denoted by words indicating their types. Optional arguments are enclosed in brackets. Typewriter font is used for terminal symbols.

Fully to understand the commands, one should know that a PESCA session takes place in an **environment**, which changes as a function of the commands. The environment consists of a **current calculus** and a **current proof**. In the beginning, the calculus is the intuitionistic predicate calculus G3i (cf. Appendix), and the proof is the one consisting of the impossible empty sequent $\Rightarrow$. While many of the commands affect the current proof, only two of them affect the current calculus.

`r` goal [`A` int] [`S` int] rule (refine)

replaces the goal by an application of the rule, if applicable, and leaves the current proof unchanged otherwise. The goal is denoted by a sequence of digits, as explained in the previous section. The options `[A` int`]` `[S` int`]` reset the active formulae in the antecedent and the succedent of the goal—by default, the active formula is number 1, which in the antecedent is the first and in the succedent the last printed formula. If the number exceeds the length, resetting the active formula has no effect.

`i` parametre term (instantiate)

replaces all occurrences of the parametre in the current proof by the term. The term can be any variable or constant or other parametre, or a complex term built by function applications. Notice that, if a parametre occurs in the term, it must be prefixed by the character `?`, so that it is not parsed as a variable (cf. Section 4.3 below).

`t` goal int (try to refine)

replaces the goal by the first proof that it finds by recursively trying to apply all rules maximally int times. This is the **automatic proof search method** of PESCA, based on brute force but always terminating. With certain calculi, such as G4ip, this method always finds a proof after a predictable number of steps. With predicate calculus rules that require instantiations, the method usually fails.

`n` sequent (new)

replaces the current proof by a proof consisting of the sequent that is its open subgoal number 1. The old proof cannot be restored. But the work is not completely lost: one can save the history of the session by the command `h` and then follow the same steps to construct again whatever was constructed in the same session.

`u` subtree (undo)

replaces the subtree by a goal consisting of the conclusion of the subtree. Subtrees are identified by sequences of digits in the same way as subgoals.

`s` (show subgoals)

shows all open subgoals in the format 'goal `=` sequent'. If the system responds by showing nothing, the current proof is complete.

`a` goal (applicable rules)

shows all refinement commands applicable to the goal. To each command, the corresponding permutation of the goal sequent is appended, separated by `--`.

`c` calculus (change calculus)

changes the current calculus. The help command `?` shows available calculi, and a set of them is also shown in the Appendix. As a calculus is just a set of rules, calculi can be unioned by the operation `+`. Thus the command `c G3c + Geq` selects classical predicate calculus with equality.

`x` file (read axioms)

reads a file with nonlogical axioms, parses it into rules, adds the rules into the current calculus, and writes the rules into the file `myaxioms.tex`. The command also makes a system call to perform the LaTeX processing and show the xdvi imade on the background. The theory of nonlogical axioms is explained in chapter 6 of Negri & von Plato (1999). The syntax of axiom files is explained in Section 4.4 below.

`l` [file] (print proof in a LaTeX file)

prints the current proof in LaTeX format in the indicated file. If no file name is given, the name is `myproof.tex`. Old files are overwritten without warning. To process the LaTeX file, the style file `proof.sty` (Tatsuta 1990) is needed. It is distributed together with PESCA. The command also makes a system call to perform the LaTeX processing and show the xdvi imade on the background.

`d` [file] (print proof in natural deduction)

prints the current proof in natural deduction LaTeX format in the indicated file. If no file name is given, the name is `mydeduction.tex`. Old files are overwritten without warning. To process the LaTeX file, the style file `proof.sty` (Tatsuta 1990) is needed. It is distributed together with PESCA. The command also makes a system call to perform the LaTeX processing and show the xdvi imade on the background. The command only works for the calculi G3i and G3ip.

`h` [file] (print history in a file)

prints the sequence of command lines entered during the session into file. The `h` command line itself is not included. The default file name is `myhistory.txt`. Old files are overwritten without warning.

`?` (help)

shows a synopsis of available commands.

`m` (manual)

runs LaTeX on the manual file `manual.tex` and shows the xdvi image on the background.
shows a synopsis of available commands.

`q` (quit)

terminates the PESCA session. The sequence of command lines entered during the session (except the `q` itself) are written into the file `myhistory.txt`. Old files are overwritten without warning.

4 The syntax of sequents, formulae, and axioms

The only occasion in which the user of PESCA has to type in a sequent is as an argument of a refine (`r`) command. In addition, terms must be typed in instantiation (`i`) commands. Proofs and inference rules need never be typed. If the user wants to enter his own nonlogical axioms from an axiom file, he will have to follow a limited syntax of formulae. This section will explain the syntax of sequents, formulae, and terms that is recognized by the PESCA parser. The expressions that PESCA prints out may differ from this syntax; the LaTeX output certainly does.

Junk, that is, spaces, tabulators, and newlines, may appear between any items, but not, of course, inside identifiers. In some cases, the presence of junk is necessary for the parser to tell where one identifier ends and the next one begins.

The syntax is given in Backus-Naur style, using | for alternatives, $^{\circ}$ for optionality, and $\{\ \}^{\mathtt{t}}$ for lists separated by the token `t`. The last column of each table shows an example string satisfying the definition of that row.

4.1 Sequents and formulae

A sequent is a pair of (possibly empty) contexts separated by the double arrow `=>`. Contexts are lists of formulae separated by commas.

Sequent	::=	Context$^{\circ}$ `=>` Context$^{\circ}$	`A, B => B, C, A`
Context	::=	{ Formula }`,`	`A & B, C`

Formulae themselves are those of predicate calculus: conjunctions, disjunctions, implications, negations, and universal and existential quantifications. Logically simple formulae fall into three categories: predications, schematic formulae, and the absurdity $\bot$, of which the last-mentioned one is not counted as an atomic formula.

Formula	::=	Atom	`Apt(a,b)`
	|	Formula `&` Formula	`A & B`
	|	Formula `v` Formula	`A v B`
	|	Formula `->` Formula	`A -> B`
	|	`~` Formula	`~A`
	|	`_|_`	`_|_`
	|	`(/A` Var `)` Formula	`(/Ax)B(x)`
	|	`(/E` Var `)` Formula	`(/Ex)B(x)`
Atom	::=	Scheme ArgList$^{\circ}$	`A(x,y,z)`
	|	Predicate ArgList$^{\circ}$	`Apt(x,y)`
	|	Term InfixPred Term	`2 < 3`

Formulae are grouped in accordance with their **precedences**, which are the usual ones, from stronger to weaker: simple and quantified and negation, conjunction and disjunction, implication. These precedences are overridden by the use of parentheses.

4.2 Argument lists and terms

Argument lists are lists of terms in parentheses, and terms are either parametres or variables or constants, the last-mentioned oprionally followed by an argument list.

ArgList	::=	({ Term }')	`(x,y,z)`
Term	::=	Variable	`x`
	|	Parametre	`?c`
	|	Constant ArgumentList^{o}	`pi`
	|	Term InfixFun Term	`x + 2`

A notion of precedence would make sense for infix terms, but there is so far none in PESCA. Nested infix terms should thus be disambiguated by parentheses.

4.3 Identifiers

In the syntactic rules above, several classes of identifiers have been presupposed. The parser of PESCA keeps them disjoint, to avoid the need of lookup in a dictionary. Thus they are different combinations of capital and small letters, digits, and other characters. No junk (e.g. space) is tolerated inside an identifier.

Scheme	::=	{ Capital }	`A`
Predicate	::=	Capital { Small }	`Apt`
Variable	::=	Small { ' }	`x''`
Parametre	::=	? Variable	`?c''`
Constant	::=	Small { Small }	`pi`
	|	{ Digit }	`123`
InfixPred	::=	= | < | > | #	`>`
	|	\ { Letter }	`\leq`
InfixFun	::=	+ | - | *	`+`
	|	\ { Letter }	`\circ`

Notice that the definition of infix predicates and functions allows the use of the mathematical symbolism of LaTeX. Also notice that, since there is no distinction in LaTeX symbols between predicates and individual functions, parentheses are needed if both are present. For instance, the LaTeX/PESCA formula

```
c \leq a \wedge b
```

is ambiguous between whether `\wedge` or `\leq` is the predicate, and the normal way of reading it is achieved by

```
c \leq (a \wedge b)
```

Since the intended structure is clear for the human reader, the LaTeX output is still

$$c \leq a \wedge b$$

4.4 Axioms

As shown in section 6 of Negri & von Plato (1999), certain kinds of formulae can be interpreted as sequent calculus rules that preserve the possibility of cut elimination and are hence favourable for proof search. These formulae are implications with conjunctions of atoms on their left-hand sides, and disjunctions of atoms on their right-hand sides. Either side can be empty. An empty left-hand side is represented by the omission of

the implication sign. An empty right-hand side is represented by the absurdity $\bot$. Alternatively, the negation of a left-hand side is interpreted as its impication of absurdity; parentheses are not strictly necessary, since the whole conjunction must be in the scope of the negation.

Axiom	::=	LeftSide `->` RightSide	`x=y -> y=x`
	|	RightSide	`0=0`
	|	`~` LeftSide	`~ x#x`
LeftSide	::=	{ Atom }$^{\texttt{\&}}$	`x<y & y<z`
RightSide	::=	{ Atom }$^{\texttt{v}}$	`x#z v y#z`
	|	`_|_`	`_|_`

Atoms are like in Section 4.1 above, with two exceptions:

> Schematic letters are not permitted in atoms, but only constant predicates.
>
> Parametres (variables prefixed by `?`) are not recognized in atoms. Instead, all those variables that occur on the right-hand-side but not on the left-hand-side are treated as parametres.

In axiom files, each axiom is preceded by an identifier to be used as a rule label. This identifier can contain any characters except junk. The second example of the following section shows an example axiom file.

An axiom file may contain *comments*, which are lines beginning with `--`. Comments are ignored, except the first one, which has a special effect: the part of the file preceding that comment is passed to LaTeX as it is, permitting the user to define macros for notation. Thus an axiom file must contain at least one comment; otherwise it is interpreted as empty.

4.5 Examples of sequents and axioms

Here are some well-formed PESCA sequents and their LaTeX printouts:

```
=>
```

$\Rightarrow$

```
A & B => A, B
```

$A\&B \Rightarrow A, B$

```
=> ~~(A & B) -> ~~A & ~~B
```

$\Rightarrow\sim\sim (A\&B) \supset \sim\sim A\& \sim\sim B$

```
(/Ey)(/Ax)C(x,y) => (/Ax)(/Ey)C(x,y)
```

$(\exists y)(\forall x)C(x,y) \Rightarrow (\forall x)(\exists y)C(x,y)$

```
=> (/Ax)(x > 0 -> (/Ey)(y \leq (2 \div x) & y \geq 0))
```

$\Rightarrow (\forall x)(x > 0 \supset (\exists y)(y \leq 2 \div x \& y \geq 0))$

The second example is an axiom file and the corresponding rules in LaTeX output: some axioms of the theory of lattices from section 6.6 of Negri & von Plato (1999):

```
%% this time no macros for lattice theory
-- axioms of lattice theory
Mtl    (a \wedge b) \leq a
Mtr    (a \wedge b) \leq b
Jnl    a \leq (a \vee b)
Jnr    b \leq (a \vee b)
Unimt  c \leq a & c \leq b -> c \leq (a \wedge b)
Unijn  a \leq c & b \leq c -> (a \vee b) \leq c
Ref    a \leq a
Trans  a \leq b & b \leq c -> a \leq c
```

$$\frac{a \wedge b \leq a, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ Mtl$$

$$\frac{a \wedge b \leq b, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ Mtr$$

$$\frac{a \leq a \vee b, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ Jnl$$

$$\frac{b \leq a \vee b, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ Jnr$$

$$\frac{c \leq a \wedge b, c \leq a, c \leq b, \Gamma \Rightarrow \Delta}{c \leq a, c \leq b, \Gamma \Rightarrow \Delta}\ Unimt$$

$$\frac{a \vee b \leq c, a \leq c, b \leq c, \Gamma \Rightarrow \Delta}{a \leq c, b \leq c, \Gamma \Rightarrow \Delta}\ Unijn$$

$$\frac{a \leq a, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ Ref$$

$$\frac{a \leq c, a \leq b, b \leq c, \Gamma \Rightarrow \Delta}{a \leq b, b \leq c, \Gamma \Rightarrow \Delta}\ Trans$$

5 Usage and compiling

5.1 Running PESCA

The executable PESCA program consists of a shell script, named `pesca`. If this file exists in one's path, one can start a PESCA session by just typing the name of the file and pressing enter in the shell:

```
shell$ ./pesca
```

The script calls the Haskell interpreter `hugs` in the batch mode, `runhugs`. To obtain hugs, consult the hugs home page mentioned in the references.

5.2 Compiling PESCA

PESCA is a small and light program, which hardly needs to be compiled to be usable. Anyone who wishes to do so, however, can use e.g. `hbc` or `ghc`. The `Main` module lies in the file `Editor.hs`.

6 The implementation

This section will go through the basic data structures and algorithms of PESCA, and thereby also explain its main theoretical ideas. The presentation is given in Haskell code. Some of it may be readable to anyone familiar with type theory or some functional programming language; the Haskell home page mentioned in references gives more information.

6.1 The structure of the source code

Here is a wc dump of the PESCA source files. Each file contains a module of the same name. The sizes of the modules may still change a bit, but the total code will not greatly differ from the indicated 1396 lines, which includes comments and empty lines.

```
 178     923    5967 Axioms.hs      -- operations on nonlogical axioms
 193     966    7249 Calculi.hs     -- predefined calculi
 198     968    7582 Editor.hs      -- dialogue and command language
 103     598    4410 Interaction.hs -- refinement, proof search, etc
 169     993    6906 Natural.hs     -- natural deduction
  82     429    2613 PSequent.hs    -- parsing
 112     616    3756 PrSequent.hs   -- printing
 161     939    4249 PrelSequent.hs -- prelude : auxiliary functions
 200     980    6457 Sequent.hs     -- basic data types and functions
1396    7412   49189 total
```

6.2 The module `Sequent`—basic data types and functions

This module defines the **abstract syntax** of sequents, formulae, singular terms, proofs, etc. On the level of abstract syntax, all these expressions are Haskell data objects, which could also be called **syntax trees**. In user input and PESCA output, syntax trees are represented by **strings** of characters. The relation between syntax trees and strings is defined by the **parsing** and **printing** functions explained in the next section.

With the exception of the communication with the user, PESCA always operates on syntax trees and not on strings. There are lots of operations defined by pattern matching on the basic types of syntax trees: `Sequent`, `Formula`, `Term`, `Proof`, `AbsRule`. The operations defined in the module `Sequent` include substitution and tree-node numbering. Most of this material is unsurprising. But there are a couple of features that identify PESCA as an editor precisely for shared-context multiset calculi.

The definition that expresses the top-down determinacy of PESCA (cf. Section 1) is the definition of abstract inference rules:

```
type AbsRule = Sequent -> Maybe [Either Sequent Ident]
```

A rule is a function that takes a conclusion sequent as its argument and returns either a list of premises or a failure. Returning an empty list of premises means that the proof of the conclusion is complete. The premises can be either sequents or parametre identifiers. (We already argued, in Section 2.2 above, that parametres are really premises.)

Sequents are treated as pairs of multisets of formulae. The definition of the type of sequents is as pairs of lists rather than multisets: the multiset aspect is implicit in

various functions that consider variants of these lists obtained by making one formula in turn the active formula. In most cases, this is enough, and it is not necessary to consider all permutations.

Some calculi restrict the succedents of sequents to be single formulae. PESCA does not use a distinct type for these calculi: it simply is the property of certain calculi, such as G3ip, that the rules never require nor produce multi-succedent sequents.

6.3 The modules `PSequent` and `PrSequent`—parsing and printing

Parsing follows the top-down method explained in Wadler (1985). The basic **parsing combinators** are defined in the module `PrelSequent`. We have chosen to define parsing independently of an environment that could, for instance, give lists of constants and predicates: because of this, we have had to distinguish between different types of identifiers, as explained in Section 4.3 above. In some crucial places, such as in parsing formulae with infix connectives, the parser is optimized by left factorization and thus does not strictly follow the abstract syntax.

The printing functions first produce terminal output for PESCA sessions. A simple string transformer `makeLatex` produces LaTeX code from this, except in one case: proof trees. The proof trees printed on terminal do not always represent well the structure of the proof, and they become practically unreadable when the proofs grow. LaTeX proof trees are printed separately, directly from syntax trees.

6.4 The module `Calculi`—predefined calculi

Some intuitionistic and classical calculi are defined directly as sets of abstract rules. Because abstract rules are functions, they cannot be read from separate files at runtime, but must be compiled into PESCA. We also considered a special syntax that could be used for reading calculi from files, but finally restricted this to nonlogical axioms: there is, after all, so much variation and irregularity in the rules, that the syntax and the operations on it become complicated, and are not guaranteed to cope with "arbitrary sequent-calculus rules".

At least when using PESCA in the interpreter hugs, it should be easy to experiment with new calculi by editing the file `Calculi.hs`. No other files need be edited.

6.5 The module `Interaction`—refinement and proof search

This module defines the central proof search functions: refinement, instantiation, undoing, and automatic search. All of these operations are based on the underlying operation of replacing a subtree by another tree,

```
replace :: Proof -> [Int] -> Proof -> Proof
```

where lists of integers denote subtrees. Another underlying operation is to list all those rules of a clculus that apply to a given sequent,

```
applicableRules ::
  AbsCalculus -> Sequent -> [((Ident,AbsRule),Obligations)]
```

Automatic proving calls this function in performing top-down proof search, following the methods of Wadler (1985) just like parsing.

6.6 The module `Editor`—dialogue and command language

This is the `Main` module, which also contains the definition of the command language and the parser of the latter, as well as the help message. The central function is the execution of commands,

```
exec :: Command -> (AbsCalculus,Proof) -> (Proof,String)
```

which works in an environment of a current calculus and a current proof, and possibly changes it, by calling the functions of `Interaction`.

6.7 The module `Axioms`—operations on nonlogical axioms

This module defines a format for a special kind of rules, nonlogical axioms. These axioms are just pairs of lists of atoms, which are easy read from familiar logical notation (cf. Section 4.4 above). Their translation into abstract PESCA rules is somewhat complicated because of the presence of implicit parametres.

6.8 The module `Natural`—translation into natural deduction

This module defines a translation from sequent calculus proofs to natural deduction in a rather *ad hoc* way, which only applies to the calculi G3i and G3ip. Systematizing and extending the translation would be a welcome contribution to PESCA!

7 Known bugs and defects

PESCA is an experimental system still under development. Contributions to this Section are thus welcome. As of February 15, 2024, the most urgent problems are the following:

> The printing of proof trees on terminal is inadequate for large branching proofs.
>
> In LaTeX printing, it would be nice to put multicharacter identifiers into `\mbox`es.
>
> The equality rules in `Calculus` are unfinished.

In the Haskell code, those lines that we find obscure or suboptimal have been suffixed by three hyphens, `---`.

References

G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1934.

HBC home page, `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`

Hugs home page, `http://www.haskell.org/hugs/`

S. Kleene, *Introductin to Metamathematics*, North-Holland, Amsterdam, 1952.

L. Magnusson, *The implementation of ALF—a proof editor based on Martin-Löf's monomorphic type theory with explicit substitution*, PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, 1994.

P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984.

S. Negri & J. von Plato, "Structural Proof Theory", manuscript, University of Helsinki, 1999.

PESCA home page, to appear.

K. Schütte, *Proof Theory*, Springer, 1977.

M. Tatsuta, `proof.sty` (Proof Figure Macros), LaTeX style file, 1990.

A. Troelstra & H. Schwichtenberg, *Basic Proof Theory*, Cambridge University Press, 1996.

P. Wadler, "How to replace failure by a list of successes", *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pp. 113–128, *Lecture Notes in Computer Science* vol. 201. Springer, Heidelberg, 1985.

Appendix: Some calculi

The rules are from Negri & von Plato (1999).

G*n*ip

$$P, \Gamma \Rightarrow P \qquad \frac{}{\bot, \Gamma \Rightarrow C}\ L\bot \qquad \frac{A, \Gamma \Rightarrow B}{\Gamma \Rightarrow A \supset B}\ R\supset$$

$$\frac{A, B, \Gamma \Rightarrow C}{A\&B, \Gamma \Rightarrow C}\ L\& \qquad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A\&B}\ R\&$$

$$\frac{A, \Gamma \Rightarrow C \quad B, \Gamma \Rightarrow C}{A \vee B, \Gamma \Rightarrow C}\ L\vee \qquad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B}\ R\vee 1 \qquad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B}\ R\vee 2$$

G3ip = G*n*ip +

$$\frac{A \supset B, \Gamma \Rightarrow A \quad B, \Gamma \Rightarrow C}{A \supset B, \Gamma \Rightarrow C}\ L\supset$$

G4ip = G*n*ip +

$$\frac{P, B, \Gamma \Rightarrow E}{P, P \supset B, \Gamma \Rightarrow E}\ L0\supset \qquad \frac{C \supset (D \supset B), \Gamma \Rightarrow E}{C\&D \supset B, \Gamma \Rightarrow E}\ L\&\supset$$

$$\frac{C \supset B, D \supset B, \Gamma \Rightarrow E}{C \vee D \supset B, \Gamma \Rightarrow E}\ L\vee\supset \qquad \frac{C, D \supset B, \Gamma \Rightarrow D \quad B, \Gamma \Rightarrow E}{(C \supset D) \supset B, \Gamma \Rightarrow E}\ L\supset\supset$$

G3cp

$$P, \Gamma \Rightarrow \Delta, P \qquad \frac{}{\bot, \Gamma \Rightarrow \Delta}\ L\bot$$

$$\frac{A, B, \Gamma \Rightarrow \Delta}{A\&B, \Gamma \Rightarrow \Delta}\ L\& \qquad \frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A\&B}\ R\&$$

$$\frac{A, \Gamma \Rightarrow \Delta \quad B, \Gamma \Rightarrow \Delta}{A \vee B, \Gamma \Rightarrow \Delta}\ L\vee \qquad \frac{\Gamma \Rightarrow \Delta, A, B}{\Gamma \Rightarrow \Delta, A \vee B}\ R\vee$$

$$\frac{\Gamma \Rightarrow \Delta, A \quad B, \Gamma \Rightarrow \Delta}{A \supset B, \Gamma \Rightarrow \Delta}\ L\supset \qquad \frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \supset B}\ R\supset$$

G3i = G3ip +

$$\frac{A(t/x), \forall x A, \Gamma \Rightarrow C}{\forall x A, \Gamma \Rightarrow C}\ L\forall \qquad \frac{\Gamma \Rightarrow A(y/x)}{\Gamma \Rightarrow \forall x A}\ R\forall \qquad \frac{A(y/x), \Gamma \Rightarrow C}{\exists x A, \Gamma \Rightarrow C}\ L\exists \qquad \frac{\Gamma \Rightarrow A(t/x)}{\Gamma \Rightarrow \exists x A}\ R\exists$$

G3c = G3cp +

$$\frac{A(t/x), \forall x A, \Gamma \Rightarrow \Delta}{\forall x A, \Gamma \Rightarrow \Delta}\ L\forall \qquad \frac{\Gamma \Rightarrow \Delta, A(y/x)}{\Gamma \Rightarrow \Delta, \forall x A}\ R\forall$$

$$\frac{A(y/x), \Gamma \Rightarrow \Delta}{\exists x A, \Gamma \Rightarrow \Delta}\ L\exists \qquad \frac{\Gamma \Rightarrow \Delta, \exists x A, A(t/x)}{\Gamma \Rightarrow \Delta, \exists x A}\ R\exists$$