Rule-based Deduction and Views in MATHEMATICA

Mircea Marin[1⋆] and Florina Piroi[2⋆⋆]

[1] Johann Radon Institute for Computational and Applied Mathematics
Austrian Academy of Sciences
Altenberger Straße 69, A-4040 Linz, Austria
Mircea.Marin@oeaw.ac.at
[2] Research Institute for Symbolic Computation
Johannes Kepler University
A-4232 Hagenberg, Austria
Florina.Piroi@risc.uni-linz.ac.at

Abstract. We propose a rule-based system built on top of the capabilities of MATHEMATICA to program non-deterministic and partially defined computations. The system is called ρLOG and has primitive operators for defining elementary rules and for computing with unions, compositions, reflexive-transitive closures, and normal forms of rule applications. Moreover, ρLOG can compute proof objects, which are internal representations of deduction derivations which respect a specification given by the user. We describe the programming principles and constructs of ρLOG, the structures used to encode deduction derivations, and the methods provided to manipulate and visualize them.

1 Preliminaries

ρLOG is a renamed version of the rule-based programming system FUNLOG [9, 10]. We did this in order to avoid confusing it with `FUNLOG` [12], a system for functional programming and logic programming developed in the eighties. Like Elan [2], ρLOG provides a suitable environment for specifying and implementing deduction systems in a language based on rewrite rules whose application is controlled by user-defined strategies. More precisely, ρLOG allows to:

1. program non-deterministic computations by using the advanced features of MATHEMATICA [14], such as: matching with sequence patterns, and access to state-of-the art libraries of methods for symbolic and numeric computation;
2. program rules l whose reduction relation $\rightarrow_l$ can be defined, possibly recursively, in terms of already defined reduction relations $\rightarrow_{l_1}, \ldots, \rightarrow_{l_n}$;
3. 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 $\exists^? x : E \rightarrow_l x$;

⋆ Mircea Marin has been supported by the Austrian Academy of Sciences.
⋆⋆ Florina Piroi has been supported by the Austrian Science Foundation FWF, under the SFB grant F1302.

4. generate proof objects which encode deductions to 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:

1. Mathematica 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,
2. it has very good support for symbolic and numeric computations,
3. rule-based programming, as envisioned by us, could be used efficiently to implement provers, solvers and simplifiers which could be integrated in the Theorema framework [5]. Since Theorema is implemented in Mathematica, a Mathematica implementation of a powerful rule-based system could become a convenient programming tool for Theorema developers.

The rest of this paper is structured as follows. Section 2 explains the principles of programming with rules in ρLog. The programming constructs of ρLog are described in Section 3. We illustrate the deductive capabilities of ρLog in Section 4. In Section 5 we describe the general structure of deduction derivations in ρLog. Section 6 is about proof objects, which constitute the internal representation of deduction derivations. Section 7 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 8 concludes.

In order to help the understanding of the pieces of Mathematica code included in this report, we provide Appendix A with a brief description of the pattern matching constructs of the Mathematica language.

2 Towards ρLog Rules

This section is intended to give the reader a clear understanding of what ρLog rules are.

2.1 Rules

ρLog is a system for rule-based programming, in which rules specify non-deterministic and partially defined computations. Formally, a rule has a specification of the form

$$l :: \mathit{patt} :\rightarrow \mathit{rhs} \tag{1}$$

where l is the *rule name* and $\mathit{patt} :\rightarrow \mathit{rhs}$ is called the *rule code*. *patt* is a pattern expression called the *left-hand side* of the rule, and *rhs* is called the *right-hand side* of the rule. *rhs* specifies a computation in terms of the variables which occur in *patt*. The computation described by *rhs* may be non-deterministic and partially defined (see Section 2.2).

The main usage of rules is to apply them on expressions. Any expression which can be represented in the language of Mathematica [14] is a valid 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 is 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''.

Note that 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.

Example 1. Take the expressions

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

E_1, E_2, E_2 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

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

□

The procedure which attempts to apply a rule l on a ρ-valid expression E and returns the first result found is described below.

`Procedure` **ApplyRule**$(E[\![E']\!], l)$
$[E[\![E']\!]$ is a ρ-valid expression, l is the name of a rule $l :: \mathit{patt} \to \mathit{rhs}]$
$TS = \{\ \}$;
`while` (there is a substitution θ such that $\theta(\mathit{patt}) = E'$) and ($\theta \notin TS$) `do`
    `if` $\theta(rhs)$ has a value
        `then`
            E'' = first value of $\theta(rhs)$;
            `return` $E[\![E'/E'']\!]$
        `else` $TS = TS \cup \{\theta\}$;
    `fi`;
`od`.

We call the substitutions θ *matchers* between *patt* and E'. The meaning of $\theta(rhs)$ is the following: compute the list of all possible values of *rhs* after instantiating its variables with the bindings provided by the substitution θ.

The procedure which attempts to apply a rule l on a ρ-valid expression E and returns *all* results is:

`Procedure` **ApplyRuleList**$(E[\![E']\!], l)$
$[E[\![E']\!]$ is a ρ-valid expression, l is the name of a rule $l :: \mathit{patt} \rightarrow \mathit{rhs}]$
$TS = \{\ \}$; $V = \{\ \}$;
`while` (there is a substitution θ such that $\theta(\mathit{patt}) = E'$) and $(\theta \notin TS)$ `do`
  $TS = TS \cup \{\theta\}$;
  $V = V \cup \{E[\![E'/E'']\!] \mid E'' \in \theta(rhs)\}$;
`od`;
`return` V.

For a given rule $l :: \mathit{patt} :\rightarrow \mathit{rhs}$, we convene to write $E' \rightarrow_l E''$ if there exist

1. an enumeration strategy for matchers, and
2. an enumeration strategy of the elements of the lists of instances of *rhs*

for which the call **ApplyRule**$[E', l]$ yields E''. If the call **ApplyRule**$[E, l]$ does not produce any value then we write $E \not\rightarrow_l$. We write $\rightarrow_l^*$ for the reflexive-transitive closure of $\rightarrow_l$.

We conclude this section with the following observations:

1. a rule is applied to a whole expression or to a selected subexpression of an expression,
2. rule application is a non-deterministic operation,
3. rule application is a partially defined operation,
4. rules can be composed into more complex rules via various combinators.

In the sequel we will assume implicitly that $E, E', E_1, E_2, \ldots$ denote ρ-valid expressions, and that $l, l', l_0, l_1, l_2, \ldots$ denote ρLOG rules.

2.2 Non-determinism

There are two sources of non-determinism when trying to apply a rule l, with $l :: \mathit{patt} :\rightarrow \mathit{rhs}$, on some expression E: non-unique matchers θ, and non-unique ways to evaluate a partially defined computation $\theta(\mathit{rhs})$. Clearly, the result of an application attempt $E \rightarrow_l$ depends on the enumerations of matchers and results which are built into a particular implementation. These enumeration strategies are relevant to the programmer and are described in the specification of the operational semantics of ρLOG (Section 3).

2.2.1 Matchers. The matching mechanism of ρLOG uses the one of MATHEMATICA [14, Sect. 2.3.8], which allows the usage of pattern constructs which have no unique matchers. Such patterns can be specified if we employ sequence variables[1] and/or alternative patterns via the "|" construct.

Example 2. In MATHEMATICA, the pattern $\{x\_\_\_, 1, y\_\_\_\} \mid \{y\_\_\_, 2, x\_\_\_\}$ matches the list $\{a, 2, 1, 2, b\}$ in 3 possible ways:

$$\{x \mapsto \ulcorner a, 2\urcorner, y \mapsto \ulcorner 2, b\urcorner\}, \quad \{y \mapsto \ulcorner a\urcorner, x \mapsto \ulcorner 1, 2, b\urcorner\}, \quad \{y \mapsto \ulcorner a, 2, 1\urcorner, x \mapsto \ulcorner b\urcorner\}.$$

[1] They are called *named sequence patterns* in the MATHEMATICA book [14].

The symbols x and y in the pattern are followed by three underscores, and therefore they denote variables which can be bound to an arbitrary sequence of elements. To improve the readability of matchers, we have written the bindings of sequence variables between $\ulcorner$ and $\urcorner$. The first matcher is the result of matching with the first alternative of the pattern, whereas the last two are the result of matching with the second alternative of the pattern. $\square$

The MATHEMATICA interpreter relies on the following built-in enumeration strategy [14, Sect. 2.3.8]: it tries first those matches that assign the shortest possible sequences of arguments to the first sequence variables that are encountered while traversing the pattern in a leftmost-innermost manner.

2.2.2 Partially defined computations. Because ρLOG is implemented in MATHEMATICA, the partially defined computations in ρLOG are specified via MATHEMATICA's pattern matching constraints and/or by attaching side conditions to *patt* or to *rhs* of (1). We write $E/;cond$ for an expression E whose instances $\sigma(E)$ are defined if and only if $\sigma(cond)$ evaluates to `True` (for some substitution σ). The boolean expression *cond* is called the *side condition* which determines whether a certain instance of E is defined or not.

Example 3. The expression

$$x - y/;(\texttt{IntegerQ}[x] \wedge \texttt{IntegerQ}[y] \wedge (x > y))$$

specifies a binary subtraction operation $- : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ which is defined only for pairs $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ with $x > y$. $\square$

Thus, a rule with a condition attached to the right-hand side is of the form

$$l :: patt :\longrightarrow\ rhs/;cond \tag{2}$$

and it defines a reduction relation $\to_l$ such that

$$\forall E_1, E_2 : E_1 \to_l E_2 \text{ iff } E_2 = \theta(rhs) \wedge \theta(cond) = \texttt{True}$$

where θ is a matcher between E_1 and *patt*.

The rule

$$l' :: patt/;cond :\longrightarrow\ rhs$$

differs from the rule l of (2) by having the side condition attached to the pattern instead of the right-hand side of the rule code. The applicative behaviors of rules l and l' are the same.

It is possible to share variables between *cond* and *rhs*. This feature comes in very handy when we want to use variables instantiated during the evaluation of $\theta(cond)$ for computations in $\theta(rhs)$, when $\theta(cond)$ gives `True`. This capability is achieved by enclosing the evaluation of $rhs/;cond$ in a MATHEMATICA `Block` or `Module` construct, with a declaration of the variables shared between *rhs* and *cond*.

Example 4. Consider the rule "second" defined by

$$\begin{array}{l}\texttt{"second"} :: \\ \quad \texttt{X\_} :\!\to \texttt{Module}\ [\{\texttt{res}, \texttt{OK}\}, \texttt{res}/; (\\ \qquad\qquad\qquad \texttt{OK} = \texttt{True};\\ \qquad\qquad\qquad \texttt{res} = \texttt{Replace}[\texttt{X}, \{\{\_, x\_, \_\_\_\} :\!\to x,\ x\_ :\!\to (\texttt{OK} = \texttt{False};\ x)\}];\\ \qquad\qquad\qquad \texttt{OK})];\end{array}$$

The rule defines the extraction of the second element from a list expression[2]. An attempt to apply rule "second" to a concrete MATHEMATICA expression E will bind E to X (ba matching E with X_), and thus the evaluation of the Module-construct will assume the value E for X. The evaluation of the right-hand side of "second" proceeds as follows:

– If the transformation rule $\{\_, x\_, \_\_\_\} :\!\to x$ is applicable to E, i.e. E is a list and it has at least two elements, then the command

$$\texttt{res} = \texttt{Replace}[\texttt{X}, \{\{\_, x\_, \_\_\_\} :\!\to x,\ x\_ :\!\to (\texttt{OK} = \texttt{False};\ x)\}]$$

  will bind x to the second element of E and assign the value of x to res. In this way, res gets bound to the second element of E. The value True of OK remains unchanged. Therefore, the return value of the rule application is the second element of E. In symbols, this means $E \to_{\texttt{"second"}} E_2$ where E_2 is the second element of the list expression E.
– If the transformation rule $\{\_, x\_, \_\_\_\} :\!\to x$ is not applicable to E then the effect of the command

$$\texttt{res} = \texttt{Replace}[\texttt{X}, \{\{\_, x\_, \_\_\_\} :\!\to x,\ x\_ :\!\to (\texttt{OK} = \texttt{False};\ x)\}]$$

  is to assign the value of E to x (via pattern matching), assign False to OK, and finally assign the value of x, which is E, to res. In this way, the side condition of the right-hand side of rule "second" evaluates to False. Therefore $E \not\to_{\texttt{"second"}}$. □

Example 5. The rule

$$\texttt{"split"} :: \{x\_\_\_, y\_\_\_\}/; (\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}]) :\!\to \{x\}$$

takes as input a list L and computes a prefix sublist of L whose length is longer than half the length of L. The attempt to apply rule "split" to $\{a, b, c\}$ proceeds as follows:

1. It starts to enumerate the matchers between $\{a, b, c\}$ and $\{x\_\_\_, y\_\_\_\}$ in the order which is specific to the MATHEMATICA interpreter, i.e.:

   $\{x \mapsto \ulcorner\urcorner, y \mapsto \ulcorner a, b, c\urcorner\},\ \ \{x \mapsto \ulcorner a\urcorner, y \mapsto \ulcorner b, c\urcorner\},\ \ \{x \mapsto \ulcorner a, b\urcorner, y \mapsto \ulcorner c\urcorner\},\ \ldots$
2. The enumeration is generated until we reach a matcher θ for which the condition $\theta(\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}])$ holds. In this example, the enumeration stops when it reaches the matcher $\theta = \{x \mapsto \ulcorner a, b\urcorner, y \mapsto \ulcorner c\urcorner\}$ and the computation resumes with the value $\{a, b\}$ of the instance $\theta(\{x\})$. □

[2] Replace[E, *rules*] applies a rule or list of rules *rules* to E in an attempt to transform the entire expression E [14].

2.3 Operations on rules

Rules can be composed with various combinators into more complex rules which capture the most common ways of programming computations. ρLog provides implementations for the following combinators:

choice: If $l_1, \ldots, l_n$ are rules then $l_1 \mid \ldots \mid l_n$ is a rule such that $E_1 \rightarrow_{l_1|\ldots|l_n} E_2$ 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 such that $E_1 \rightarrow_{l_1 \circ l_2} E_2$ 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 such that $E_1 \rightarrow_{\texttt{NF}[l]} E_2$ 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 the declaration $\texttt{RWRule}[l_1, l, \mathit{options}]$ defines the rule l with

$E \rightarrow_l E'$ iff there exists a subexpression E_1 of E such that $E_1 \rightarrow_{l_1} E_2$, and E' is the result of replacing in E the occurrence of E_1 with E_2.

l is called the *rewrite rule* induced by l_1. The operational semantics of rewriting depends on the choice of the subexpression on which l_1 can act. The choice strategy of such expressions can be controlled via certain *options* to the `RWRule[ ]` call. (See subsection 3.5.)

The implementation of ρLog is compositional, i.e., the meaning and enumeration strategy of each program construct can be defined in terms of the meanings and enumeration strategies of the component rules. These issues are addressed in the following section.

3 Programming Constructs

A rule l is applied to an expression E via the call

$$\texttt{ApplyRule}[E, l]$$

If $E \rightarrow_l$, the call yields the value computed by the procedure **ApplyRule** described in Section 2.1. Otherwise, $E \not\rightarrow_l$ and the call returns the unevaluated expression E. The call

$$\texttt{ApplyRuleList}[E, l]$$

returns the list computed by the procedure **ApplyRuleList** described in Section 2.1. Both methods `ApplyRule[ ]` and `ApplyRuleList[ ]` recognize a number of options which can affect the output format and accuracy of the computed result (more about this in Section 7).

In the remainder of this section we will describe the methods and constructs mentioned in Section 2.3. For each of them we will explain how the enumeration of values is done, and give some illustrative examples.

3.1 Basic rule

Basic rules are the most elementary programming constructs of ρLOG. A basic rule is a named MATHEMATICA transformation rule. A basic rule $l :: patt :\!\rightarrow rhs$ is declared by

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

The enumeration strategy for basic rules depends only on the enumeration strategy of matchers between the (selection in the) input expression and the pattern of the rule, which is the enumeration strategy implemented in MATHEMATICA.

Example 6. The rule "perm" introduced by the declaration

$$\texttt{DeclareRule}[\{x\_\_\_, m\_, y\_\_\_, n\_, z\_\_\_\}/;(m > n) :\!\rightarrow \{x, n, y, m, z\}, \texttt{"perm"}]$$

takes as input a list of elements and permutes the elements which occur in descending order. The outcome of the call

$$\texttt{ApplyRule}[\{1, 2, 5, 4, 3\}, \texttt{"perm"}]$$

is $\{1,\ 2,\ 4,\ 5,\ 3\}$ where, the matcher is $\theta = \{x \mapsto \ulcorner 1, 2\urcorner, m \mapsto 5, y \mapsto \ulcorner\urcorner, n \mapsto 4, z \mapsto \ulcorner 3\urcorner\}$. □

3.2 Choice

$l_1 \mid \ldots \mid l_n$ is the rule whose applicative behavior is given by

$$E_1 \rightarrow_{l_1|\ldots|l_n} E_2 \text{ iff } E_1 \rightarrow_{l_i} E_2 \text{ for some } i \in \{1, \ldots, n\}.$$

Enumerating the values for $E_1 \rightarrow_{l_1|\ldots|l_n} E_2$ starts with enumerating the values for $E_1 \rightarrow_{l_1}$, followed by the enumeration of values for $E_1 \rightarrow_{l_2}$, and so on up to the enumeration of the values for $E_1 \rightarrow_{l_n}$.

Example 7. Consider the declarations

$$\texttt{DeclareRule}[\{x\_\_\_, m\_, y\_\_\_, n\_, z\_\_\_\}/;(m > n) :\!\rightarrow \texttt{False}, \texttt{"test"}];$$
$$\texttt{DeclareRule}[\{x\_\_\_\} :\!\rightarrow \texttt{True}, \texttt{"else"}];$$

Then

$$\texttt{ApplyRule}[L, \texttt{"test"} \mid \texttt{"else"}]$$

yields `True` iff L is a list with elements arranged in ascending order. □

3.3 Composition

$l_1 \circ l_2$ is the rule whose applicative behavior is given by

$$E_1 \rightarrow_{l_1 \circ l_2} E_2 \text{ iff } E_1 \rightarrow_{l_1} E \text{ and } E \rightarrow_{l_2} E_2 \text{ for some } E.$$

The enumeration of values E_2 for which the relation $E_1 \rightarrow_{l_1 \circ l_2} E_2$ holds, proceeds by enumerating values E_2 such that $E \rightarrow_{l_2} E_2$ during an enumeration of the values E such that $E_1 \rightarrow_{l_1} E$.

Example 8. The piece of code

$$\begin{array}{l}\texttt{DeclareRule}[x\_\texttt{Real}/; x < 0 :\rightarrow x + 7, \texttt{"f"}];\\ \texttt{DeclareRule}[x\_\texttt{Real}/; x > 0 :\rightarrow \sqrt{x}, \texttt{"g"}];\end{array}$$

defines rules named "f" and "g" which encode the partially defined functions

$$\begin{array}{l}f : (-\infty, 0.) \rightarrow \mathbb{R},\ x \mapsto x + 7,\\ g : (0., \infty) \rightarrow \mathbb{R},\quad x \mapsto \sqrt{x}.\end{array}$$

Then "f" $\circ$ "g" is the rule which encodes the function $g \circ f : (-7, 0) \rightarrow \mathbb{R}$, and the call

$$\texttt{ApplyRule}[-2., \texttt{"f"} \circ \texttt{"g"}]$$

gives the result 2.23607. □

3.4 Reflexive-transitive closures

Their applicative behavior of $\texttt{Repeat}[l_1, l_2]$ and $\texttt{Until}[l_2, l_1]$ is obtained by unfolding the built-in recursive definitions:

$$\begin{array}{r}\texttt{Repeat}[l_1, l_2] = (l_1 \circ \texttt{Repeat}[l_1, l_2]) \mid l_2,\\ \texttt{Until}[l_2, l_1] = l_2 \mid (l_1 \circ \texttt{Until}[l_2, l_1]).\end{array}$$

It is easy to see that if $l \in \{\texttt{Repeat}[l_1, l_2], \texttt{Until}[l_2, l_1]\}$ then

$$E_1 \rightarrow_l E_2 \text{ iff } E_1 \rightarrow_{l_1}^{*} E \text{ and } E \rightarrow_{l_2} E_2 \text{ for some } E$$

where $\rightarrow_{l_1}^{*}$ denotes the reflexive-transitive closure of $\rightarrow_{l_1}$. The enumeration strategy for these rules can be expressed in terms of the enumeration strategies for rule composition and choice described before.

Example 9 (Sorting). Consider the declarations of basic rules

$$\begin{array}{l}\texttt{DeclareRule}[x\_ :\rightarrow x, \texttt{"Id"}];\\ \texttt{DeclareRule}[\{x\_\_\_,\ m\_,\ y\_\_\_, n\_,\ z\_\_\_\}/; (m > n) :\rightarrow \{x, n, y, m, z\}, \texttt{"perm"}];\end{array}$$

Then, an application of rule Repeat["perm", "Id"] to any list of integers yields the sorted version of that list. For example

$$\texttt{ApplyRule}[\{3, 1, 2\}, \texttt{Repeat}[\texttt{"perm"}, \texttt{"Id"}]]$$

gives $\{1, 2, 3\}$ via the following sequence of rule application steps:

$$\{3, 1, 2\} \rightarrow_{\texttt{"perm"}} \{1, 3, 2\} \rightarrow_{\texttt{"perm"}} \{1, 2, 3\} \rightarrow_{\texttt{"Id"}} \{1, 2, 3\}.$$ □

3.5 Selection shift rules and rewrite rules

ρLOG allows programming rules which behave like term rewriting rules induced by an already existing rule. In other words, if we have a rule l_1 we can define a rule l with the following behavior:

$$E \rightarrow_l E_1 \text{ iff there exists a subexpression } E' \text{ of } E \text{ such that}$$
$$E' \rightarrow_{l_1} E'' \text{ and } E_1 = E[\![E'/E'']\!].$$

A rewrite step $E \rightarrow_l E_1$ can be regarded as a composition of two steps: one which selects the subexpression of E to be rewritten, followed by another one which rewrites it. To achieve suitable selection strategies for rewriting, we have designed two kinds of rules:

1. the basic rule "Rw" whose applicative behavior can be depicted as follows:

$$E[\![E']\!] \rightarrow_{\texttt{"Rw"}} E[\![E'/\underline{E'}]\!].$$

   This means that, if E has no selected subexpressions then we add a selection to it as a whole, otherwise we add a selection to the innermost selected subexpression.
2. 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) a computable function $\texttt{shift}_l$, and (b) a rule $\texttt{r}_l$, with

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

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

$$\texttt{RWRule}[l_1, l, \texttt{Traversal} \rightarrow val, \texttt{Prohibit} \rightarrow \{f_1, \ldots, f_n\}] \tag{3}$$

with $val \in \{\texttt{"LeftIn"}, \texttt{"LeftOut"}\}$ and $\{f_1, \ldots, f_n\}$ a list of MATHEMATICA symbols. The option `Traversal` defines the choice strategy of the rewrite rule. `Traversal` $\rightarrow$ "LeftIn" will 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 rewrite process will ignore the subexpressions of the expressions with the outermost symbol one of $f_1, \ldots, f_n$. The default value of `Prohibit` is $\{\}$; this means that rewriting can be performed everywhere.

The call (3), 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

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

The call (3) will also add the recursive definition $l = \texttt{"Rw"} \circ \texttt{r}_{\texttt{SEL}[l]}$ into the ρLog session.

Throughout this paper we will always assume that $\texttt{r}_l$ and $\texttt{shift}_l$ represent the rule and the computable function associated with a selection shift rule l.

Example 10. Consider the declarations

$$\begin{array}{l} \texttt{DeclareRule}[\texttt{f}[x\_, \texttt{e}] :\to x, \texttt{"N"}]; \\ \texttt{RWRule}[\texttt{"N"}, \texttt{"N*"}, \texttt{Traversal} \to \texttt{"LeftOut"}]; \end{array}$$

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

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 conclude the description of rewriting with an example which shows how one can implement the evaluator of pure λ-calculus as a ρLog rewrite rule.

Example 11 (Pure λ-calculus). In λ-calculus [1], a value is an expression which has no β-redexes outside λ-abstractions. We adopt the following syntax for λ-terms:

$$\begin{array}{lll} \texttt{term} ::= & & \\ \quad \mid & x & \text{variable} \\ \quad \mid & \texttt{app}[\texttt{term}_1, \texttt{term}_2] & \text{application} \\ \quad \mid & \lambda[x, \texttt{term}] & \text{abstraction} \end{array}$$

β-redexes are eliminated by applications of the β-conversion rule, which can be encoded in ρLog as follows:

$$\texttt{DeclareRule}[\texttt{app}[\lambda[x\_, t1\_], t2\_] :\to \texttt{repl}[t_1, \{x, t2\}], \texttt{"}\beta\texttt{"}]$$

where $\texttt{repl}[t_1, \{x, t2\}]$ is defined by the user. In our example we want this function to replace all free occurrences of x in $t1$ by $t2$. A straightforward implementation of repl in MATHEMATICA is[3]:

$$\begin{array}{l}\texttt{repl}[\lambda[x\_,t\_],\{x\_,\_\}] := \lambda[x,t];\\ \texttt{repl}[x\_,\{x\_,t\_\}] := t;\\ \texttt{repl}[\lambda[x\_,t\_],\sigma\_] := \lambda[x,\texttt{repl}[t,\sigma]];\\ \texttt{repl}[\texttt{app}[t1\_,t2\_],\sigma\_] := \texttt{app}[\texttt{repl}[t1,\sigma],\texttt{repl}[t2,\sigma]];\\ \texttt{repl}[t\_,\_] := t;\end{array}$$

The computation of a value of a λ-term proceeds by repeated reductions of the redexes which are not inside abstractions. In ρLOG, the reduction of such a redex coincides with an application of the rewrite rule "β-elim" defined by

$$\texttt{RWRule}[\text{"}\beta\text{"},\text{"}\beta\texttt{-elim"},\texttt{Prohibit}\to\{\lambda\}]$$

The following calls illustrate the behavior of this rule:

$$\begin{array}{l}t := \texttt{app}[z,\texttt{app}[\lambda[x,\texttt{app}[x,\lambda[y,\texttt{app}[x,y]]]],\lambda[z,z]]];\\ t1 := \texttt{ApplyRule}[t,\text{"}\beta\texttt{-elim"}]\\ \ \texttt{app}[z,\texttt{app}[\lambda[z,z],\lambda[y,\texttt{app}[\lambda[z,z],y]]]]\end{array}$$

$$\begin{array}{l}t2 := \texttt{ApplyRule}[t1,\text{"}\beta\texttt{-elim"}]\\ \ \texttt{app}[z,\lambda[y,\texttt{app}[\lambda[z,z],y]]]\end{array}$$

$$\begin{array}{l}t3 := \texttt{ApplyRule}[t2,\text{"}\beta\texttt{-elim"}]\\ \ \texttt{app}[z,\lambda[y,\texttt{app}[\lambda[z,z],y]]]\end{array}$$

Thus, $t2$ is a normal form of t with respect to the rule "β-elim". To compute a value of t directly, we could call

$$\begin{array}{l}\texttt{ApplyRule}[t,\texttt{Repeat}[\text{"}\beta\texttt{-elim"},\texttt{"Id"}]]\\ \ \texttt{app}[z,\lambda[y,\texttt{app}[\lambda[z,z],y]]]\end{array}$$

□

3.6 Normal form

If l is a rule then $\texttt{NFQ}[l]$ is a built-in construct for a rule whose applicative behavior is defined by

$$E \to_{\texttt{NFQ}[l]} E \text{ iff } E \not\to_l .$$

$\texttt{NF}[l]$ denotes a rule whose applicative behavior is given by

$$E \to_{\texttt{NF}[l]} E' \text{ iff } (E \to_l^* E' \text{ and } E' \not\to_l).$$

The enumeration strategy of $\texttt{NF}[l]$ is obtained by unfolding the built-in definition

$$\texttt{NF}[l] = \texttt{NFQ}[l] \mid (l \circ \texttt{NF}[l]).$$

[3] The MATHEMATICA function definitions are tried top-down, and this guarantees a proper interpretation of the replacement operation.

Example 12. Consider the rule "perm" defined in Example 9. The rule NF["perm"] can be used to compute normal forms with respect to "perm". It is easy to see that the normal form of an unordered list L with respect to "perm" is unique and it coincides with the sorted version of L. Thus, the call

$$\texttt{ApplyRule}[L, \texttt{NF}[\texttt{"perm"}]]$$

will always compute the sorted version of list L. For instance

$$\texttt{ApplyRule}[\{3,1,4,2\}, \texttt{NF}[\texttt{"perm"}]]$$

yields the sorted list $\{1, 2, 3, 4\}$. □

3.7 Abstraction

If l_0 is a rule built from other rules, the call

$$\texttt{SetAlias}[l_0, l]$$

defines l as an alias to the composed rule l_0. Note the distinction between SetAlias[ ] and DeclareRule[ ]: the first argument of DeclareRule[ ] must be a concrete MATHEMATICA transformation rule, whereas the first argument of SetAlias[ ] must be a rule composition, i.e., an expression made of rule names and applications of the rule combinators described so far.

Example 13. For Example 9, the call

$$\texttt{SetAlias}[\texttt{Repeat}[\texttt{"perm"}, \texttt{"Id"}], \texttt{"sort"}]$$

declares a rule named "sort" whose applicative behavior coincides with that of the construct Repeat["perm", "Id"]. □

Declaring aliases makes compositions of rules easier and intuitive.

4 Illustrative Example

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 as basic ρLOG rules as follows:

$$\begin{array}{l}\texttt{DeclareRule}[\texttt{f}[\texttt{f}[x\_, y\_], z\_] :\longrightarrow \texttt{f}[x, \texttt{f}[y, z]], \texttt{"A"}];\\ \texttt{DeclareRule}[\texttt{f}[x\_, \texttt{e}] :\longrightarrow x, \texttt{"N"}];\\ \texttt{DeclareRule}[\texttt{f}[x\_, \texttt{i}[x\_]] :\longrightarrow \texttt{e}, \texttt{"I"}];\end{array}$$

The call

$$\texttt{SetAlias}[\texttt{"A"} \mid \texttt{"N"} \mid \texttt{"I"}, \texttt{"G"}];$$

defines "G" as an alias for the rule "A" | "N" | "I".

The rewrite relation induced by rules "A", "N", "I" can be programmed as

$$\texttt{RWRule}[\texttt{"G"}, \texttt{"Group"}, \texttt{Traversal} \to \texttt{"LeftOut"}];$$

The relation $\to_{\texttt{"Group"}}$ is terminating but not confluent, since $\texttt{f}[\texttt{f}[x,\texttt{e}],x] \to_{\texttt{"Group"}} \texttt{f}[x,x] \not\to_{\texttt{"Group"}}$, $\texttt{f}[\texttt{f}[x,\texttt{e}],x] \to_{\texttt{"Group"}} \texttt{f}[x,\texttt{f}[\texttt{e},x]] \not\to_{\texttt{"Group"}}$. Therefore, checking whether two terms s, t are joinable requires a systematic search for a term u such that $s \to^*_{\texttt{"Group"}} u$ and $t \to^*_{\texttt{"Group"}} u$.

A simple way to program this joinability test in ρLog is to declare:

$$\texttt{DeclareRule}[\texttt{eq}[x\_, x\_] :\to \texttt{True}, \texttt{"Eq"}];$$
$$\texttt{SetAlias}[\texttt{Until}[\texttt{"Eq"}, \texttt{"Group"}], \texttt{"Join"}];$$

Then the call:

$$\texttt{ApplyRule}[\texttt{eq}[s,t], \texttt{"Join"}]$$

will decide whether s and t are joinable or not by computing `True` if s and t are joinable, and $\texttt{eq}[s,t]$ otherwise. If we want to get more details about the existence or non-existence of a rewrite derivation which makes the terms s and t joinable, then 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 shows the notebook generated by the call

$$\texttt{ApplyRule}[\texttt{eq}[\texttt{f}[\texttt{f}[x,\texttt{e}],\texttt{i}[x]], \texttt{f}[\texttt{f}[\texttt{e},y]],\texttt{i}[y]], \texttt{"Join"}, \texttt{TraceStyle} \to \texttt{"Compact"}];$$

5 Deduction Trees

A deduction tree (*D-tree* for short) is intuitively a trace of a validity-check for the rule application query $\exists^? x : E \to_l x$, for some given input expression E and rule l. The construction of a D-tree proceeds by successively reducing the query $\exists^? x : E \to_l x$ 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:

$$\frac{S_1 \quad \ldots \quad S_n}{\exists^? x : E \to_l x} \tag{4}$$

where S_i $(1 \leq i \leq n)$ are either (a) queries of the form $\exists^? x : E_i \to_{l_i} x$, or (b) valid reductions of the form $E_i \to_l E'_i$, or (c) expressions of the form $E_i \to_{l'} E'_i \wedge \exists^? x : E'_i \to_{l_1} x$ where $E_i \to_{l_2} E'_i$ 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:

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

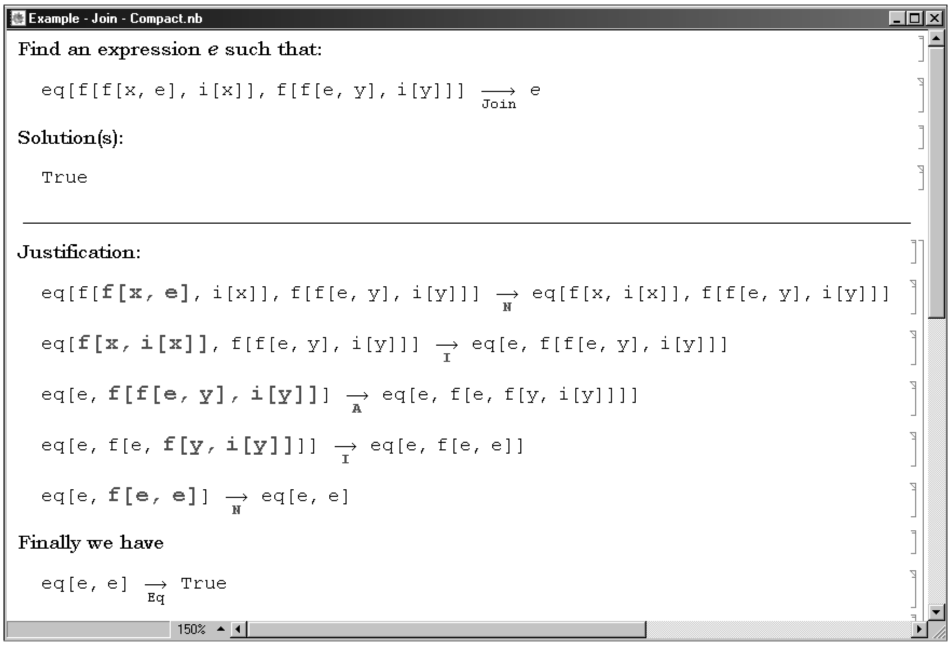

Fig. 1. "Compact" presentation of a rule-based deduction.

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

We say that a rule l is *elementary* if it is either basic or of the form $\texttt{NFQ}[l_1]$ with l_1 some rule.

The *reduct* of l, denoted by $\texttt{red}[l]$, is defined by[4]:

$$\texttt{red}[l] = \begin{cases} l & l \text{ elementary, or choice, or selection shift,} \\ & \text{or } l_1 \circ l_2 \text{ with } l_1 \text{ elementary or selection shift} \\ l_1 & l \rightsquigarrow \texttt{SetAlias}[l_1, l] \\ l_1 \circ (l_2 \circ l_3) & l \rightsquigarrow (l_1 \circ l_2) \circ l_3 \\ (l_1 \circ l_0) \mid \ldots \mid (l_n \circ l_0) & l \rightsquigarrow (l_1 \mid \ldots \mid l_n) \circ l_0 \\ \texttt{"Rw"} \circ (l_0 \mid \texttt{SEL}[l]) & l \rightsquigarrow \texttt{RWRule}[l_0, l, \texttt{Traversal} \to \texttt{"LeftOut"}] \\ \texttt{"Rw"} \circ (\texttt{SEL}[l] \mid l_0) & l \rightsquigarrow \texttt{RWRule}[l_0, l, \texttt{Traversal} \to \texttt{"LeftIn"}] \\ (l_1 \circ \texttt{Repeat}[l_1, l_2]) \mid l_2 & l \rightsquigarrow \texttt{Repeat}[l_1, l_2] \\ l_2 \mid (l_1 \circ \texttt{Until}[l_2, l_1]) & l \rightsquigarrow \texttt{Until}[l_2, l_1] \\ \texttt{red}[l_1] \circ l_2 & l \rightsquigarrow l_1 \circ l_2 \\ (l_0 \circ \texttt{NF}[l_0]) \mid \texttt{NFQ}[l_0] & l \rightsquigarrow \texttt{NF}[l_0] \end{cases}$$

[4] For lack of space, we will use the symbol '$\rightsquigarrow$' for the reading 'is defined by'.

A rule l is *reducible* if $\texttt{red}[l] \neq l$, and *irreducible* otherwise. Ambiguities in the definition of $\texttt{red}[\ ]$ are resolved by considering the first definition which becomes applicable during a top-down scan in the function definition.

It is a routine proof to show that if l is a reducible rule then

$$\forall x, y : x \rightarrow_l y \Leftrightarrow x \rightarrow_{\texttt{red}[l]} y.$$

Thus, the reduction of a rule l gives us an equivalent rule l' such that $\exists x : E \rightarrow_l x$ is valid iff $\exists x : E \rightarrow_{l'} x$ is also valid. We regard the result of the call $\texttt{red}[l]$ for an irreducible rule l as the definition assigned by ρLOG to l.

We define the relation $\succ$ by

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

$\succ$ is a well-founded relation because for any rule l, the string $\{l_n\}_{n \in \mathbb{N}}$ defined by

$$l_0 = l \text{ and } l_{n+1} = \texttt{red}[l_n]$$

has an irreducible element l_{n_0}.

Reducing a query $\exists^? x : E \rightarrow_l x$ to a simpler query is done by reducing the rule l as much as possible, using the $\texttt{red}[\ ]$ function described above, until we arrive at an irreducible rule which we try to apply. The termination of this reduction process is guaranteed by the well-founded property of $\succ$.

We proceed now with describing the inference rules.

1. If l is reducible then the corresponding inference rule is:

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

2. If l is an elementary rule 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 $E_1, \ \ldots, \ E_n$ $(n \geq 0)$ are all expressions such that $E \rightarrow_l E_i$.

3. If l is a selection shift rule then the corresponding inference rule is:

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

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

4. $l \equiv l_1 \circ l_2$ 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 $E_1, \ldots, E_n$ 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 $\{E_1, \ldots, E_n\} = \mathtt{shift}_{l_1}[E]$.

5. $l \equiv l_1 \mid \ldots \mid l_n$. The corresponding inference rule is

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

The definition of $\mathtt{red}[\,]$ guarantees that these inference rules cover all the possible situations for the shape of a query.

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

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

To avoid the generation of such infinite data 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:

$$
\begin{array}{rll}
T_0(E,l) & ::= \dfrac{}{\exists^? x : E \rightarrow_l x} & \\
T_{m+1}(E,l) & ::= \dfrac{E \rightarrow_l E_1 \quad \ldots \quad E \rightarrow_l E_n}{\exists^? x : E \rightarrow_l x} & l \text{ elementary} \\
\mid & \dfrac{T_m(E_1, \mathtt{r}_{l_1}) \quad \ldots \quad T_m(E_n', \mathtt{r}_{l_1})}{\exists^? x : E \rightarrow_l x} & l = l_1,\ l_1 \text{ selection shift} \\
\mid & \dfrac{T_{m+1}(E, \mathtt{red}[l])}{\exists^? x : E \rightarrow_l x} & l \text{ reducible} \\
\mid & \dfrac{T_m(E, l_1) \quad \ldots \quad T_m(E, l_n)}{\exists^? x : E \rightarrow_l x} & l \ = l_1 \mid \ldots \mid l_n \\
\mid & \dfrac{(E \rightarrow_{l_1} E_1) \wedge T_m(E_1, l_2) \quad \ldots \quad (E \rightarrow_{l_1} E_k) \wedge T_m(E_k, l_2)}{\exists^? x : E \rightarrow_l x} & l = l_1 \circ l_2,\ l_1 \text{ elementary} \\
\mid & \dfrac{T_m(E_1', \mathtt{r}_{l_1} \circ l_2) \quad \ldots \quad T_m(E_k', \mathtt{r}_{l_1} \circ l_2)}{\exists^? x : E \rightarrow_l x} & l = l_1 \circ l_2,\ l_1 \text{ selection shift rule}
\end{array}
$$

where $k, m \in \mathbb{N}$ and $\{E'_1, \ldots, E'_k\} = \mathtt{shift}_{l_1}[E]$. 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 (1) 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 (2) with $n \geq 1$.

failure D-tree is a D-tree with no leaf nodes computed by the application of inference rule of type (2) 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.

6 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. [13].

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:

$$\begin{aligned}
N ::= &\ \$\mathtt{SNODE}[\{E, \mathit{lexpr}, E'\}] \\
| &\ \$\mathtt{SNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
| &\ \$\mathtt{FNODE}[\{E, \mathit{lexpr}\}] \\
| &\ \$\mathtt{FNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
| &\ \$\mathtt{PNODE}[\{E, \mathit{lexpr}\}, N_1, \ldots, N_n] \\
| &\ \$\mathtt{EPNODE}[\{E, \mathit{lexpr}\}] \\
\mathit{lexpr} ::= &\ l \mid \{l_1, \ldots, l_n\} \mid \langle l, E, \mathit{lexpr}_1 \rangle.
\end{aligned}$$

In the sequel we describe the intended meaning of our proof objects. The procedure which generates them is described in Section 6.4.

6.1 Success objects

A success object is a proof object of the form $\texttt{\$SNODE}[\ldots]$. 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 $(E \rightarrow_{l_1} E') \wedge (\exists x : E' \rightarrow_{l_2} x)$.

The success objects which justify the validity of $\exists x : E \rightarrow_l x$ are of one of the following forms:

- $\texttt{\$SNODE}[\{E, l, E'\}]$ with l is elementary and $E \rightarrow_l E'$ is the only way to reduce E with $\rightarrow_l$,
- $\texttt{\$SNODE}[\{E, l\}, \texttt{\$SNODE}[\{E, l, E_1\}], \ldots, \texttt{\$SNODE}[\{E, l, E_n\}]]$ with l elementary, $n > 1$, and $E \rightarrow_l E_i$ $(1 \leq i \leq n)$ are all possibilities to reduce E with $\rightarrow_l$,
- $\texttt{\$SNODE}[\{E, \{l_1, \ldots, l_n\}, E'\}]$ where $l_1, \ldots, l_n$ are rules with the same applicative behavior and $\texttt{\$SNODE}[\{E, l_n, E'\}]$ justifies the validity of $\exists x : E \rightarrow_{l_n} x$,
- $\texttt{\$SNODE}[\{E, \{l_1, \ldots, l_n\}\}, N_1, \ldots, N_k]$ where $l_1, \ldots, l_n$ are rules with the same applicative behavior and $\texttt{\$SNODE}[\{E, l_n\}, N_1, \ldots, N_k]$ justifies the validity of $\exists x : E \rightarrow_{l_n} x$,
- $\texttt{\$SNODE}[\{E, l_1 \mid \ldots \mid l_n\}, N_1, \ldots, N_n]$ where N_i are proof objects for $\exists^? x : E \rightarrow_{l_i} x$ $(1 \leq i \leq n)$, and at least one N_i is a success object,
- $\texttt{\$SNODE}[\{E, l\}, N_1, \ldots, N_k]$ where l is a selection shift rule with $\texttt{shift}_l[E] = \{E_1, \ldots, E_k\}$, N_i are proof objects for $\exists^? x : E_i \rightarrow_{\mathtt{r}_l} x$ $(1 \leq i \leq k)$, and at least one N_i is a success object,
- $\texttt{\$SNODE}[\{E, l_1 \circ l_2\}, N_1, \ldots, N_k]$ where l_1 is elementary, $E \rightarrow_{l_1} E_i$ $(1 \leq i \leq k)$ are all possible ways to reduce E with $\rightarrow_{l_1}$, N_i are proof objects for the logical conjunction $E \rightarrow_{l_1} E_i \wedge \exists^? x : E_i \rightarrow_{l_2} x$ $(1 \leq i \leq k)$, and at least one N_i is a success object,
- $\texttt{\$SNODE}[\{E, l_1 \circ l_2\}, N_1, \ldots, N_k]$ where l_1 is a selection shift rule, $\texttt{shift}_{l_1}[E] = \{E_1, \ldots, E_k\}$, N_i are proof objects for $\exists^? x : E_i \rightarrow_{\mathtt{r}_{l_1} \circ l_2} x$ $(1 \leq i \leq k)$, and at least one N_i is a success object.

A success object for the logical conjunction $E \rightarrow_{l_1} E' \wedge \exists x : E' \rightarrow_{l_2} x$ is of one of the forms:

- $\texttt{\$SNODE}[\{E, \langle l_1, E', \mathit{lexpr} \rangle, E''\}]$ where $\texttt{\$SNODE}[\{E', \mathit{lexpr}, E''\}]$ justifies the validity of $\exists x : E' \rightarrow_l x$,
- $\texttt{\$SNODE}[\{E, \langle l_1, E', \mathit{lexpr} \rangle\}, N_1, \ldots, N_k]$ where $\texttt{\$SNODE}[\{E', \mathit{lexpr}\}, N_1, \ldots, N_k]$ justifies the validity of $\exists x : E' \rightarrow_l x$.

6.2 Failure objects

A failure object is a proof object of the form $\texttt{\$FNODE}[\ldots]$. 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 $(E \rightarrow_{l_1} E') \wedge (\nexists x : E' \rightarrow_{l_2} x)$.

The failure objects which justify the validity of $\nexists x : E \rightarrow_l x$ are of one of the following forms:

- $\texttt{\$FNODE}[\{E, l\}]$ with l is elementary and $E \not\rightarrow_l$,

– $\mathtt{\$FNODE}[\{E, \{l_1, \ldots, l_n\}\}, N_1, \ldots, N_k]$ where $l_1, \ldots, l_n$ are rules with the same applicative behavior and $\mathtt{\$FNODE}[\{E, l_n\}, N_1, \ldots, N_k]$ justifies the validity of $\nexists x : E \rightarrow_{l_n} x$,
– $\mathtt{\$FNODE}[\{E, l_1 \mid \ldots \mid l_n\}, N_1, \ldots, N_n]$ where each N_i is a failure object which justifies that $\nexists x : E \rightarrow_{l_i} x$,
– $\mathtt{\$FNODE}[\{E, l\}, N_1, \ldots, N_k]$ where l is a selection shift rule with $\mathtt{shift}_l[E] = \{E_1, \ldots, E_k\}$ and N_i are failure objects which justify that $\nexists x : E_i \rightarrow_{\mathtt{r}_l} x$.
– $\mathtt{\$FNODE}[\{E, l_1 \circ l_2\}, N_1, \ldots, N_k]$ where l_1 is elementary, $E \rightarrow_{l_1} E_i$ $(1 \leq i \leq k)$ are all possible ways to reduce E with $\rightarrow_{l_1}$, and N_i are failure objects which justify that $E \rightarrow_{l_1} E_i \wedge \nexists x : E_i \rightarrow_{l_2} x$ $(1 \leq i \leq k)$,
– $\mathtt{\$FNODE}[\{E, l_1 \circ l_2\}, N_1, \ldots, N_k]$ where l_1 is a selection shift rule, $\mathtt{shift}_{l_1}[E] = \{E_1, \ldots, E_k\}$, and N_i are failure objects which justify that $\nexists x : E_i \rightarrow_{\mathtt{r}_{l_1} \circ l_2} x$.

A failure object for the logical conjunction $E \rightarrow_{l_1} E' \wedge \nexists x : E' \rightarrow_{l_2} x$ is of the form $\mathtt{\$FNODE}[\{E, \langle l_1, E', \mathit{lexpr}\rangle\}]$ where $\mathtt{\$FNODE}[\{E', \mathit{lexpr}\}]$ justifies the validity of $\nexists x : E' \rightarrow_l x$.

6.3 Pending objects

A pending object is a proof object of the form $\mathtt{\$EPNODE}[\ldots]$ or $\mathtt{\$PNODE}[\ldots]$. Pending objects of the form $\mathtt{\$EPNODE}[\ldots]$ 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 $\exists x : E \rightarrow_l x$.

6.4 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:

1. If $T_m \equiv \dfrac{}{\exists^? x : E \rightarrow_l x}$ with $m < d$ then $\langle\!\langle T_m\rangle\!\rangle = \mathtt{\$FNODE}[\{E, l\}]$.
2. 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 $\langle\!\langle T_m\rangle\!\rangle = \mathtt{\$EPNODE}[\{E, l\}]$.
3. 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

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

4. Otherwise, if there exists a sequence of partial D-trees $T^1, \ldots, T^n$ of depth most m such that
$T_m = T^1(E, l)$, $l_{i+1} = \mathtt{red}[l_i]$, $T^i(E, l_i) = \dfrac{T^{i+1}(E, \mathtt{red}[l_i])}{\exists^? x : E \to_{l_i} x}$ for $1 \le i < n$ and $T^n(E, l_n) \equiv \dfrac{T^{n,1} \quad \ldots \quad T^{n,k}}{\exists^? x : E \to_{l_n} x}$ 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
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$SNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\!\langle T^{n,1} \rangle\!\rangle, \ldots, \langle\!\langle T^{n,k} \rangle\!\rangle]$$
   – if all $T^{n,1}, \ldots, T^{n,k}$ are failure D-trees then
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$FNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\!\langle T^{n,1} \rangle\!\rangle, \ldots, \langle\!\langle T^{n,k} \rangle\!\rangle]$$
   – 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
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$PNODE}[\{E, \{l_1, \ldots, l_n\}\}, \langle\!\langle T^{n,1} \rangle\!\rangle, \ldots, \langle\!\langle T^{n,k} \rangle\!\rangle].$$
5. Otherwise, if $T_m \equiv \dfrac{T^1 \quad \ldots \quad T^k}{\exists^? x : E \to_l x}$ with T^i of depth most $m-1$ for all $1 \le i \le k$ then
   – if there is $i \in \{1, \ldots, k\}$ such that T^i is a success D-tree then
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$SNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$
   – if all $T^1, \ldots, T^k$ are failure D-trees then
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$FNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$
   – 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
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$PNODE}[\{E, l\}, \langle\!\langle T^1 \rangle\!\rangle, \ldots, \langle\!\langle T^k \rangle\!\rangle]$$
6. Otherwise, $T_m \equiv \dfrac{(E \to_{l_1} E_1) \wedge T^1 \quad \ldots \quad (E \to_{l_1} E_k) \wedge T^k}{\exists^? x : E \to_{l_1 \circ l_2} x}$ where T^i are partial D-trees of depth at most $m-1$ for $\exists^? x : E_i \to_{l_2} x$, for all $1 \le i \le k$.
   For this situation we will make use of the function $\mathtt{Annotate}[E, l, N]$ which is defined for an expression E, rule l and proof object N as follows:
   - $\mathtt{\$SNODE}[\{E, \langle l, E_1, l' \rangle, E_2\}]$ if $N = \mathtt{\$SNODE}[\{E_1, l', E_2\}]$,
   - $n[\{E, \langle l, E_1, \mathit{lexpr} \rangle\}, N_1, \ldots, N_k]$ if $N = n[\{E_1, \mathit{lexpr}\}, N_1, \ldots, N_k]$ with $n \in \{\mathtt{\$SNODE}, \mathtt{\$FNODE}, \mathtt{\$PNODE}\}$.

   In the case $k = 0$, meaning that $E \not\to_{l_1}$, we have
   $$\langle\!\langle T_m \rangle\!\rangle = \mathtt{\$FNODE}[\{E, l_1 \circ l_2\}]$$
   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

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

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

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

– 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

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

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

Example 14. Consider the rule declarations

$$\begin{array}{l}
\mathtt{DeclareRule}[x_\mathtt{Real}/; (x < 0) :\to x + 7, \mathtt{"f1"}];\\
\mathtt{DeclareRule}[x_\mathtt{Real}/; (x < 1) :\to x + 4, \mathtt{"f2"}];\\
\mathtt{DeclareRule}[x_\mathtt{Real}/; (x > 0) :\to x/2, \mathtt{"g"}];\\
\mathtt{SetAlias}[\mathtt{"f1"} \mid \mathtt{"f2"}, \mathtt{"f"}];\\
\mathtt{SetAlias}[\mathtt{"f"} \circ \mathtt{"g"}, \mathtt{"fg"}];
\end{array}$$

and the query $\exists^? x : -4.0 \to_{\mathtt{"fg"}} x$. The corresponding D-tree is $T(-4.0,\ \mathtt{"fg"})$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{(-4.0 \to_{\mathtt{"f1"}} 3.0) \wedge \cfrac{3.0 \to_{\mathtt{"g"}} 1.5}{\exists^? x : 3.0 \to_{\mathtt{"g"}} x}}{\exists^? x : -4.0 \to_{\mathtt{"f1"} \circ \mathtt{"g"}} x} \qquad \cfrac{(-4.0 \to_{\mathtt{"f2"}} 0.) \wedge \cfrac{}{\exists^? x : -0. \to_{\mathtt{"g"}} x}}{\exists^? x : -4.0 \to_{\mathtt{"f2"} \circ \mathtt{"g"}} x}}{\exists^? x : -4.0 \to_{(\mathtt{"f1"} \circ \mathtt{"g"}) \mid (\mathtt{"f2"} \circ \mathtt{"g"})} x}}{\exists^? x : -4.0 \to_{(\mathtt{"f1"} \mid \mathtt{"f2"}) \circ \mathtt{"g"}} x}}{\exists^? x : -4.0 \to_{\mathtt{"f"} \circ \mathtt{"g"}} x}}{\exists^? x : -4.0 \to_{\mathtt{"fg"}} x}$$

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:

$$T_1 = \frac{3.0 \to_{\mathtt{"g"}} 1.5}{\exists^? x : 3.0 \to_{\mathtt{"g"}} x} \quad \text{and} \quad T_2 = \frac{}{\exists^? x : 0. \to_{\mathtt{"g"}} x}$$

with the corresponding proof objects

$$\begin{array}{l}
\langle\langle T_1 \rangle\rangle = \mathtt{\$SNODE}[\{3.0,\ \mathtt{"g"},\ 1.5\}] \text{ and}\\
\langle\langle T_2 \rangle\rangle = \mathtt{\$FNODE}[\{0.,\ \mathtt{"g"}\}].
\end{array}$$

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

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

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 (6) of the encoding procedure.

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

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

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 (4), $\langle\!\langle T_5 \rangle\!\rangle$ gives us

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

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

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

generates a Mathematica notebook with a detailed description of the rule-based deduction encoded in $\langle\!\langle T \rangle\!\rangle$. This notebook is shown in Figure 2. □

7 Visualizing and Manipulating ρLog Proof Objects

Having implemented a data structure for storing a (partial) D-tree, we, of course, desire to see it 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 the proof object described in section 6. 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.

We have seen in the example presented in Section 1 how the rewrite mechanism of ρLog can be invoked. Triggering the different presentation styles is done via Mathematica's options mechanism ([14], 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 the eventually found solutions or the original expression if no solution was found.

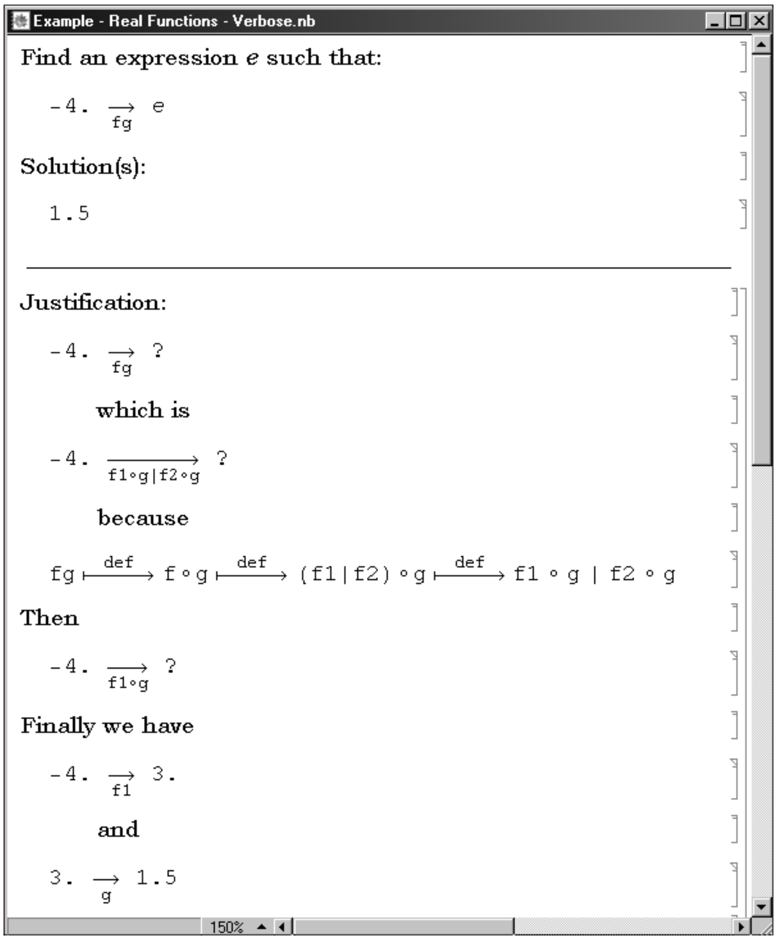

Fig. 2. `"Verbose"`-style presentation of a rule-based deduction.

- `"Object"` – choosing this value will cause the return of the proof object internal data structure. This can be very large, and inspecting it requires a clear understanding of how the data structure is defined (Section 6)
- `"Compact"` – this value of `TraceStyle` will generate a Mathematica notebook with a user friendly presentation of the (partial) D-tree encoded in the internal data structure for 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 unfolding the rules.
- `"Verbose"` – it is similar with the previous option value. The difference consists in the amount of information displayed for each step that the rewrite engine took in finding (or not) the solution.

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 tree 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.

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 $\exists^? x : E \rightarrow_l x$. For example, a call

$$\texttt{ApplyRuleList}[E, l,\ \texttt{TraceStyle} \rightarrow \texttt{"Compact"},\ \texttt{MaxSols} \rightarrow 3]$$

will start to insert pending objects in the proof object of the query $\exists^? x : E \rightarrow_l x$ as soon as it had found 3 expressions E_1, E_2, E_3 for which $E \rightarrow_l E_i$. It is easy that this situation is reached when we succeed to compute 3 success objects of type 1. (Section 6.4).

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

We have seen how ρLOG can be employed to compute proof objects and visualize. There is also a number of methods which can operate directly on proof objects.

A very useful capability is to further expand the partial D-tree encoded in a proof object, and ti obtain the proof object corresponding to the expanded partial D-tree. This can be done by invoking

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

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.

Another useful capability is to visualize the proof encapsulated in a given proof object *obj*. This is achieved by calling

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

The most important option is DetailLevel. The possible values of DetailLevel and their meaning in DisplayProof[ ] calls coincide with those of TraceStyle in ApplyRule[ ] calls.

For the same return value, obj, the user can invoke

$$\texttt{GetObjectSolutions}[obj]$$

to extract, from obj, the list of all expressions E_{sol} for which it is known that $E \rightarrow_l E_{sol}$. If obj encodes a failure or a pending D-tree then the call yields the empty list $\{\}$.

8 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 [11].

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.

In this paper, we have described how the system 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 [3–5], 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 [6, 9]. 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 [7, 8] 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/`

References

1. Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 90. North Holland, second edition, 1984.
2. Peter Borovansky, Horaţiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN: User Manual, January 27 2000.
3. Bruno Buchberger. Theorema: A short introduction. *Mathematica Journal*, 8(2):247–252, 2001.

4. 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.
5. Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuţa, and Daniela Văsaru. A survey of the THEOREMA project. In W. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. Stephen Wolfram. *The Mathematica Book*. Wolfram Media Inc. Champaign, Illinois, USA and Cambridge University Press, 1999.

A Pattern Matching in MATHEMATICA

MATHEMATICA is a language with powerful capabilities for symbolic and numeric computation. It supports functional programming via a matching mechanism based provides advanced features such as: sequence variables, alternative patterns, side conditions, a.s.o.

The most peculiar feature of MATHEMATICA for pattern matching is the concept of sequence variable. A sequence variable is a function parameter which can be instantiated with a (possibly empty) sequence of terms.

Example 15. The partial function defined by

`IntElem[{x___,y_Integer,z___}]:=y`

yields an integer element of a list `l` upon the call `IntElem[l]`. For instance, the call `IntElem[`$\{a, 1, x, s, 2\}$`]` can yield either 1 by computing the matcher $\{x \to \ulcorner a \urcorner,\ y \to 1,\ z \to \ulcorner x, s, 2 \urcorner\}$, or 2 by computing $\{x \to \ulcorner a, 1, x, s \urcorner,\ y \to 2,\ z \to \ulcorner\urcorner\}$. Note that, to aid the reading the matchers, we have written the bindings of sequence variables between $\ulcorner$ and $\urcorner$. □

The MATHEMATICA book [14] provides several convincing examples in favor of programming with sequence variables. It is important to note that programming with sequence variables relies on choosing a particular matcher during the evaluation of a function call. The interpreter of MATHEMATICA chooses the matcher which assigns the shortest possible lengths to the bindings of the first sequence variables that show up when traversing the pattern in a leftmost-innermost manner. In our example, the MATHEMATICA interpreter chooses the matcher $\{x \to \ulcorner a \urcorner,\ y \to 1,\ z \to \ulcorner x, s, 2 \urcorner\}$ because the binding $\ulcorner a \urcorner$ for x in the first matcher is shorter than the binding $\ulcorner a, 1, x, s \urcorner$ for x in the second matcher.

In the sequel we give a brief account of the programming capabilities with patterns of MATHEMATICA. Table 1 depicts the most common pattern constructs used in MATHEMATICA.

Pattern	Meaning
`_`	one term
`__`	sequence of 1 or more terms
`___`	sequence of 0 or more terms
`__h`	sequence of 1 or more terms, all of whose heads are `h`
`___h`	sequence of 0 or more terms, all of whose heads are `h`
`__?`*test*	sequence of 1 or more term which satisfy *test*
`___?`*test*	sequence of 0 or more term which satisfy *test*

Table 1. Patterns in *Mathematica*

A pattern *patt* may be named for later reference, e.g., we can write x : *patt* for a pattern *patt* named x. The separator : is usually omitted when *patt* starts with an underline. For example, we prefer to write `x_Real` and `y__?`*test* instead of `x:_Real` and `y:__?`*test.*

Another useful construct is *patt*/; *cond* which defines a pattern which matches iff *patt* matches and the evaluation of *cond* yields `True`.

Example 16. The pattern $\{x\_\_\_, y\_\_\}$ matches the list $\{1, 2\}$ in 2 possible ways: with the matcher $\theta_1 = \{x \to \ulcorner\urcorner, y \to \ulcorner 1, 2 \urcorner\}$ or with the matcher $\theta_2 = \{x \to$

$\ulcorner 1 \urcorner, y \rightarrow \ulcorner 2 \urcorner\}$. By contrast, the pattern $\{x\_\_\_, y\_\_\}/;\mathtt{Length}[\{x\}] < \mathtt{Length}[\{y\}]$ matches with $\{1,2\}$ only in one way, with matcher θ_1, because

$$(\mathtt{Length}[\{x\}] < \mathtt{Length}[\{y\}])/.\theta_1 = \mathtt{Length}[\{\}] < \mathtt{Length}[\{1,2\}] = 0 < 2 = \mathtt{True},$$
$$(\mathtt{Length}[\{x\}] < \mathtt{Length}[\{y\}])/.\theta_2 = \mathtt{Length}[\{1\}] < \mathtt{Length}[\{2\}] = 1 < 1 = \mathtt{False}.$$

$\square$

Finally, the construct $patt_1 \mid \ldots \mid patt_n$ defines a pattern which matches iff there exists a pattern $patt_i$ which matches.

We refer the interested reader to [14, Sect. 2.3] for a complete description of the MATHEMATICA patterns.