
ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 93 (2004) 161–182
www.elsevier.com/locate/entcs

Deduction and Presentation in ρLog

Mircea Marin[1,3]

Johann Radon Institute for Computational and Applied Mathematics
Austrian Academy of Sciences
Linz, Austria

Florina Piroi[2,4]

Research Institute for Symbolic Computation
Johannes Kepler University
Hagenberg, Austria

Abstract

We describe the deductive and proof presentation capabilities of a rule-based system implemented in MATHEMATICA. The system can compute proof objects, which are internal representations of deduction derivations which respect a specification given by the user. It can also visualize such deductions in human readable format, at various levels of detail. The presentation of the computed proof objects is done in a natural-language style which is derived and simplified for our needs from the proof presentation styles of THEOREMA.

Keywords: Rule-based deduction, proof presentation, rewriting.

1 Introduction

One of the main problems of mathematical knowledge management is to structure the knowledge from various mathematical sources in a way which is suitable for testing and deriving new mathematical results. A promising approach

[1] Mircea Marin has been supported by the Austrian Academy of Sciences.
[2] Florina Piroi has been supported by the Austrian Science Foundation FWF, under the SFB grant F1302.
[3] Email: Mircea.Marin@oeaw.ac.at
[4] Email: fpiroi@risc.uni-linz.ac.at

1571-0661 © 2004 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2003.12.033

to solve this problem is to express the knowledge as collections of transformation rules, and to control the lookup for new results via user-defined strategies. The derivation of a new result is witnessed by a sequence of transformation steps, where each step is an instance of a transformation rule. The purpose of a strategy is to guide the search of such derivations by imposing a regular structure on the sequence of rule applications which constitute them.

The success of this approach relies on the availability of a system with programming primitives for rules and strategies. For this purpose, we have developed a programming system called ρLOG. ρLOG is a renamed version of the rule-based programming system FUNLOG [7,8]. We did this in order to avoid confusing it with `FUNLOG` [13], a programming system of the eighties. ρLOG is a suitable environment for specifying and implementing deduction systems in a language based on rules whose application is controlled by user-defined strategies. More precisely, ρLOG allows:

- to program non-deterministic computations by using the advanced features of MATHEMATICA [15], like matching with sequence patterns, and access to state-of-the-art libraries of methods for symbolic and numeric computation;
- to program rules l whose reduction relation $\rightarrow_l$ can be defined, possibly recursively, in terms of already defined reduction relations $\rightarrow_{l_1}, \ldots, \rightarrow_{l_n}$;
- to enquire whether, for a given expression E and rule l, there exists an expression x such that the derivation relation $E \rightarrow_l x$ holds. We denote such a query by $\exists^{?} x : E \rightarrow_l x$.
- to generate proof objects which encode deductions that decide the validity of a formula $\exists x : E \rightarrow_l x$. The system has the capability to visualize such deductions in human readable format, at various levels of detail.

We decided to implement ρLOG in MATHEMATICA mainly because this computer algebra system has advanced features for pattern matching and for computing with transformation rules. These features provide good support for implementing a full-fledged rule-based system. MATHEMATICA also offers a very good support for symbolic and numeric computations. Another reason is that rule-based programming, as envisioned by us, could be used efficiently to implement provers, solvers, and simplifiers which could, then, be integrated in the THEOREMA framework [3]. Since THEOREMA is implemented in MATHEMATICA, a MATHEMATICA implementation of a powerful rule-based system may become a convenient programming tool for THEOREMA developers.

We refer to [10] for a complete description of the programming capabilities of our system, and to [15] for a description of the pattern matching constructs of MATHEMATICA. The following example shows a problem which can be reduced to a query for ρLOG, and how we can find a solution with ρLOG.

Example 1.1 Consider the functions $f_1 : (-\infty, 0) \to \mathbb{R}$, $f_2 : (-\infty, 1) \to \mathbb{R}$, $g : (0, \infty) \to \mathbb{R}$ defined by $f_1(x) = x + 7$, $f_2(x) = x + 4$, $g(x) = x/2$. Consider, now, the non-deterministic operation $f : (-\infty, 1) \to \mathbb{R}$ defined by

$$f(x) = \begin{cases} f_1(x) \text{ if } x < 0, \\ f_2(x) \text{ if } x < 1. \end{cases}$$

We want to program a rule which encodes the partially defined and non-deterministic computation of $g(f(x))$ for all $x \in \mathbb{R}$.

First, we encode the functions f_1, f_2 and g as ρLOG transformation rules `"f1"`, `"f2"` and `"g"`:

$$\texttt{DeclareRule}[x\_\texttt{Real}/; (x < 0) :\rightarrow x + 7, \texttt{"f1"}];$$
$$\texttt{DeclareRule}[x\_\texttt{Real}/; (x < 1) :\rightarrow x + 4, \texttt{"f2"}];$$
$$\texttt{DeclareRule}[x\_\texttt{Real}/; (x > 0) :\rightarrow x/2, \texttt{"g"}];$$

Each call `DeclareRule`$[patt :\rightarrow rhs, l]$ declares a new rule $patt :\rightarrow rhs$ which is named l for later reference. The construct x`_Real` specifies the pattern variable x which stands for a real value. All patterns have, in this example, side conditions which impose additional constraints on the values allowed for x. For instance, rule `"f1"` requires the value of x to be negative ($x < 0$).

We convene to write $\rightarrow_l$ for the reduction relation associated with a rule l. For example, we have $0.5 \rightarrow_{\texttt{"f2"}} 4.5$ because 0.5 is a real number smaller than 1 which can be replaced by `"f2"` with $0.5 + 4 = 4.5$.

The call

$$\texttt{SetAlias}[\texttt{"f1"} \mid \texttt{"f2"}, \texttt{"f"}];$$

declares the rule `"f"` whose reduction relation coincides with $\rightarrow_{\texttt{"f1"}} \cup \rightarrow_{\texttt{"f2"}}$ and, therefore, it encodes the computation of f.

The call

$$\texttt{SetAlias}[\texttt{"f"} \circ \texttt{"g"}, \texttt{"fg"}];$$

declares the rule `"fg"` whose associated reduction relation $\rightarrow_{\texttt{"fg"}}$ coincides with the composition $\rightarrow_{\texttt{"f"}} \circ \rightarrow_{\texttt{"g"}}$ of the relations $\rightarrow_{\texttt{"f"}}$ and $\rightarrow_{\texttt{"g"}}$.[5] It is obvious that $x \rightarrow_{\texttt{"fg"}} r$ iff r is a possible result of the computation $g(f(x))$. Thus, the rule `"fg"` encodes the computation which we look for.

The call `ApplyRule`$[E, \texttt{"fg"}]$ enquires the system to decide whether, for a given expression E, the operation $g(f(E))$ is defined or not. If the operation is defined then the call yields a possible result, otherwise it returns E

[5] Note that the rule composition `"f"` $\circ$ `"g"` does not correspond to the composition $f \circ g$ as understood in mathematics, but to the composition $g \circ f$.

unevaluated. For instance, the call:

$$\texttt{ApplyRule}[-7.1, \texttt{"fg"}]$$

returns -7.1 because $\nexists x, y:\ (-7.1 \rightarrow_{\texttt{"f"}} y) \wedge (y \rightarrow_{\texttt{"g"}} x)$, and

$$\texttt{ApplyRule}[0.2, \texttt{"fg"}]$$

returns 3.6 because $(0.2 \rightarrow_{\texttt{"f"}} 7.2) \wedge (7.2 \rightarrow_{\texttt{"g"}} 3.6)$. □

The rest of the paper is structured as follows. Section 2 describes the main programming principles and constructs of ρLog. In Section 3 we describe the general structure of deduction derivations in ρLog. Section 4 is about proof objects, which constitute the internal representation of deduction derivations. Section 5 gives an account to the methods provided by ρLog to manipulate proof objects, and to view the encoded rule-based proofs in a human-readable format. Section 6 concludes.

2 Programming Principles

In this section we introduce the concepts of ρLog rule, ρ-valid expression, and describe how rules can be composed and applied.

2.1 Rules and Expressions

Rules are the main programming concept of ρLog. They specify partially defined and non-deterministic computations. Formally, a *rule* is an expression $l :: patt :\rightarrow rhs$ where l is the rule name, *patt* is a pattern expression and *rhs* specifies a possibly non-deterministic computation in terms of the variables which occur in *patt*.

The main usage of rules is to apply them on expressions. Any expression which can be represented in the language of Mathematica [15] is a valid input expression for ρLog. Moreover, an expression may contain a distinguished number of selected subexpressions. The current implementation of ρLog is capable to apply rules on expressions with a (possibly empty) sequence of selected subexpressions $E_1, \ldots, E_n$ such that E_{i+1} is a subexpression of E_i whenever $1 \leq i < n$. Such expressions are called *ρ-valid*.

Thus, the expressions which are meaningful for the current implementation of ρLog have at most one innermost selected subexpression.

When we want to emphasize that a ρ-valid expression E has the innermost selected subexpression E', then we will write $E[\![E']\!]$. We may also write $E[\![E]\!]$ when E has no selected subexpressions, and $E[\![E'/E'']\!]$ for the expression

obtained from $E[\![E']\!]$ by replacing the distinguished subexpression E' with the unselected expression E''.

The notation $E[\![E]\!]$ is ambiguous: either E has no selected subexpressions or E is selected itself. We allow this ambiguity because it is harmless in our framework. In illustrative examples we will simply underline the selected subexpressions of an expression as in the following:

Example 2.1 Take the expressions

$$E_1 = \mathtt{f}[\mathtt{f}[x, \mathtt{e}], \mathtt{e}],\ E_2 = \underline{\underline{\mathtt{f}[x, \mathtt{e}]}},\ E_3 = \underline{\mathtt{f}[\underline{\mathtt{f}[\underline{\mathtt{f}[x, \mathtt{e}]}, x]}, y]},\ E_4 = \mathtt{f}[\underline{\mathtt{f}[x, \mathtt{e}]}, \underline{\mathtt{f}[\mathtt{e}, x]}].$$

E_1, E_2, E_3 are ρ-valid, whereas E_4 is not ρ-valid because it has two innermost selected subexpressions.

We can write $E_1[\![E_1]\!]$, $E_2[\![E_2]\!]$ and $E_3[\![f[x, \mathtt{e}]]\!]$ to give information about the innermost selected subexpressions (if any) of these expressions. We have

$$E_1[\![E_1/z]\!] = z,\ E_2[\![E_2/z]\!] = \underline{z},\ E_3[\![\mathtt{f}[x, \mathtt{e}]/z]\!] = \underline{\mathtt{f}[\mathtt{f}[z, x], y]}.$$ □

In the sequel we will assume implicitly that E, E', E'', E_1, E_2, ... denote ρ-valid expressions, and l, l', l_1, l_2, ... denote ρLOG rules.

2.2 Rule Application

There are two procedure calls which trigger the application of a rule l on a ρ-valid expression E: `ApplyRule`$[E, l]$ and `ApplyRuleList`$[E, l]$. The first call attempts to apply an already declared rule $l :: patt :\rightarrow rhs$ on E by looking at the innermost selected subexpression of E, if any. `ApplyRule`$[E, l]$ computes

- $E_1 = E[\![E'/E'']\!]$ if we have $E[\![E']\!]$ and it is possible to identify a substitution θ for which $\theta(patt) = E'$ and the computation $\theta(rhs)$ can be carried out to produce a result E''. In this case we write $E \rightarrow_l E_1$.
- E, otherwise. In this case we write $E \not\rightarrow_l$.

We emphasize that the binary relation $\rightarrow_l$ may not be one-to-one because either (1) the matching substitution θ for which $\theta(patt) = E'$ is not unique, or (2) there may be more than one possibilities to evaluate the instance $\theta(rhs)$ to a result (as in Example 1.1). These two sources of non-determinism are discussed in more detail in [10].

The call `ApplyRuleList`$[E, l]$ will compute the possibly empty list of ρ-valid expressions $\{E' \mid E \rightarrow_l E'\}$.

2.3 Combining Rules

The programming primitives of ρLOG are the *basic* rules. These are named MATHEMATICA transformation rules declared via a call

$$\texttt{DeclareRule}[patt :\rightarrow rhs, l]$$

where $patt :\rightarrow rhs$ is a MATHEMATICA specification of a transformation rule [15] and l is a string which identifies uniquely the newly declared rule.

Rules can be combined into more complex rules. ρLOG provides the following combinators to program rules:

choice: If $l_1, \ldots, l_n$ are rules then $l_1 \mid \ldots \mid l_n$ is a rule with $E_1 \rightarrow_{l_1|\ldots|l_n} E_2$ iff $E_1 \rightarrow_{l_i} E_2$ for some $1 \leq i \leq n$,

composition: If l_1, l_2 are rules then $l_1 \circ l_2$ is a rule with $E_1 \rightarrow_{l_1 \circ l_2} E_2$ iff there exists E such that $E_1 \rightarrow_{l_1} E$ and $E \rightarrow_{l_2} E_2$,

reflexive-transitive closures: If l_1, l_2 are rules then $\texttt{Repeat}[l_1, l_2]$ is a rule with $E_1 \rightarrow_{\texttt{Repeat}[l_1,l_2]} E_2$ iff there exists E such that $E_1 \rightarrow_{l_1}^* E$ and $E \rightarrow_{l_2} E_2$. We write $\rightarrow_{l_1}^*$ for the reflexive-transitive closure of $\rightarrow_{l_1}$.

Similarly, $\texttt{Until}[l_2, l_1]$ is a rule with the same denotational semantics as $\texttt{Repeat}[l_1, l_2]$; the only difference is that $\texttt{Repeat}[l_1, l_2]$ applies l_1 as many times as possible before applying l_2 , whereas $\texttt{Until}[l_2, l_1]$ applies l_1 repeatedly until l_2 is applicable.

normal form: If l is a rule then $\texttt{NF}[l]$ is a rule with $E_1 \rightarrow_{\texttt{NF}[l]} E_2$ iff $E_1 \rightarrow_l^* E_2$ and $E_2 \not\rightarrow_l$. Also, $E \rightarrow_{\texttt{NFQ}[l]} E$ iff $E \not\rightarrow_l$.

rewrite rule: If l_1 is a rule then we can introduce the rule l with $E \rightarrow_l E_1$ iff there exists a subexpression E' of E such that $E' \rightarrow_{l_1} E''$ and $E_1 = E[\![E'/E'']\!]$. l is called the *rewrite rule* induced by l.

The operational semantics of rewriting depends on the choice of the subexpression on which l_1 can act. Introducing the rule together with its choice strategy can be done by a `RWRule[ ]` call. (See the following section.)

To make the combination of rules easier and more intuitive, we can define aliases via `SetAlias[ ]` calls, like we did in Example 1.1.

2.4 Rewriting

We continue now with a description of how rewriting is implemented in ρLOG. A rewrite step can be regarded as a composition of two steps: one which selects the subexpression to be rewritten, followed by another one which rewrites it. To achieve suitable selection strategies for rewriting, we have designed two kinds of rules:

(i) the basic rule `"Rw"` whose relation is $E[\![E']\!] \rightarrow_{\texttt{"Rw"}} E[\![E'/\underline{E'}]\!]$. This means that we add a selection to the innermost selected subexpression of E. If E has no selected subexpressions then we add a selection to it as a whole.

(ii) selection shift rules, which can shift the innermost selection on a proper subexpression of the innermost selected subexpression. Selection shift rules are important for navigating through the subexpressions of an expression, until we reach one which can be rewritten.

Formally, a selection shift rule l is characterized by a computable function $\texttt{shift}_l$ and a rule $\texttt{r}_l$, such that

$$E_1 \rightarrow_l E_2 \text{ iff } \exists E' \in \texttt{shift}_l[E_1] \text{ such that } E' \rightarrow_{\texttt{r}_l} E_2.$$

It is assumed that $\texttt{shift}_l[E_1]$ is a finite list of values for every input E_1.

ρLOG has only one built-in selection shift rule, $\texttt{SEL}[l]$, whose applicative behavior is defined as a side-effect of a call

$$\texttt{RWRule}[\, l_1,\ l,\ \texttt{Traversal} \rightarrow val,\ \texttt{Prohibit} \rightarrow \{f_1, \ldots, f_n\}] \qquad (1)$$

with $val \in \{\texttt{"LeftIn"}, \texttt{"LeftOut"}\}$ and $\{f_1, \ldots, f_n\}$ is a list of MATHEMATICA symbols. The option `Traversal` defines the choice strategy of the rewrite rule. `Traversal` $\rightarrow$ `"LeftIn"` will make the selecting process look for a rewritable subexpression of the input expression in leftmost-innermost order, while with `Traversal` $\rightarrow$ `"LeftOut"` will look in the leftmost-outermost order.

When `Prohibit` is given a list of symbols $\{f_1, \ldots, f_n\}$, the selection process will ignore the subexpressions of the expressions with the outermost symbol being one of $f_1, \ldots, f_n$. The default value of `Prohibit` is $\{\}$, i.e., any subexpression might be selected and rewriting can be performed everywhere.

The call (1), besides stating that l is a rewrite rule induced by l_1, declares the selection shift rule $\texttt{SEL}[l]$ to be associated with the computable function $\texttt{shift}_{\texttt{SEL}[l]}$ defined by

$$\texttt{shift}_{\texttt{SEL}[l]}[E[\![E']\!]] = \begin{cases} \{E'_1, \ldots, E'_m\} & \text{if } E' = f[E_1, \ldots, E_m] \text{ with} \\ & f \notin \{f_1, \ldots, f_n\}, \text{ and} \\ & E'_i = E[\![E'/f[E_1, \ldots, \underline{E_i}, \ldots, E_m]]\!] \\ & \text{for all } i \in \{1, \ldots, m\}, \\ \{\,\} & \text{otherwise} \end{cases}$$

and with the rule

$$\texttt{r}_{\texttt{SEL}[l]} = \begin{cases} l_1 \mid \texttt{SEL}[l] \text{ if } val = \texttt{"LeftOut"}, \\ \texttt{SEL}[l] \mid l_1 \text{ if } val = \texttt{"LeftIn"}. \end{cases}$$

The call (1) will also add the recursive definition l = "Rw" $\circ$ $\mathtt{r}_{\mathtt{SEL}[l]}$ into the ρLOG session. Throughout this paper we will always assume that $\mathtt{r}_l$ and $\mathtt{shift}_l$ represent the rule and the computable function associated with a selection shift rule l.

Example 2.2 Consider the declarations

$$\mathtt{DeclareRule}[\mathtt{f}[x\_, \mathtt{e}] :\rightarrow x, \texttt{"N"}];$$
$$\mathtt{RWRule}[\texttt{"N"}, \texttt{"N*"}, \mathtt{Traversal} \rightarrow \texttt{"LeftOut"}];$$

and let $E = \mathtt{f}[\mathtt{f}[x, \mathtt{e}], y]$. Then $E \rightarrow_{\texttt{"N*"}} \mathtt{f}[x, y]$ because of the following: "N*" can be reduced to "Rw" $\circ$ $\mathtt{r}_{\mathtt{SEL}[\texttt{"N*"}]}$, and $E \rightarrow_{\texttt{"Rw"}} \underline{\mathtt{f}[\mathtt{f}[x, \mathtt{e}], y]}$; then we take $\mathtt{f}[\underline{\mathtt{f}[x, \mathtt{e}]}, y] \in \mathtt{shift}_{\mathtt{SEL}[\texttt{"N*"}]}[\underline{\mathtt{f}[\mathtt{f}[x, \mathtt{e}], y]}]$, and apply $\mathtt{r}_{\mathtt{SEL}[\texttt{"N*"}]} = \texttt{"N"} \mid \mathtt{SEL}[\texttt{"N*"}]$; we choose the alternative "N" and compute $\mathtt{f}[\underline{\mathtt{f}[x, \mathtt{e}]}, y] \rightarrow_{\texttt{"N"}} \mathtt{f}[x, y]$. □

Selection shift rules can be defined by users too, but the current way to do it is quite cumbersome. We are working on extending the actual implementation of ρLOG with a convenient definitional mechanism for selection shift rules.

We will give now an example that illustrates how ρLOG can be used as a deductive system.

Example 2.3 [Joinability test in group theory] Consider the axioms of a non-commutative group with associative operation f, right-neutral element e, and inversion operation i. These axioms can be encoded in ρLOG rules as follows:

$$\mathtt{DeclareRule}[\mathtt{f}[\mathtt{f}[x\_, y\_], z\_] :\rightarrow \mathtt{f}[x, \mathtt{f}[y, z]], \texttt{"A"}];$$
$$\mathtt{DeclareRule}[\mathtt{f}[x\_,\ \mathtt{e}] :\rightarrow x, \texttt{"N"}];$$
$$\mathtt{DeclareRule}[\mathtt{f}[x\_,\ \mathtt{i}[x\_]] :\rightarrow \mathtt{e}, \texttt{"I"}];$$

The call SetAlias["A" | "N" | "I", "G"] defines "G" as an alias for the composed rule "A" | "N" | "I". The rewrite relation induced by "G" is declared by:

$$\mathtt{RWRule}[\texttt{"G"}, \texttt{"Group"}];$$

The relation $\rightarrow_{\texttt{"Group"}}$ is terminating but not confluent, since

$$\mathtt{f}[\mathtt{f}[x, \mathtt{e}], x] \rightarrow_{\texttt{"Group"}} \mathtt{f}[x, x] \not\rightarrow_{\texttt{"Group"}};\ \ \mathtt{f}[\mathtt{f}[x, \mathtt{e}], x] \rightarrow_{\texttt{"Group"}} \mathtt{f}[x, \mathtt{f}[\mathtt{e}, x]] \not\rightarrow_{\texttt{"Group"}}$$

and $\mathtt{f}[x, x] \neq \mathtt{f}[x, \mathtt{f}[\mathtt{e}, x]]$. Therefore, checking whether two terms s and t are joinable requires a systematic search for a term u such that $s \rightarrow^*_{\texttt{"Group"}} u$ and

$t \to^*_{\texttt{"Group"}} u$. A simple way to program this joinability test in ρLOG is:

$$\begin{aligned}&\texttt{DeclareRule}[\texttt{eq}[x\_, x\_] :\to \texttt{True}, \texttt{"Eq"}];\\ &\texttt{SetAlias}[\texttt{Until}[\texttt{"Eq"}, \texttt{"Group"}], \texttt{"Join"}];\\ &\texttt{ApplyRule}[\texttt{eq}[s, t], \texttt{"Join"}]\end{aligned}$$

The last call returns `True` if s and t are joinable, and $\texttt{eq}[s, t]$ otherwise. If we want to see a justification of the result we can call

$$\texttt{ApplyRule}[\texttt{eq}[s, t], \texttt{"Join"}, \texttt{TraceStyle} \to \texttt{"Compact"}];$$

This call will generate a MATHEMATICA notebook with a human readable presentation of the underlying deduction derivation. Figure 1 presents such a notebook for $s = \texttt{f}[\texttt{f}[x, \texttt{e}], \texttt{i}[x]]$ and $t = \texttt{f}[\texttt{f}[\texttt{e},\ y],\ \texttt{i}[y]]$. □

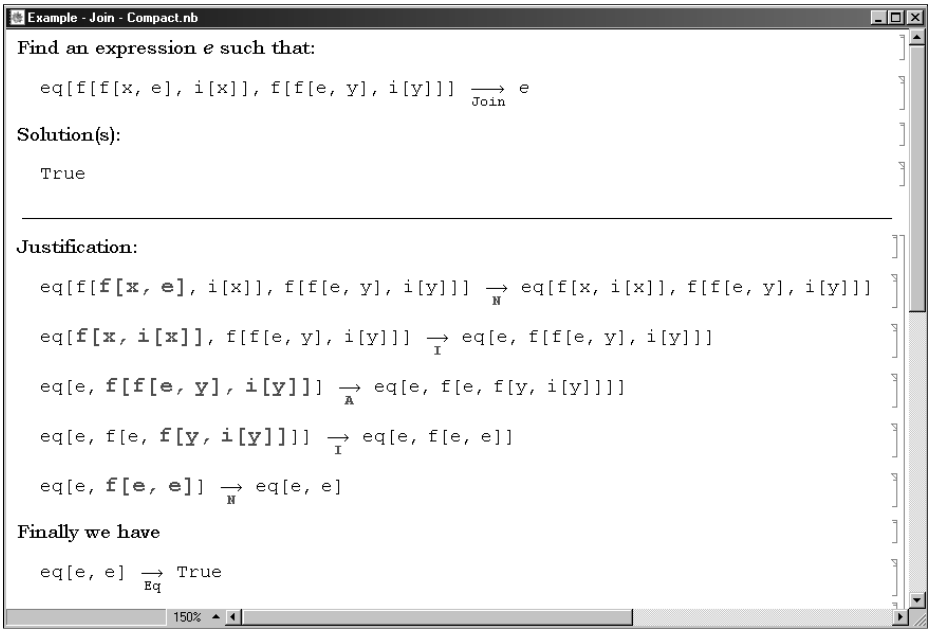

Fig. 1. `"Compact"`-style presentation of a rule-based deduction.

We summarize this section with the following observations:

(i) a rule is applied to a whole expression or to a selected subexpression of an expression,

(ii) rule application is a non-deterministic operation,

(iii) rule application is a partially defined operation,

(iv) rules can be composed into more complex rules via various combinators.

3 Deduction Trees

Intuitively, a deduction tree (*D-tree* for short) is a trace of a validity-check for the rule application query $\exists^{?}x : E \rightarrow_l x$, where the ρ-valid expression E and the rule l are given. The construction of a D-tree proceeds by successively reducing the query $\exists^{?}x : E \rightarrow_l x$ to a finite number of simpler queries. This reduction process is driven by the application of a set of inference rules.

We depict our inference rules as follows:

$$\frac{S_1 \quad \ldots \quad S_n}{\exists^{?}x : E \rightarrow_l x} \tag{2}$$

where S_i $(1 \leq i \leq n)$ are either (a) queries of the form $\exists^{?}x : E_i \rightarrow_{l_i} x$, or (b) valid reductions of the form $E_i \rightarrow_l E_i'$, or (c) expressions of the form $E_i \rightarrow_{l'} E_i' \wedge \exists^{?}x : E_i' \rightarrow_{l_1} x$ where $E_i \rightarrow_{l_2} E_i'$ is a valid reduction. We write $[\![S_i]\!]$ for the logical formula obtained by dropping the '?' superscripts from S_i. With this convention, each inference rule of our system will have the following meaning:

$$\exists x : E \rightarrow_l x \text{ iff } [\![S_1]\!] \text{ or } \ldots \text{or } [\![S_n]\!].$$

Before describing the inference rules, one by one, we would like to treat first the subject of rule reduction, used tacitly in Example 2.2, and define some auxiliary notions.

A rule l is *elementary* if it is either basic or of the form $\texttt{NFQ}[l_1]$. We remind here that `"Rw"` is a built-in basic rule.

The *reduct* of l, denoted by $\texttt{red}[l]$, is a rule whose applicative behavior coincides with that of l. This means that, for all E, E' we have

$$E \rightarrow_l E' \text{ iff } E \rightarrow_{\texttt{red}[l]} E'.$$

A rule l is *reducible* if $l \neq \texttt{red}[l]$, and *irreducible* otherwise.

Based on the definition of $\texttt{red}[\,]$ (detailed in [10]), we can define a well-founded relation $\succ$ by

$$l \succ l' \text{ iff } l' = \texttt{red}[l] \text{ and } l \neq l'.$$

Reducing a query $\exists^{?}x : E \rightarrow_l x$ to a simpler query is done by reducing l as much as possible, using the function $\texttt{red}[\,]$, until we arrive at an irreducible rule which we try to apply on E. The fact that we eventually reach an irreducible rule is guaranteed by the well-foundedness property of $\succ$.

We proceed now with describing the inference rules.

(i) If l is reducible then the corresponding inference rule is:

$$\frac{\exists^{?}x : E \rightarrow_{\mathtt{red}[l]} x}{\exists^{?}x : E \rightarrow_{l} x}$$

(ii) If l is elementary then the corresponding inference rule is:

$$\frac{E \rightarrow_{l} E_1 \quad \ldots \quad E \rightarrow_{l} E_n}{\exists^{?}x : E \rightarrow_{l} x}$$

where $E_1, \ldots, E_n$ are all expressions such that $E \rightarrow_{l} E_i$.

(iii) If l is $l_1 \mid \ldots \mid l_n$ then the corresponding inference rule is

$$\frac{\exists^{?}x : E \rightarrow_{l_1} x \quad \ldots \quad \exists^{?}x : E \rightarrow_{l_n} x}{\exists^{?}x : E \rightarrow_{l_1 \mid \ldots \mid l_n} x}$$

(iv) If l is a selection shift rule then the corresponding inference rule is:

$$\frac{\exists^{?}x : E_1 \rightarrow_{\mathtt{r}_l} x \quad \ldots \quad \exists^{?}x : E_n \rightarrow_{\mathtt{r}_l} x}{\exists^{?}x : E \rightarrow_{l} x}$$

where $\{E_1, \ldots, E_n\} = \mathtt{shift}_l[E]$.

(v) l is $l_1 \circ l_2$ where l_1 is either elementary or a selection shift rule.
If l_1 is elementary then the corresponding inference rule is:

$$\frac{(E \rightarrow_{l_1} E_1) \wedge (\exists^{?}x : E_1 \rightarrow_{l_2} x) \quad \ldots \quad (E \rightarrow_{l_1} E_n) \wedge (\exists^{?}x : E_n \rightarrow_{l_2} x)}{\exists^{?}x : E \rightarrow_{l_1 \circ l_2} x}$$

where $E_1, \ldots, E_n$ ($n \geq 0$) are all the expressions such that $E \rightarrow_{l_1} E_i$.
If l_1 is a selection shift rule then the corresponding inference rule is:

$$\frac{\exists^{?}x : E_1 \rightarrow_{\mathtt{r}_{l_1} \circ l_2} x \quad \ldots \quad \exists^{?}x : E_n \rightarrow_{\mathtt{r}_{l_1} \circ l_2} x}{\exists^{?}x : E \rightarrow_{l_1 \circ l_2} x}$$

where $\{E_1, \ldots, E_n\} = \mathtt{shift}_{l_1}[E]$.

The definitions of `red[ ]` and of the rule combinators guarantee that these inference rules cover all the possible situations for the shape of a query.

The D-tree for a query $\exists^{?}x : E \rightarrow_{l} x$, denoted by $T(E,\ l)$, is obtained by successive applications of the six inference rules defined above. It is easy to see that a D-tree $T(E,\ l)$ may be infinite for certain values of E and l. Consider as an example the expression $E = A \wedge B$ and the rule `"comm"` $:: X\_ \wedge Y\_ :\rightarrow Y \wedge X$,

then the D-tree $T(E,\ l)$ for $\exists^{?}x : E \rightarrow_{\texttt{"comm"}} x$ is the following

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A \wedge B \rightarrow_{\texttt{"comm"}} B \wedge A}}{B \wedge A \rightarrow_{\texttt{"comm"}} A \wedge B}}{A \wedge B \rightarrow_{\texttt{"comm"}} B \wedge A}}{B \wedge A \rightarrow_{\texttt{"comm"}} A \wedge B}$$

To avoid the generation of such infinite structures, we restrict the system to the construction of *partial D-trees* which are obtained by imposing a limit on the maximum number of inference applications along the branches of the tree.

Formally, the *partial D-tree of maximum depth* m, $T_m(E,\ l)$, for the query $\exists^{?}x : E \rightarrow_l x$, is defined by:

$$T_0(E,l) \quad ::= \frac{}{\exists^{?}x : E \rightarrow_l x}$$

$$T_{m+1}(E,l) ::= \frac{E \rightarrow_l E_1 \quad \ldots \quad E \rightarrow_l E_n}{\exists^{?}x : E \rightarrow_l x} \qquad l \text{ elementary}$$

$$\mid \quad \frac{T_{m+1}(E, \texttt{red}[l])}{\exists^{?}x : E \rightarrow_l x} \qquad l \text{ reducible}$$

$$\mid \quad \frac{T_m(E,l_1) \quad \ldots \quad T_m(E,l_n)}{\exists^{?}x : E \rightarrow_l x} \qquad l = l_1 \mid \ldots \mid l_n$$

$$\mid \quad \frac{(E \rightarrow_{l_1} E_1) \wedge T_m(E_1,l_2) \quad \ldots \quad (E \rightarrow_{l_1} E_k) \wedge T_m(E_k,l_2)}{\exists^{?}x : E \rightarrow_l x} \qquad \begin{array}{l} l = l_1 \circ l_2 \\ l_1 \text{ elementary} \end{array}$$

$$\mid \quad \frac{T_m(E'_1, \mathtt{r}_{l_1}) \quad \ldots \quad T_m(E'_n, \mathtt{r}_{l_1})}{\exists^{?}x : E \rightarrow_l x} \qquad \begin{array}{l} l = l_1 \\ l_1 \text{ selection} \\ \text{shift} \end{array}$$

$$\mid \quad \frac{T_m(E'_1, \mathtt{r}_{l_1} \circ l_2) \quad \ldots \quad T_m(E'_k, \mathtt{r}_{l_1} \circ l_2)}{\exists^{?}x : E \rightarrow_l x} \qquad \begin{array}{l} l = l_1 \circ l_2 \\ l_1 \text{ selection} \\ \text{shift rule} \end{array}$$

where $k, m \in \mathbb{N}$ and $\{E'_1, \ldots, E'_k\} = \texttt{shift}_{l_1}[E]$. Such a partial D-tree is obtained by successive applications of the inferences defined earlier up to m times along each branch. Inference steps of type (i) are not taken into account when computing the depth of the D-tree.

In the sequel we will omit the expression E, the rule l and the depth m from the notation of a (partial) D-tree $T_m(E,l)$ whenever we consider it irrelevant.

The following classification of partial D-trees is relevant for interpreting the data stored in their structure:

success D-tree is a partial D-tree which has at least one leaf node computed by the application of inference rule of type (ii) with $n \geq 1$.

failure D-tree is a D-tree with no leaf nodes computed by the application of inference rule of type (ii) with $n \geq 1$.

pending D-tree is a partial D-tree which is neither success D-tree nor failure D-tree.

The meaning of a partial D-tree $T_m(E, l)$ is: $\exists x : E \rightarrow_l x$ if $T_m(E, l)$ is a success D-tree; $\nexists x : E \rightarrow_l x$ if $T_m(E, l)$ is a failure D-tree; and undefined otherwise.

4 Proof Objects

The two most often invoked reasons for using proof objects in automated reasoning, and also the reasons for which ρLOG implements one, are:

- keeping a complete record of a prover's activity, and
- providing guidance to users (graphical/natural language display of proof objects).

Other reasons for having a proof object in reasoning systems are extracting proof tactics, later checking, extracting algorithms and computational methods, etc. [14]. The proof objects of ρLOG are internal representations of partial D-trees.

The proof objects of ρLOG are intended to be a compact and more explicit representation of the structure of a partial D-tree. They are defined by the following grammar:

$$
\begin{aligned}
N ::= {} & \texttt{\$SNODE}[\{E, l, E'\}] \\
\mid {} & \texttt{\$SNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
\mid {} & \texttt{\$FNODE}[\{E, \mathit{lexpr}\}] \\
\mid {} & \texttt{\$FNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
\mid {} & \texttt{\$PNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
\mid {} & \texttt{\$EPNODE}[\{E, \mathit{lexpr}\}]
\end{aligned}
$$

$$
\mathit{lexpr} ::= l \mid \{l_1, \ldots, l_n\} \mid \langle l, E, \mathit{lexpr}_1 \rangle.
$$

A success object is a proof object of the form \$SNODE[...]. Success objects are encodings of success D-trees, thus they justify the validity of a formula $\exists x : E \rightarrow_l x$ or of a formula $(E \rightarrow_{l_1} E') \wedge (\exists x : E' \rightarrow_{l_2} x)$.

A failure object is a proof object of the form \$FNODE[...]. Failure objects encode failure D-trees, and therefore they justify the validity of a formula $\nexists x : E \rightarrow_l x$ or of a logical conjunction $(E \rightarrow_{l_1} E') \wedge (\nexists x : E' \rightarrow_{l_2} x)$.

A pending object is a proof object of the form \$EPNODE[...] or \$PNODE[...]. Pending objects of the form \$EPNODE[...] are called elementary pending objects, and they correspond to the objects of D-trees which are computed when

the depth limit for search is reached. The pending object corresponding to a pending D-tree $T_m(E, l)$ justifies the fact that an exhaustive search until depth m is insufficient for deciding the validity of a formula $\exists x : E \rightarrow_l x$.

4.1 The encoding procedure

We will denote the encoding of a partial D-tree T by $\langle\langle T \rangle\rangle$. We will show how the encoding function of a partial D-tree T can be defined recursively, in terms of the partial D-subtrees of T.

Let T_m be a partial D-tree of depth m and d the search depth limit. The computation of the proof object $\langle\langle T_m \rangle\rangle$ for T_m proceeds as follows:

(i) If $T_m \equiv \dfrac{}{\exists^? x : E \rightarrow_l x}$ with $m < d$ then $\langle\langle T_m \rangle\rangle = \$\texttt{FNODE}[\{E, l\}]$.

(ii) Otherwise, if $T_m \equiv \dfrac{}{\exists^? x : E \rightarrow_l x}$ with $m = d$ then T_m is a partial D-tree of maximum search depth, and $\langle\langle T_m \rangle\rangle = \$\texttt{EPNODE}[\{E, l\}]$.

(iii) Otherwise, if $T_m \equiv \dfrac{E \rightarrow_l E_1 \quad \ldots \quad E \rightarrow_l E_n}{\exists^? x : E \rightarrow_l x}$ with $n > 0$ then

$$\langle\langle T_m \rangle\rangle = \begin{cases} \$\texttt{SNODE}[\{E, l, E_1\}] & \text{if } n = 1, \\ \$\texttt{SNODE}[\{E, l\}, \$\texttt{SNODE}[\{E, l, E_1\}], \ldots, \$\texttt{SNODE}[\{E, l, E_n\}]] & \text{if } n > 1. \end{cases}$$

(iv) Otherwise, if there exists a sequence of partial D-trees $T^1, \ldots, T^n$ of depth most m such that
$T_m = T^1(E, l)$, $l_{i+1} = \texttt{red}[l_i]$, $T^i(E, l_i) = \dfrac{T^{i+1}(E, \texttt{red}[l_i])}{\exists^? x : E \rightarrow_{l_i} x}$ for $1 \leq i < n$
and $T^n(E, l_n) \equiv \dfrac{T^{n,1} \quad \ldots \quad T^{n,k}}{\exists^? x : E \rightarrow_{l_n} x}$ with l_n irreducible and $T^{n,i}$ of depth most $m - 1$, then we have the following cases:

- if there is $i \in \{1, \ldots, k\}$ such that $T^{n,i}$ is a success D-trees then

$$\langle\langle T_m \rangle\rangle = \$\texttt{SNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\langle T^{n,1} \rangle\rangle, \ldots, \langle\langle T^{n,k} \rangle\rangle]$$

- if all $T^{n,1}, \ldots, T^{n,k}$ are failure D-trees then

$$\langle\langle T_m \rangle\rangle = \$\texttt{FNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\langle T^{n,1} \rangle\rangle, \ldots, \langle\langle T^{n,k} \rangle\rangle]$$

- if there is $i \in \{1, \ldots, k\}$ such that $T^{n,i}$ is a pending D-tree, and none of $T^{n,1}, \ldots, T^{n,k}$ is a success D-tree then

$$\langle\langle T_m \rangle\rangle = \$\texttt{PNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\langle T^{n,1} \rangle\rangle, \ldots, \langle\langle T^{n,k} \rangle\rangle].$$

(v) Otherwise, if $T_m \equiv \dfrac{T^1 \ \ldots \ T^k}{\exists^? x : E \rightarrow_l x}$ with T^i of depth most $m-1$ for all $1 \leq i \leq k$ then

- if there is $i \in \{1, \ldots, k\}$ such that T^i is a success D-tree then

$$\langle\!\langle T_m \rangle\!\rangle = \$\texttt{SNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$

- if all $T^1, \ldots, T^k$ are failure D-trees then

$$\langle\!\langle T_m \rangle\!\rangle = \$\texttt{FNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$

- if there is $i \in \{1, \ldots, k\}$ such that T^i is a pending D-tree, and for all $j \in \{1, \ldots, k\} \setminus \{i\}$, T^j are pending or failure D-trees then

$$\langle\!\langle T_m \rangle\!\rangle = \$\texttt{PNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$

(vi) Otherwise, $T_m \equiv \dfrac{(E \rightarrow_{l_1} E_1) \wedge T^1 \ \ldots \ (E \rightarrow_{l_1} E_k) \wedge T^k}{\exists^? x : E \rightarrow_{l_1 \circ l_2} x}$ where T^i are partial D-trees of depth at most $m-1$ for $\exists^? x : E_i \rightarrow_{l_2} x$, for all $1 \leq i \leq k$.

For this situation we will make use of the function $\texttt{Annotate}[E, l, N]$ which is defined for an expression E, rule l and proof object N as follows:

- $\$\texttt{SNODE}[\{E, \langle l, E_1, l' \rangle, E_2\}]$ if $N = \$\texttt{SNODE}[\{E_1, l', E_2\}]$,
- $n[\{E, \langle l, E_1, \mathit{lexpr} \rangle\}, N_1, \ldots, N_k]$ if $N = n[\{E_1, \mathit{lexpr}\}, N_1, \ldots, N_k]$ with $n \in \{\$\texttt{SNODE}, \$\texttt{FNODE}, \$\texttt{PNODE}\}$.

In the case $k = 0$, meaning that $E \not\rightarrow_{l_1}$, we have

$$\langle\!\langle T_m \rangle\!\rangle = \$\texttt{FNODE}[\{E, l_1 \circ l_2\}]$$

For the case $k > 0$ we have the following subcases:

- if there is $i \in \{1, \ldots, k\}$ such that T^i is a success D-tree then $\langle\!\langle T_m \rangle\!\rangle$ is

$$\$\texttt{SNODE}[\{E, l_1 \circ l_2\}, \texttt{Annotate}[E, l_1, \langle\!\langle T^1 \rangle\!\rangle], \ldots, \texttt{Annotate}[E, l_1, \langle\!\langle T^k \rangle\!\rangle]].$$

- if for all $i \in \{1, \ldots, k\}$, T^i is a failure D-tree then $\langle\!\langle T \rangle\!\rangle$ is

$$\$\texttt{FNODE}[\{E, l_1 \circ l_2\}, \texttt{Annotate}[E, l_1, \langle\!\langle T^1 \rangle\!\rangle], \ldots, \texttt{Annotate}[E, l_1, \langle\!\langle T^k \rangle\!\rangle]].$$

- if there is $i \in \{1, \ldots, k\}$ such that T^i is a pending D-tree, and for all $j \in \{1, \ldots, k\} \setminus \{i\}$, T^j are pending or failure D-trees then $\langle\!\langle T_m \rangle\!\rangle$ is

$$\$\texttt{PNODE}[\{E, l_1 \circ l_2\}, \texttt{Annotate}[E, l_1, \langle\!\langle T^1 \rangle\!\rangle], \ldots, \texttt{Annotate}[E, l_1, \langle\!\langle T^k \rangle\!\rangle]].$$

The following example illustrates the behavior of the encoding procedure in a concrete situation.

Example 4.1 Consider the rules declared in Example 1.1 and the query $\exists^? x : -4.2 \rightarrow_{\texttt{"fg"}} x$. The corresponding D-tree is $T(-4.2,\ \texttt{"fg"})$

$$\dfrac{\dfrac{(-4.2 \rightarrow_{\texttt{"f1"}} 2.8) \wedge \dfrac{2.8 \rightarrow_{\texttt{"g"}} 1.4}{\exists^? x : 2.8 \rightarrow_{\texttt{"g"}} x}}{\exists^? x : -4.2 \rightarrow_{\texttt{"f1"}\circ\texttt{"g"}} x} \quad \dfrac{(-4.2 \rightarrow_{\texttt{"f2"}} -0.2) \wedge \dfrac{}{\exists^? x : -0.2 \rightarrow_{\texttt{"g"}} x}}{\exists^? x : -4.2 \rightarrow_{\texttt{"f2"}\circ\texttt{"g"}} x}}{\dfrac{\dfrac{\dfrac{\exists^? x : -4.2 \rightarrow_{(\texttt{"f1"}\circ\texttt{"g"})|(\texttt{"f2"}\circ\texttt{"g"})} x}{\exists^? x : -4.2 \rightarrow_{(\texttt{"f1"}|\texttt{"f2"})\circ\texttt{"g"}} x}}{\exists^? x : -4.2 \rightarrow_{\texttt{"f"}\circ\texttt{"g"}} x}}{\exists^? x : -4.2 \rightarrow_{\texttt{"fg"}} x}}$$

We construct the proof object $\langle\!\langle T \rangle\!\rangle$ incrementally, by traversing it from leaves towards the root.

There are two leaf D-trees:

$$T_1 = \dfrac{3.0 \rightarrow_{\texttt{"g"}} 1.5}{\exists^? x : 3.0 \rightarrow_{\texttt{"g"}} x} \quad \text{and} \quad T_2 = \dfrac{}{\exists^? x : 0. \rightarrow_{\texttt{"g"}} x}$$

with the corresponding proof objects

$$\langle\!\langle T_1 \rangle\!\rangle = \$\texttt{SNODE}[\{3.0,\ \texttt{"g"},\ 1.5\}] \text{ and}$$
$$\langle\!\langle T_2 \rangle\!\rangle = \$\texttt{FNODE}[\{0.,\ \texttt{"g"}\}].$$

The D-subtrees of T which have T_1 and T_2 as direct subtrees, are

$$T_3 = \dfrac{(-4.0 \rightarrow_{\texttt{"f1"}} 3.0) \wedge T_1}{\exists^? x : -4.0 \rightarrow_{\texttt{"f1"}\circ\texttt{"g"}} x} \quad \text{and} \quad T_4 = \dfrac{(-4.0 \rightarrow_{\texttt{"f2"}} 0.) \wedge T_2}{\exists^? x : -4.0 \rightarrow_{\texttt{"f2"}\circ\texttt{"g"}} x}.$$

The corresponding proof objects are

$$\langle\!\langle T_3 \rangle\!\rangle = \$\texttt{SNODE}[\{-4.0, \langle \texttt{"f1"}, 3.0, \texttt{"g"} \rangle, 1.5\}] \text{ and}$$
$$\langle\!\langle T_4 \rangle\!\rangle = \$\texttt{FNODE}[\{-4.0, \langle \texttt{"f2"}, 0., \texttt{"g"} \rangle\}]$$

computed in the way described in case (vi) of the encoding procedure.

The D-subtree of T which has T_3 and T_4 as direct subtrees is

$$T_5 = \dfrac{T_3 \qquad T_4}{\exists^? x : -4.0 \rightarrow_{(\texttt{"f1"}\circ\texttt{"g"})|(\texttt{"f2"}\circ\texttt{"g"})} x}$$

and the corresponding proof object is

$$\langle\!\langle T_5 \rangle\!\rangle = \$\texttt{SNODE}[\{-4.0, (\texttt{"f1"} \circ \texttt{"g"}) \mid (\texttt{"f1"} \circ \texttt{"g"})\}, \langle\!\langle T_3 \rangle\!\rangle, \langle\!\langle T_4 \rangle\!\rangle].$$

The following D-trees correspond to the sequence of rules

$$(\texttt{"f1"} \circ \texttt{"g"}) \mid (\texttt{"f2"} \circ \texttt{"g"}),\ (\texttt{"f1"} \mid \texttt{"f2"}) \circ \texttt{"g"},\ \texttt{"f"} \circ \texttt{"g"},\ \texttt{"fg"}$$

where every element is a reduct of the element which follows it. Therefore, by case (iv), $\langle\langle T_5 \rangle\rangle$ gives us

$$\langle\langle T \rangle\rangle = \$\texttt{SNODE}[\ -4.0, \{\texttt{"fg"}, \texttt{"f"} \circ \texttt{"g"}, (\texttt{"f1"} \mid \texttt{"f2"}) \circ \texttt{"g"}, (\texttt{"f1"} \circ \texttt{"g"}) \mid (\texttt{"f2"} \circ \texttt{"g"})\}, \langle\langle T_3 \rangle\rangle, \langle\langle T_4 \rangle\rangle].$$

It can be easily checked that T has the depth 3. □

5 Visualizing and Manipulating ρLog Proof Objects

Having implemented a data structure for storing a partial D-tree, we desire to see and handle it in a useful way. Because speed is one of the main issues that we had in mind when designing ρLog, by default, the system does not create a proof object. However, the user has the possibility to trigger the creation of it, and choose between different styles of presentation: `"Object"`-style meant for debugging; `"Compact"`- and `"Verbose"`-style for a user friendly presentation. To achieve a natural style presentation of the ρLog proof objects, we adapted and simplified the proof presentation routines of the Theorema system [3,12], which also implements a tree-style data structure for storing proofs.

We have already seen in the Section 1 how ρLog can be invoked. Triggering the different presentation styles is done via Mathematica's options mechanism [15, Section 1.9.5]. The options of `ApplyRule[ ]` and `ApplyRuleList[ ]` are `TraceStyle`, `MaxDepth` and `MaxSols`.

`TraceStyle` can have the following values:

- `"None"` – the default value. `ApplyRule[ ]` will only return an eventually found solution or the original expression if no solution was found. `ApplyRuleList[ ]` will return a possible empty list of solutions. When this value of the option is chosen, no proof object will be created by ρLog. This makes the time needed for obtaining an answer from the system considerably shorter compared with the time needed when using the other options of `TraceStyle`, where an object is created.
- `"Object"` – choosing this value will display the proof object's internal data structure. This can be very large, and inspecting it requires a clear understanding of how the data structure is defined (Section 4).
- `"Compact"` – this value of `TraceStyle` will cause the generation of a Mathematica notebook with a user friendly presentation of the partial

D-tree encoded in the proof object. As the name of the option-value says, it is a concise presentation of the rewriting process, skipping all the details about reducing the rules and selecting subexpressions.

- `"Verbose"` – it is similar with the previous option value. The difference consists in the amount of presented information. Steps that reflect reduction of rules, selection of subexpressions are now shown to the user. See Example 5.1 for an illustration of this option's effect.

The `"Compact"` and `"Verbose"` styles of presentation take advantage of MATHEMATICA's notebook features, the most important which we mention here is having nested cells to reflect the structure of the (partial) D-tree.

`MaxDepth` The purpose of this option is to avoid infinite computations determined by infinitely long branches in the search space for a derivation. Its default value is 10000. When `TraceStyle` has a different value than `"None"`, the value of `MaxDepth` determines the maximum depth of the partial D-tree that is encoded in the proof object which will be created.

`MaxSols` The purpose of this option is to impose an upper limit on the number of expressions E' to be found as witnesses for the validity of the query $\exists^{?} x : E \to_l x$. For example, a call

$$\texttt{ApplyRuleList}[E,\ l,\ \texttt{MaxSols} \to 3]$$

will stop the solution search process as soon as it had found 3 expressions E_1, E_2, E_3 for which $E \to_l E_i$.

The default value of `MaxSols` is ∞. This means that, by default, we do not impose any upper bound on the number of solutions to be found.

A very useful feature of ρLOG is the possibility to further expand a pending partial D-tree encoded in a proof object, and obtain a new proof object corresponding to the expanded partial D-tree. This can be done by invoking

$$\texttt{ExpandObject}[obj,\ \texttt{MaxDepth} \to n];$$

where obj is a proof object and $n \in \mathbb{N}$ is the depth limit for the partial D-trees which will be computed and encoded to replace the elementary pending objects which occur in obj.

Displaying a proof object in a nice, user friendly manner, can be done not only using the `TraceStyle` option of `ApplyRule[ ]` but also by a call

$$\texttt{DisplayProof}[obj,\ options];$$

where the only option available at this time is `DetailLevel`. The possible values of `DetailLevel` and their meaning in `DisplayProof[ ]` calls coincide

with those of `TraceStyle` in `ApplyRule[ ]` calls.

Example 5.1 As an illustration of `TraceStyle` → `"Verbose"` usage we consider the query in Example 1.1, with the requirement for a `"Verbose"` presentation style:

$$\texttt{ApplyRule}[-4.0, \texttt{"fg"}, \texttt{TraceStyle} \rightarrow \texttt{"Verbose"}];$$

In this case, the notebook generated by the call looks as in Figure 2.

Notice that, with this presentation style, the information about the reducing sequence of `"fg"` rule is presented to the user. □

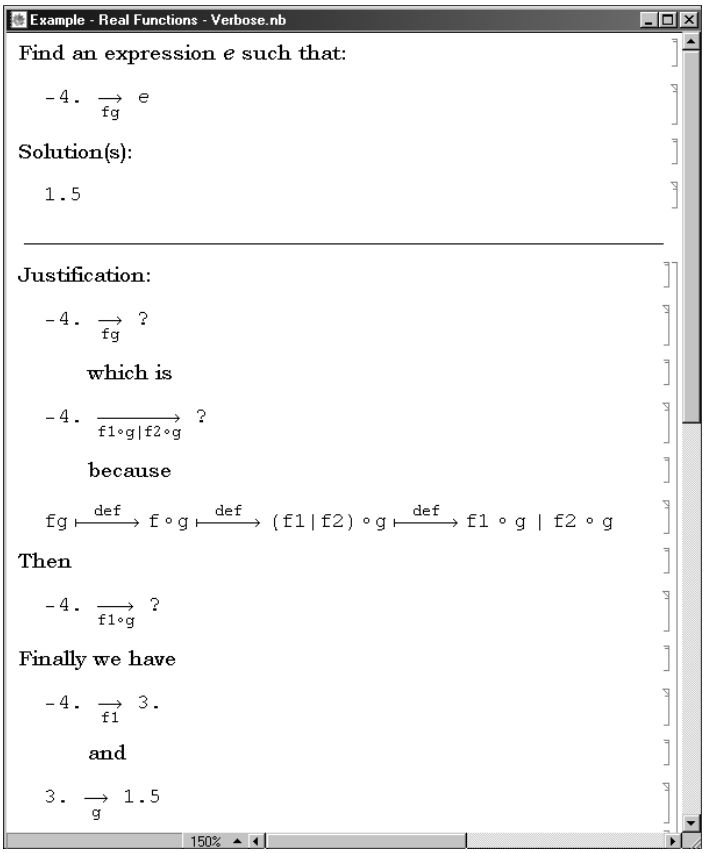

Fig. 2. `"Verbose"`-style presentation of a rule-based deduction.

6 Conclusion and Future Work

The design and implementation of ρLOG was motivated by the desire to have a convenient tool to program reasoners with MATHEMATICA. The design of its proof presentation capabilities is inspired from that of THEOREMA, whose provers work in a "natural style". That is, the inference rules are similar to the heuristics used by mathematicians, and the produced output is similar to the proofs written by them [12].

Obviously, the range of problems which can be modelled and solved efficiently with ρLOG is very large. Most of its expressive power stems from the capability to model non-deterministic computations as compositions of possibly non-deterministic rules. We expect to identify new combinators for composing rules in ways which are useful in strategy specifications. For instance, it is often desirable to have a combinator

$$\texttt{XOR}[l_1, l_2, \ldots, l_n]$$

whose applicative behavior is: $E \rightarrow_{\texttt{XOR}[l_1,l_2,\ldots,l_n]} E'$ iff $E \rightarrow_{l_i} E'$, where l_i is the first element of the sequence $l_1, l_2, \ldots, l_n$ for which $E \rightarrow_{l_i}$. This combinator turned out to be useful in implementing the lazy narrowing calculi for E-unification described in [9,11]. Please note that the semantics of this combinator is different from that of the choice combinator, and that the extension of our language with this combinator will need a revision of notion of pending object in particular, and of proof object in general. We are currently working on integrating this combinator in our system.

We have described how ρLOG can be employed as a deductive system and be used to generate deduction trees for a certain kind of queries. We expect our system to become a useful tool in the ongoing development of the THEOREMA system [1,2,3], which is a framework aimed to support the main activities of the working mathematician: proving, solving, computing, exploring and developing new theories. Up to now, we have implemented a library of unification procedures for free, flat and restricted flat theories with sequence variables and flexible arity symbols [4,7]. These unification procedures have straightforward and efficient implementations in ρLOG because they are based on the non-deterministic application of a finite set transformation rules.

Another direction of future work is to introduce control mechanisms for pattern matching with sequence variables. In the current implementation, when enumerating the matching substitutions θ during a call of `ApplyRule[ ]`, ρLOG relies entirely on the enumeration strategy which is built into the MATHEMATICA interpreter. However, there are many situations when this enumeration strategy is not desirable. We addressed this problem in [5,6] and

implemented the package SEQUENTICA with language extensions which can overwrite the default enumeration strategy of the MATHEMATICA interpreter. The integration of those language extensions in ρLOG will certainly increase the expressive power of our rule based system. We are currently working on integrating SEQUENTICA with ρLOG.

The current implementation of ρLOG can be downloaded from

http://heaven.ricam.uni-linz.ac.at/people/page/marin/RhoLog/

Acknowledgement

We would like to thank our colleague and friend Temur Kutsia for his useful comments and suggestions on how to improve the paper.

References

[1] Bruno Buchberger. THEOREMA: A short introduction. *Mathematica Journal*, 8(2):247–252, 2001.

[2] Bruno Buchberger, Claudio Dupré, Tudor Jebelean, Franz Kriftner, Koji Nakagawa, Daniela Văsaru, and Wolfgang Windsteiger. The THEOREMA project: A progress report. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning. Proceedings of Calculemus'2000*, pages 98–113, St.Andrews, UK, 6–7 August 2000.

[3] Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuţa, and Daniela Văsaru. A survey of the THEOREMA project. In Wolfgang Küchlin, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC'97*, pages 384–391, Maui, Hawaii, US, 21–23 July 1997. ACM Press.

[4] Temur Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Institute RISC-Linz, Johannes Kepler University, Hagenberg, Austria, June 2002.

[5] Mircea Marin. Functional Programming with Sequence Variables: The Sequentica Package. In Jordi Levy, Michael Kohlhase, Joachim Niehren, and Mateu Villaret, editors, *Proceedings of the 17th International Workshop on Unification (UNIF 2003)*, pages 65–78, Valencia, June 2003.

[6] Mircea Marin and Dorin Ţepeneu. Programming with Sequence Variables: The Sequentica Package. In Peter Mitic, Phil Ramsden, and Janet Carne, editors, *Challenging the Boundaries of Symbolic Computation. Proceedings of 5th International Mathematica Symposium (IMS 2003)*, pages 17–24, Imperial College, London, July 7–11 2003. Imperial College Press.

[7] Mircea Marin and Temur Kutsia. On the Implementation of a Rule-Based Programming System and some of its Applications. In Boris Konev and Renate Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics*, pages 55–69, Almaty, Kazakhstan, September 26 2003.

[8] Mircea Marin and Temur Kutsia. Programming with Transformation Rules. In *Proceedings of the 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing*, pages 157–167, Timişoara, Romania, October 1-4 2003.

[9] Mircea Marin and Aart Middeldorp. New Completeness Results for Lazy Conditional Narrowing. In *Proceedings of 6h International Workshop on Unification (UNIF 2002)*, Copenhagen, Denmark, July 22-26 2002.

[10] Mircea Marin and Florina Piroi. Rule-based Deduction and Views in Mathematica. Technical Report SFB-2003-43, Johannes Kepler University, Linz, October 2003.

[11] Aart Middeldorp and Satoshi Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[12] Florina Piroi and Tudor Jebelean. Advanced proof presentation in THEOREMA. In *Proceedings of the 3rd International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2001)*, Timişoara, Romania, October 2-5 2001. Also avaialable as RISC-Linz Report Series No. 01-20.

[13] P. A. Subrahmanyam and Jia-Huai You. FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming. *Logic Programming: Functions, Relations, and Equations*, pages 157–198, 1986.

[14] Geoffrey Norman Watson. Proof representation in theorem provers. Technical Report 98-13, Software Verification Centre, School of Informatics Technology, The University of Queensland, Queensland 4072, Australia, September 1998.

[15] Stephen Wolfram. *The Mathematica Book*. Wolfram Media Inc. Champaign, Illinois, USA and Cambridge University Press, 1999.