Johann Radon Institute for
Computational and Applied Mathematics
Austrian Academy of Sciences (ÖAW)

Rule-Based Programming with Mathematica

M. Marin, F. Piroi

RICAM-Report 2004-03

Rule-Based Programming with *Mathematica*

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

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

Abstract. Recent years have witnessed renewed developments of the rule-based programming style, addressing both its theoretical foundations and its practical implementations. New rule-based programming languages have emerged, and several practical applications have shown that rules are indeed a useful programming tool. We believe that *Mathematica* has the right basic ingredients for supporting rule-based programming efficiently. Because the main features of a true rule-based programming language are still missing, we developed a *Mathematica* package, ρLog, which enables advanced rule-based programming within *Mathematica*. We describe here the capabilities of ρLog and illustrate its usage with several examples.

1 Introduction

The rule-based programming style is currently experiencing a period of rapid growth with the rise of new concepts and systems that allow a better understanding and usage of this paradigm. New rule-based languages such as ASM, ASF+SDF [vdBDH$^+$01], Claire [CL96], ELAN [BKK$^+$96,BCD$^+$00], Maude [CDE$^+$01], and Stratego [Vis01] have emerged, and several practical applications have shown that rules are indeed a useful programming tool.

The rule-based programming style is characterized by repeated transformations of a data object such as a term, graph, proof situation, or constraint store. The transformations are described by rules which specify the calculation of an object that is intended to replace another object described by a pattern. Rules can have further conditions attached that restrict their applicability, and can be combined into strategies which control the order and number of their application. The rule-based programming style is built on concepts which are central to important chapters of theoretical computer science and their practical implementations. Namely, term rewriting strategies are used to describe the meaning of programming languages, to perform computations, or to perform deductions in situations where the inference rules of a logic are implemented as transformation rules.

The *Mathematica* language is especially suitable for rule-based programming, since its core engine is based on a higher-order rewrite logic. Unfortunately, *Mathematica* lacks the core features of a rule-based programming language, as outlined by the community involved in this kind of research. To justify our work, we start by giving a brief account to the main capabilities of rule-based programming in *Mathematica*, and point out the desirable but missing programming features.

Rules are specifications of partially defined and possibly non-deterministic computations which describe the calculation of a new object from another object described by a pattern.

⋆ Mircea Marin has been supported by the Austrian Academy of Sciences.
⋆⋆ Florina Piroi has been supported by the Austrian Science Foundation FWF under the project SFB 1302.

A *Mathematica* transformation rule is an expression

$$\texttt{patt} :\rightarrow \texttt{Expr} \tag{1}$$

where `patt` is a *Mathematica* pattern, and `Expr` is an expression which describes the computation of the new object from the variable bindings obtained by matching `patt` with some input expression. Optionally, `patt` can have additional conditions which restrict matching (See [Wol99, Section 2.3].) The attempt to apply rule (1) to an input expression $\texttt{Expr}_0$ can have one of the following outcomes:

1. `patt` does not match with $\texttt{Expr}_0$; in this case, the rule application attempt fails,
2. `patt` matches with $\texttt{Expr}_0$; in this case, the rule application attempt succeeds and yields an object produced by evaluating `Expr` for the bindings of a matcher between `patt` and $\texttt{Expr}_0$.

In the sequel we will denote by $\rightarrow_r$ the reduction relation associated with a rule r:

$$\rightarrow_r = \{\langle \texttt{Expr}_1, \texttt{Expr}_2\rangle \mid \texttt{Expr}_2 \text{ is a possible result of applying rule } r \text{ to the input expression } \texttt{Expr}_1\}$$

We write $\texttt{Expr}_1 \rightarrow_r \texttt{Expr}_2$ whenever $\langle \texttt{Expr}_1, \texttt{Expr}_2\rangle \in \rightarrow_r$, and $\texttt{Expr} \not\rightarrow_r$ if

$$\forall_{\{\texttt{Expr}_1, \texttt{Expr}_2\}} (\texttt{Expr}_1 \rightarrow_r \texttt{Expr}_2) \Rightarrow \texttt{Expr} \neq \texttt{Expr}_1.$$

The *Mathematica* call

$$\text{Replace}[\texttt{Expr}_1, \texttt{patt} :\rightarrow \texttt{Expr}]$$

computes the possibly empty list of instances $\texttt{Expr}/.\theta$, where θ ranges over the matchers between $\texttt{Expr}_1$ and `patt`.

Example 1. The rule

$$\texttt{r1} := \{x\_\_\texttt{Integer}, y\_\texttt{Integer}, z\_\_\texttt{Integer}, t\_\texttt{Integer}, u\_\_\texttt{Integer}\}/; y > t :\rightarrow \{x, t, z, y, u\}$$

specifies a computation which is partially defined (it may be applied only on lists of integers) and non-deterministic. The non-determinism stems from the fact that matching against the pattern of the rule is not unique. The call:

$$\texttt{ReplaceList}[\{3, 1, 4, 2\}, \texttt{r1}]$$
$$\blacktriangleright \{\{1, 3, 4, 2\}, \{2, 1, 4, 3\}, \{3, 1, 2, 4\}\}$$

shows that there are three possibilities to match

$$\{x\_\_\text{Integer}, y\_\text{Integer}, z\_\_\text{Integer}, \text{t\_Integer}, \text{u\_\_Integer}\}/; y > t$$

with {3,1,4,2}, whereas the call:

$$\texttt{ReplaceList}[\{1, 2, 3, 4\}, \texttt{r1}]$$
$$\blacktriangleright \{\}$$

illustrates a situation when no matcher exists, and thus the rule application attempt fails: $\{1,2,3,4\} \not\rightarrow_{\texttt{r1}}$.

Still, *Mathematica* lacks most of the features which are desired for rule-based programming. Such features include:

1. the possibility to program compositions of reductions, alternative choices, reflexive-transitive closures, etc.
2. a built-in search mechanism to decide the existence of derivations $\texttt{Expr}_1 \rightarrow_{\texttt{rr}} \texttt{Expr}_2$, where $\texttt{Expr}_1$, $\texttt{Expr}_2$ are given expression schemata (patterns), and $\rightarrow_{\texttt{rr}}$ is a specification of a sequence of rule reduction steps. Typically, the specification of $\rightarrow_{\texttt{rr}}$ is built with the operators mentioned before (composition, choices, etc.)
3. the possibility to generate proofs which justify the existence or non-existence of such reduction derivations.

In order to fill in the lack of support for rule-based programming in *Mathematica*, we have implemented the package ρLog. The main capabilities of our package are the following:

- concise means to express the basic computational steps of an intended rule application as basic rules. These features are inherited from *Mathematica*, whose computational engine is based on a higher-order rewrite logic and with advanced support for symbolic and numeric computing;
- programming constructs, such as conditional basic rules and rule combinators, which make possible to structure large specifications of rules;
- built-in search mechanism to answer queries of the form $\exists_{\{R_1,\ldots,R_k\}} \texttt{Expr}_1 \rightarrow_r \texttt{Expr}_2$ where $\texttt{Expr}_1$ is a ground expression, r is the identifier of a rule, and $\texttt{Expr}_2$ is a *Mathematica* pattern whose variables are named $R_1, \ldots, R_k$ (see Section 2.2);
- support for generating proof objects, i.e., certificates that justify the correctness of the answer provided by ρLog to a query, and
- visualization tools for proof objects, which enable to analyze the deduction derivations of ρLog in a natural language formulation and at various levels of detail.

The plan of the paper is as follows. First, we describe the main programming capabilities of our package: programming basic rules, building specifications of more complex rules (also called *strategies*) via a number of built-in rule combinators, and the possibility to generate human-readable proofs for the queries submitted to the system. Next, we illustrate the capabilities of ρLog with a number of interesting examples, and indicate a number of applications which were successfully implemented with ρLog. In the last section we draw some conclusions.

2 Programming principles of ρLog

The main programming concept of ρLog is the notion of *rule*. Each rule is identified by a name. The most elementary rules are the *basic rules*. These are the rules which are declared and named by the user via DeclareRule[...] calls. The names of the basic rules are assumed to be *Mathematica* strings (see next subsection). We can build up more complex rules from the rules defined so far, via rule combinators which operate on rule names.

Formally, a rule name is an element of the following syntactic domain:

`name` ::=	
s	(∗ `Mathematica string which identifies a basic rule` ∗)
$\texttt{name}_1 \circ \texttt{name}_2$	(∗ `sequentiality` ∗)
$\texttt{Fst}[\texttt{name}_1, \ldots, \texttt{name}_n]$	(∗ `leftmost commitment` ∗)
`NF`[`name`]	(∗ `normalization` ∗)
$\texttt{name}_1 \mid \ldots \mid \texttt{name}_n$	(∗ `choice` ∗)
$\texttt{Repeat}[\texttt{name}_1, \texttt{name}_2]$	(∗ `repetition` ∗)
$\texttt{Until}[\texttt{name}_2, \texttt{name}_1]$	(∗ `repetition` ∗)

2.1 Rule combinators

ρLog provides six combinators for programming strategies: $\circ$, `Fst`, `NF`, $\mid$, `Repeat`, and `Until`. The meanings of these combinators are:

$\rightarrow_{\texttt{name}_1 \circ \texttt{name}_2} = \rightarrow_{\texttt{name}_1} \circ \rightarrow_{\texttt{name}_2}$,
$\rightarrow_{\texttt{NF}[\texttt{name}]} = \rightarrow^*_{\texttt{name}} \cap \{\langle \texttt{Expr}, \texttt{Expr} \rangle \mid \text{Expr} \not\rightarrow_{\texttt{name}}\}$,
$\rightarrow_{\texttt{Repeat}[\texttt{name}_1,\ \texttt{name}_2]} = \rightarrow_{\texttt{Until}[\texttt{name}_2,\ \texttt{name}_1]} = \rightarrow^*_{\texttt{name}_1} \circ \rightarrow_{\texttt{name}_2}$,
$\rightarrow_{\texttt{name}_1 \mid \ldots \mid \texttt{name}_n} = \rightarrow_{\texttt{name}_1} \cup \ldots \cup \rightarrow_{\texttt{name}_n}$,
$\rightarrow_{\texttt{Fst}[\texttt{name}_1,\ldots,\ \texttt{name}_n]} = \cup_{i=1}^{n} \{\langle \texttt{Expr}_1, \texttt{Expr}_2 \rangle \mid \texttt{Expr}_1 \rightarrow_{\texttt{name}_i} \texttt{Expr}_2$ and $\texttt{Expr}_1 \not\rightarrow_{\texttt{name}_j}$ with $j < i\}$.

where $\rightarrow^*_{\texttt{name}}$ is the reflexive and transitive closure of the relation $\rightarrow_{\texttt{name}}$. It is also possible to define aliases for complex rule names, in order to avoid writing them over and over again while programming. The call

`SetAlias`[*name*, *alias*]

declares *alias* as an alias of *name*. *alias* is assumed to be a *Mathematica* string.

2.2 Queries

ρLog is designed to answer queries of the form $\exists_{\{R_1,\ldots,R_q\}} \texttt{Expr} \rightarrow_{name} \texttt{patt}$ which expresses the following request:

Given an input expression `Expr` and a rule *name*
Decide whether there exists a substitution θ for the variables $R_1, \ldots, R_q$ of `patt`, such that the reducibility formula $\texttt{Expr} \rightarrow_{name} \texttt{patt}_\theta$ holds.

The expression $\texttt{patt}_\theta$ denotes the result of replacing the pattern occurrences of $R_1, \ldots, R_q$ in `patt` with their θ-values.
The ρLog call for the queries described above is

$\texttt{Query}[\texttt{Exists}[\{R_1, \ldots, R_q\}, \texttt{Expr} \rightarrow_{name} \texttt{patt}]]$

and yields either

1. `False` if the query has no solution, i.e., there is no substitution θ of the desired kind, or
2. the list $\{\texttt{True}, \tau\}$ where τ is a list $\{\langle v_1 \rangle, \ldots, \langle v_q \rangle\}$ such that $\theta = \{R_1 \rightarrow v_1, \ldots, R_q \rightarrow v_q\}$ is a substitution of the desired kind. Note that the expressions v_i may be sequences of terms because R_i can be sequence variables. In such situations we can not determine the values of v_i $(1 \leq i \leq q)$ from the list $\{v_1, \ldots, v_q\}$. To avoid this ambiguity, ρLog computes the list $\{\langle v_1 \rangle, \ldots, \langle v_q \rangle\}$.

The ρLog call `Query`[`Expr` $\rightarrow_{name}$ `patt`] abbreviates `Query`[`Exists`[{}, `Expr` $\rightarrow_{name}$ `patt`]].

Example 2. Suppose `"inv"` denotes the rule $x\_ \oplus x\_^\dagger \rightarrow 0$. Then, the query

`Query`[`Exists`[$\{R\}, a \oplus a^\dagger \rightarrow$`"inv"`$R\_$]]

yields $\{$`True`$, \langle 0 \rangle\}$ by computing $\theta = \{R \mapsto 0\}$. □

A weaker request is posed by the ρLog call

`ApplyRule`[`Expr`, *name*]

which yields either (a) `Expr` if `Expr` $\not\rightarrow_{name}$, or (b) an expression v for which `Expr` $\rightarrow_{name} v$.

Example 3. The call

`ApplyRule`[$a \oplus a^\dagger$, `"inv"`]

yields 0. □

The ρLog call

`QueryAll`[`Exists`[$\{R_1, \ldots, R_q\}$, `Expr` $\rightarrow_{name}$ `patt`]]

yields the list $\{\tau_1, \ldots, \tau_p\}$ of all lists $\tau_i = \{\langle v_1^i \rangle, \ldots, \langle v_q^i \rangle\}$ such that $\theta_i = \{R_1 \rightarrow v_1^i, \ldots, R_q \rightarrow v_q^i\}$ is a substitution of the desired kind.
A weaker request is posed by the ρLog call

`ApplyRuleList`[`Expr`, *name*]

which computes the list $\{v_1, \ldots, v_n\}$ of all values v_i for which `Expr` $\rightarrow_{name} v_i$ $(1 \leq i \leq n)$.

2.3 Basic rules

We distinguish two kinds of basic rules: unconditional basic rules and conditional basic rules. An *unconditional basic rule* corresponds to a *Mathematica* transformation rule; the only distinction is that, instead of referring to the rule via its value, we refer to it via a name. The name of a basic unconditional rule is assigned during the declaration:

`DeclareRule`[`patt`$\rightarrow_r$`Expr`]

which assigns the name r to the *Mathematica* transformation rule `patt` $:\!\rightarrow$ `Expr`.

Example 4 (Group theory). The ρLog declarations

```
<< RhoLog`RhoLog`

DeclareRule[f[x_, i[x_]] →"I" e];

DeclareRule[f[x_, e] →"N" x];

DeclareRule[f[f[x_, y_], z_] →"A" f[x, f[y, z]]];
```

define three rules which encode the axioms of a group with neutral element e, inverse operation i, and associative operation f. The call

```
Query[Exists[{R}, f[f[a, i[a]], e] →"N"∘"I" R_]]
▶ {True, {⟨e⟩}}
```

computes the derivation $f[f[a, i[a]], e] \rightarrow_{\texttt{"N"}} f[a, i[a]] \rightarrow_{\texttt{"I"}} e$ and binds R to e by matching $R\_$ with e. □

A *conditional basic rule* is a rule defined by a call of the form

$$\texttt{DeclareRule}[\texttt{ForAll}[\{R_1,\ldots,\ R_k\},\texttt{patt} \rightarrow_{name} \texttt{patt}_{n+1}/;(\texttt{cond}_1 \wedge \ldots \wedge \texttt{cond}_n)]] \quad (2)$$

where $\texttt{cond}_1,\ldots,\ \texttt{cond}_n$ are either *Mathematica* boolean formulas, or reducibility formulas of the form $e_i \rightarrow_{name_i} \texttt{patt}_i$. $R_1,\ldots,R_k$ are the names of the pattern variables which occur in $\texttt{patt}_i$ $(1 \leq i \leq n+1)$, and we assume that the following syntactic conditions hold:

(S1) $\texttt{patt}$ has no occurrences of $R_1,\ldots,R_k$,
(S2) each occurrence of an R_i as a symbol in the sequence $\texttt{cond}_1,\ldots,\texttt{cond}_n,\texttt{patt}_{n+1}$ is preceded by an occurrence of R_i as the name of a pattern,
(S3) no R_i can occur in $\texttt{patt}_j$ simultaneously as a pattern name and as a symbol.

The declaration (2) defines the rule *name* such that $e_1 \rightarrow_{name} e_2$ iff there exist a matcher θ between e_1 and $\texttt{patt}$, and a substitution γ for the variables $R_1,\ldots,R_k$, such that:

1. the logical conjunction $\texttt{cond}_1 \wedge \ldots \wedge \texttt{cond}_n$ holds if we instantiate their variables with the bindings of $\theta \cup \gamma$, and
2. e_2 is the result of evaluating $\texttt{patt}_{n+1}$ after we instantiate its variables with the bindings of $\theta \cup \gamma$.

Technically speaking, the syntactic conditions (S1)-(S3) enable the computability of e_2 such that $e_1 \rightarrow_{name} e_2$, without resort to unification, but by successive pattern matching attempts.

Example 5 (Propositional logic). We describe the inference rules of a system capable to answer queries expressed by sequents of the form

$$\{P_1,\ldots,P_p\} \vdash \{Q_1,\ldots,Q_q\}$$

where P_i and Q_j are propositions built with the logical connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication) and $\neg$ (negation). The intended meaning of such a sequent is: $\{P_1,\ldots,P_p\} \vdash \{Q_1,\ldots,Q_q\}$ is $\texttt{True}$ iff $P_1 \wedge \ldots \wedge P_p$ implies $Q_1 \vee \ldots \vee Q_q$. In order to reason in this fragment of logic, we can employ Gentzen's system **G'** [Gal85]. The system consists of the inference rules shown below.

$$\frac{}{\{\Gamma_1,A,\Delta_1\}\vdash\{\Gamma_2,A,\Delta_2\}}$$

$$\frac{\{\Gamma,A,B,\Delta\}\vdash R}{\{\Gamma,A\wedge B,\Delta\}\vdash R} \qquad \frac{L\vdash\{\Delta,A,\Lambda\} \quad L\vdash\{\Delta,B,\Lambda\}}{L\vdash\{\Delta,A\wedge B,\Lambda\}}$$

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

$$\frac{\{\Gamma,\Delta\}\vdash\{A,\Lambda\} \quad \{B,\Gamma,\Delta\}\vdash\{\Lambda\}}{\{\Gamma,A\Rightarrow B,\Delta\}\vdash\{\Lambda\}} \qquad \frac{\{A,\Gamma\}\vdash\{B,\Delta,\Lambda\}}{\{\Gamma\}\vdash\{\Delta,A\Rightarrow B,\Lambda\}}$$

$$\frac{\{\Gamma,\Delta\}\vdash\{A,\Lambda\}}{\{\Gamma,\neg A,\Delta\}\vdash\{\Lambda\}} \qquad \frac{\{A,\Gamma\}\vdash\{\Delta,\Lambda\}}{\{\Gamma\}\vdash\{\Delta,\neg A,\Lambda\}}$$

In ρLog, **G'** has the following straightforward rule-based implementation:

```
DeclareRule[{___, A_, ___} ⊢ {___, A_, ___} →"Ax" True];
DeclareRule[{Γ___, A_ ∧ B_, Δ___} ⊢ R_ →"∧-L" True/; ({Γ, A, B, Δ} ⊢ R →"PP")];
DeclareRule[ L_ ⊢ {Δ___, A_ ∧ B_, Λ___} →"∧-R"
            True/; ((L ⊢ {Δ, A, Λ} →"PP" True) ∧ (L ⊢ {Δ, B, Λ} →"PP" True))];
DeclareRule[ {Δ___, A_ ∨ B_, Λ___} ⊢ R_ →"∨-L"
            True/; ({Δ, A, Λ} ⊢ R →"PP" True) ∧ ({Δ, B, Λ} ⊢ R →"PP" True)];
DeclareRule[L_ ⊢ {Γ___, A_ ∨ B_, Δ___} →"∨-R" True/; ({L ⊢ {Γ, A, B, Δ}} →"PP" True)];
DeclareRule[ {Γ___, A_ ⇒ B_, Δ___} ⊢ {Λ} →"⇒-L"
            True/; (({Γ, Δ} ⊢ {A, Λ} →"PP" True) ∧ ({B, Γ, Δ} ⊢ {Λ} →"PP" True))];
DeclareRule[{Γ} ⊢ {Δ___, A_ ⇒ B_, Λ___} →"⇒-R" True/; ({A, Γ} ⊢ {B, Δ, Λ} →"PP" True)];
DeclareRule[{Γ___, ¬A_, Δ___} ⊢ {Λ___} →"¬-L" True/; ({Γ, Δ} ⊢ {A, Λ} →"PP" True)];
DeclareRule[{Γ___} ⊢ {Δ___, ¬A_, Λ___} →"¬-R" True/; ({A, Γ} ⊢ {Δ, Λ} →"PP" True)];

SetAlias["Ax" | "¬-L" | "¬-R" | "∧-L" | "∨-R" | "⇒-R" | "∧-R" | "∨-L" | "⇒-L", "PP"];
```

Note the mutual dependencies between the definition of `"PP"` and the definitions of its component rules.

Suppose we want to decide whether the propositional formula $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$ is valid or not. The ρLog call

```
Query[{} ⊢ {(P ⇒ Q) ⇒ (¬Q ⇒ ¬P)} →"PP" True]
▶ {True, {}}
```

confirms that the proposition holds.

We can ask ρLog to provide a justification of its answer: the call

```
Query[{} ⊢ {(P ⇒ Q) ⇒ (¬Q ⇒ ¬P)} →"PP" True, TraceStyle → "Notebook"]
```

instructs the system to generate a *Mathematica* notebook with a human-readable proof of the correctness of its answer. The notebook generated is shown in the Appendix. □

3 More examples

3.1 Term rewriting

Term rewriting strategies are commonly used to describe the evaluation of expressions in various programming languages. Usually, the value of an expression is given by the result of successively reducing the subexpressions of an expression until no more reductions are possible. Each reduction step of a subexpression is called a *rewrite step*, and the final result is called a *normal form* or a *value* of the initial expression.

Formally, the term rewriting relation $\rightarrow_{\texttt{"rw"}}$ induced by a rule named ℓ is defined as follows: $\texttt{Expr} \rightarrow_{\texttt{"rw"}} \texttt{Expr}'$ iff there exists a position p in `Expr` such that $\texttt{Expr}|_p \rightarrow_\ell \texttt{Expr}_1$ and $\texttt{Expr}' = \texttt{Expr}[\texttt{Expr}_1]_p$. The notations adopted here are standard: $\texttt{Expr}|_p$ is the expression which occurs at position p in `Expr`, and $\texttt{Expr}[\texttt{Expr}_1]_p$ is the result of inserting $\texttt{Expr}_1$ at position p in `Expr`.
With ρLog, we can program the rewrite relation induced by a rule ℓ quite easily:

```
DeclareRule[ForAll[{R}, f_[x___, y_, z___] →"rw1" f[x, R, z]/; (y →ℓ|"rw1" R_)]];
SetAlias[ℓ | "rw1", "rw"];
```

It is easy to prove that $\rightarrow_{\texttt{"rw"}}$ coincides with the rewrite relation induced by ℓ.

As a concrete example, let's consider the reduction relation `"G"` whose application corresponds with an application of one of the axioms of group theory (Example 4):

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

We name `"G*"` the rewrite rule induced by `"G"` and declare it as follows:

$$\texttt{DeclareRule}[\texttt{ForAll}[\{R\}, \texttt{f}\_[\texttt{x}\_\_\_, \texttt{y}\_, \texttt{z}\_\_\_] \to_{\texttt{"rw1"}} f[x, R, z]/; (y \to_{\texttt{"G"}\mid\texttt{"rw1"}} \texttt{R}\_)]];$$
$$\texttt{SetAlias}[\texttt{"G"} \mid \texttt{"rw1"}, \texttt{"G*"}];$$

The call

$$\texttt{Query}[\texttt{Exists}[\{R\}, f[e, f[y, i[y]]] \to_{\texttt{"G*"}} f[\texttt{R}\_, \_]]]$$
$$\blacktriangleright \{\texttt{True}, \{\langle e \rangle\}\}$$

binds R to e after performing the rewrite step $f[e, f[y, i[y]]] \to_{\texttt{"G*"}} f[e, e]$. This rewrite step is justified by the derivation tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f[y, i[y]] \to_{\texttt{"I"}} e}{f[y, i[y]] \to_{\texttt{"A"}\mid\texttt{"N"}\mid\texttt{"I"}} e}}{f[y, i[y]] \to_{\texttt{"G"}} e}}{f[y, i[y]] \to_{\texttt{"G"}\mid\texttt{"Go"}} e}}{f[e, f[y, i[y]]] \to_{\texttt{"Go"}} f[e, e]}}{f[e, f[y, i[y]]] \to_{\texttt{"G"}\mid\texttt{"Go"}} f[e, e]}}{f[e, f[y, i[y]]] \to_{\texttt{"G*"}} f[e, e]}$$

It is also possible to generate a human-readable presentation of the deduction tree computed by ρLog by calling

$$\texttt{Query}[\texttt{Exists}[\{R\}, f[e, f[y, i[y]]] \to_{\texttt{"G*"}} f[\texttt{R}\_, \_]],$$
$$\texttt{TraceStyle} \to \texttt{"Notebook"}]$$

To completely evaluate an expression to a normal form with respect to $\to_{\texttt{"G*"}}$, we can apply the rule `NF["G*"]`. For example, the call

$$\texttt{Query}[\texttt{Exists}[\{R\}, f[f[e, y], i[y]] \to_{\texttt{NF}[\texttt{"G*"}]} \texttt{R}\_]]$$
$$\blacktriangleright \{\texttt{True}, \{\langle e \rangle\}\}$$

binds R to e after computing the rewrite derivation

$$f[f[e, y], i[y]] \to_{\texttt{"G*"}} f[e, f[y, i[y]]] \to_{\texttt{"G*"}} f[e, e] \to_{\texttt{"G*"}} e.$$

Consider now the following problem: decide whether $f[f[x, e], i[x]]$ and $f[f[e, y], i[y]]$ are joinable with $\to_{\texttt{"G*"}}$, i.e., whether there exists a term u such that $f[f[x, e], i[x]] \to^*_{\texttt{"G*"}} u$ and $f[f[e, y], i[y]] \to^*_{\texttt{"G*"}} u$. This problem requires an exhaustive search of such an u, and can be programmed with ρLog as follows:

$$(* \texttt{ Joinability test } *)$$
$$\texttt{DeclareRule}[\{\texttt{x}\_, \texttt{x}\_\} \to_{\texttt{"J"}} \texttt{True}];$$
$$\texttt{Query}[\{f[f[x, e], i[x]], f[f[e, y], i[y]]\} \to_{\texttt{Until}[\texttt{"J"},\texttt{"G*"}]} \texttt{True}]$$
$$\blacktriangleright \{\texttt{True}, \{\}\}$$

The answer confirms that $f[f[x, e], i[x]]$ and $f[f[e, y], i[y]]$ are joinable with $\to_{\texttt{"G*"}}$. We can get a human-readable proof of this fact by calling

$$\texttt{Query}[\{f[f[x, e], i[x]], f[f[e, y], i[y]]\} \to_{\texttt{Until}[\texttt{"J"},\texttt{"G*"}]} \texttt{True}, \texttt{TraceStyle} \to \texttt{"Notebook"}]$$

3.2 Rewriting strategies

The rewrite relation induced by a rule has two sources of non-determinism: (1) the choice of the subterm to which the rule is applied, and (2) the choice of the matcher employed in the computation of the new subterm. To illustrate this, let's consider the rule declaration

$$\texttt{DeclareRule}[f[\_\_\_, \texttt{x}\_, \_\_\_] \rightarrow_{\texttt{"sel"}} x];$$

and the corresponding rewrite rule `"sel-rw"` defined by

$$\texttt{DeclareRule}[\texttt{ForAll}[\{R\}, h\_[\texttt{x}\_\_\_, \texttt{y}\_, \texttt{z}\_\_\_] \rightarrow_{\texttt{"sel-rw1"}} h[x, R, z]/; (y \rightarrow_{\texttt{"sel"}|\texttt{"sel-rw1"}} R\_)]];$$

$$\texttt{SetAlias}[\texttt{"sel"} \mid \texttt{"sel-rw1"}, \texttt{"sel-rw"}];$$

Then the term $f[a, f[b, c], d]$ can be rewritten by `"sel-rw"` in more than one way. There are two subterms to which `"sel"` is applicable: $f[a, f[b, c], d]$ and $f[b, c]$. Since $f[\_\_\_, x\_, \_\_\_]$ matches $f[a, f[b, c], d]$ in three ways, there are three alternative results of reducing this subterm. Similarly, the subterm $f[b, c]$ has two distinct matchers with $f[\_\_\_, x\_, \_\_\_]$, and therefore it can be reduced by `"sel"` either to b or to c. Thus, we have altogether five alternatives to rewrite $f[a, f[b, c], d]$ with `"sel-rw"`, as witnessed by the following call:

$$\texttt{ApplyRuleList}[f[a, f[b, c], d], \texttt{"sel-rw"}]$$
$$\blacktriangleright \{a, f[b, c], d, f[a, b, d], f[a, c, d]\}$$

It is often desirable to define a *rewriting strategy*, i.e., to impose some selection functions for the subterm and for the matcher employed in a rewrite step. The current implementation of ρLog provides the user with some means to control the choice of the subterm which is rewritten. More precisely, ρLog guarantees the following operational behavior: if $\texttt{ApplyRule}[\texttt{Expr}, \texttt{name}_1 \mid \ldots \mid \texttt{name}_n]$ yields $\texttt{Expr}'$, then either

1. $\texttt{Expr}' = \texttt{Expr}$ and $\texttt{Expr} \not\rightarrow_{\texttt{name}_1|\ldots|\texttt{name}_n}$, or else
2. $\texttt{Expr} \not\rightarrow_{\texttt{name}_i}$ and $\texttt{Expr} \rightarrow_{\texttt{name}_1|\ldots|\texttt{name}_{i-1}} \texttt{Expr}'$.

In other words, the attempt to apply a strategy $\texttt{name}_1 \mid \ldots \mid \texttt{name}_n$ proceeds from left to right, resuming with the application of the leftmost rule $\texttt{name}_i$ which is applicable.

It is also guaranteed that composition distributes over alternatives. This means that the methods `Query[ ]`, `ApplyRule[ ]`, `QueryAll[ ]`, and `ApplyRuleList[ ]` are insensitive to the replacements

$$(\texttt{name}_1 \mid \ldots \mid \texttt{name}_n) \circ \texttt{name} = (\texttt{name}_1 \circ \texttt{name}) \mid \ldots \mid (\texttt{name}_n \circ \texttt{name}),$$
$$\texttt{name} \circ (\texttt{name}_1 \mid \ldots \mid \texttt{name}_n) = (\texttt{name} \circ \texttt{name}_1) \mid \ldots \mid (\texttt{name} \circ \texttt{name}_n).$$

These properties allow us to program various rewriting strategies. It is easy to see that the rule `"G*"` performs leftmost outermost rewriting with respect to `"G"`. The leftmost innermost rewriting relation induced by `"G"` coincides with the reduction relation $\rightarrow_{\texttt{"Gi"}}$ defined by:

$$\texttt{DeclareRule}[\texttt{ForAll}[\{R\}, \texttt{f}\_[\texttt{x}\_\_\_, \texttt{y}\_, \texttt{z}\_\_\_] \rightarrow_{\texttt{"rwi"}} f[x, R, z]/; (y \rightarrow_{\texttt{"rwi"}|\texttt{"G"}} \texttt{R}\_)]];$$
$$\texttt{SetAlias}[\texttt{"rwi"} \mid \texttt{"G"}, \texttt{"Gi"}];$$

3.3 Pure λ-calculus

In λ-calculus [Bar84], a value is an expression which has no β-redexes outside λ-abstractions. We adopt the following syntax for λ-terms:

$$\begin{array}{ll} \texttt{term} ::= & \\ \quad x & (*\ \texttt{variable : a Mathematica symbol}\ *) \\ \quad \texttt{App}[\texttt{term}_1, \texttt{term}_2] & (*\ \texttt{application}\ *) \\ \quad \lambda[x, \texttt{term}] & (*\ \texttt{abstraction}\ *) \end{array}$$

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

$$\texttt{DeclareRule}[\texttt{App}[\lambda[\texttt{x\_}, \texttt{t1\_}], \texttt{t2\_}] \rightarrow_{\texttt{"}\beta\texttt{"}} \texttt{Repl}[\texttt{t1}, \{x, \texttt{t2}\}]];$$

The replacement operation Repl[t1,{x,t2}] can be programmed in *Mathematica* as follows:

$$\texttt{Clear}[\texttt{Repl}];$$

$$\texttt{Repl}[t : \lambda[\texttt{x\_}, \_], \{\texttt{x\_}, \_\}] := t;$$

$$\texttt{Repl}[\texttt{x\_}, \{\texttt{x\_}, \texttt{t\_}\}] := t;$$

$$\texttt{Repl}[\lambda[\texttt{x\_}, \texttt{t\_}], \sigma\_] := \lambda[x, \texttt{Repl}[t, \sigma]];$$

$$\texttt{Repl}[\texttt{App}[\texttt{t1\_}, \texttt{t2\_}], \sigma\_] := \texttt{App}[\texttt{Repl}[\texttt{t1}, \sigma], \texttt{Repl}[\texttt{t2}, \sigma]];$$

$$\texttt{Repl}[\texttt{t\_}, \_] := t;$$

The evaluation of a λ-term proceeds by repeated reductions of the redexes which are not inside abstractions. For this purpose, we declare the rules:

$$\texttt{DeclareRule}[\texttt{ForAll}[\{R\}, (\texttt{f\_}[\texttt{u\_\_\_}, \texttt{v\_}, \texttt{w\_\_\_}]/; (f =!= \lambda) \rightarrow_{\texttt{"rw2"}} f[u, R, w])/; (v \rightarrow_{\texttt{"}\beta\texttt{"}|\texttt{"rw2"}} R\_)]];$$
$$\texttt{SetAlias}[\texttt{NF}[\texttt{"}\beta\texttt{"} \mid \texttt{"rw2"}], \texttt{"}\lambda\texttt{Eval"}];$$

The call $\texttt{Query}[\texttt{Exists}[\{R\}, t \rightarrow_{\texttt{"}\lambda\texttt{Eval"}} R\_]]$ computes a pair $\{\texttt{True}, \{\langle v \rangle\}\}$ where v is a value for t. For example:

$$\texttt{Query}[\texttt{Exists}[\{R\}, \texttt{App}[\lambda[x, \texttt{App}[t, x]], \texttt{App}[\lambda[x, x], \lambda[y, \texttt{App}[z, y]]]] \rightarrow_{\texttt{"}\lambda\texttt{Eval"}} \texttt{R\_}]]$$
$$\blacktriangleright \{\texttt{True}, \{\langle \texttt{App}[t, \lambda[y, \texttt{App}[z, y]]] \rangle\}\}$$

3.4 Sorting

We illustrate how lists of integers can be sorted using bubble-sort. Each elementary step of bubble sort can be viewed as an application of the rule

$$\begin{array}{l} \texttt{DeclareRule}[\{\texttt{x\_\_\_Integer}, \texttt{y\_Integer}, \texttt{z\_\_\_Integer}, \texttt{u\_Integer}, \texttt{v\_\_\_Integer}\}/; y > u \\ \qquad \rightarrow_{\texttt{"perm"}} \{x, u, z, y, v\}]; \end{array}$$

It is easy to see that $\rightarrow_{\texttt{"perm"}}$ is a confluent and terminating reduction relation, and thus the $\rightarrow_{\texttt{"perm"}}$-normal form of any list of integers is uniquely defined. Moreover, this normal

form coincides with the sorted version of the list. The following ρLog call illustrates this fact:

```
ApplyRule[{9, 5, 7, 6, 0, 8}, NF["perm"]]
► {0, 5, 6, 7, 8, 9}
```

4 Concluding remarks

The main motivation for the development of ρLog was the desire to have a powerful tool to implement provers, solvers, and simplifiers in the frame of the *Theorema* project [Buc01]. Since many of the powerful provers and solvers integrated in *Theorema* have natural rule-based representations, we focused on the design of a rule-based system. We chose the language of *Mathematica* because *Theorema* is implemented in *Mathematica*. Also, the advanced pattern matching and rewriting capabilities of *Mathematica* made a very good starting point for our design and implementation.

A novel feature of ρLog as a rule-based programming language is the possibility to program with sequence variables and with function variables. The expressive power of sequence and function variables in the context of rewriting, as defined in *Mathematica*, was recognized by Buchberger in [Buc96]. These concepts add some unexpected programming capabilities to ρLog: term rewriting and some important rewrite strategies have straightforward specifications. Our current implementation relies on the enumeration strategy of matchers which is built into the *Mathematica* interpreter. In [MŢ03] we have shown that this enumeration strategy is sometimes undesirable, and proposed a number of programming constructs to control it. We leave the extension of ρLog with control mechanisms for pattern matching as a topic for future work.

With ρLog we wrote compact and efficient implementations of unification procedures in equational theories with sequence variables [Kut02], narrowing calculi for theories presented by various kinds of rewrite systems [MM02], and provers for classical and intuitionistic propositional logic. Some pointers to these implementations are available from our website

`www.ricam.oeaw.ac.at/people/page/marin/RhoLog/`

ρLog is distributed as a small *Mathematica* package (approx. 40 KB) which can be downloaded from the website mentioned above. This website is intended to provide links to the latest version of ρLog, and pointers to updated documentation and applications.

We believe that a deeper investigation of the capabilities provided by our calculus will help us to identify several new useful applications.

References

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 90. North Holland, second edition, 1984.

[BCD+00] P. Borovansky, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN: User Manual*. Loria, Nancy, France, v3.4 edition, January 27 2000.

[BKK+96] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4. Elsevier, 1996.

[Buc96] B. Buchberger. Mathematica as a rewrite language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, pages 1–13, Shonan Village Center, Japan, 1996. World Scientific.

[Buc01] B. Buchberger. Theorema: a short introduction. *Mathematica Journal*, 8(2):247–252, 2001.

[CL96] Y. Caseau and F. Laburthe. Claire: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming*, TU Berlin, 1996.

[CDE+01] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[Gal85] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., 1985.

[Kut02] T. Kutsia. Unification with Sequence Variables and Flexible Arity Symbols and its Extension with Pattern-Terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of Joint AISC'2002 and Calculemus'2002 Conferences*, volume 2385 of *LNAI*, pages 290–304, Marseille, France, July 1–5 2002. Springer Verlag.

[MM02] M. Marin and A. Middeldorp. New Completeness Results for Lazy Conditional Narrowing. In *Proceedings of 6h International Workshop on Unification (UNIF 2002)*, Copenhagen, Denmark, July 22-26 2002.

[MŢ03] M. Marin and D. Ţepeneu. Programming with Sequence Variables: The SEQUENTICA Package. In P. Mitic, P. Ramsden, and J. 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.

[vdBDH+01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[Vis01] E. Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357–362, 2001.

[Wol99] S. Wolfram. *The Mathematica Book*. Wolfram Media Inc. Champaign, Illinois, USA and Cambridge University Press, 1999.

A Proof generated for the query from Example 5

Before illustrating the proof for the query of Example 5, we give some general considerations about the general structure of a rule-based proof. We already mentioned that ρLog is designed to prove or disprove the validity of a formula

$$\exists_{\{R_1,\ldots,R_p\}} t_1 \rightarrow_\ell t_2$$

where t_1 is a ground term[1] and t_2 is a term which may contain the variables $R_1, \ldots, R_p$. The system handles a problem by successively decomposing it into subproblems of the form

$$\exists_{\{R_1,\ldots,R_n\}} cond_1 \wedge \ldots \wedge cond_m \tag{3}$$

where $\wedge$ is interpreted as logical conjunction and each $cond_i$ $(1 \leq i \leq m)$ is an atomic formula of one of the following kinds:

1. a reducibility formula $u_i \rightarrow_{\ell_i} v_i$, or
2. an equality between two terms u_i and v_i, depicted as $u_i \equiv v_i$, or
3. a *Mathematica* boolean test.

In ρLog, the decomposition of a problem of the form (3) into subproblem(s) is always driven by the syntactic structure of $cond_1$. The syntactic restrictions (S1)-(S3) imposed on the conditional basic rules (cf. Section 2.3) guarantee that all subproblems produced by ρLog have the first atomic formula $cond_1$ simple enough to be solved by pattern matching or by reducing it to a finite number of subproblems.

[1] That is, a term with no variable occurrences.

Due to lack of space, we omit a thorough description of the nice properties of ρLog proofs; instead, we illustrate how ρLog solves the problem

$$\{\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\}\} \rightarrow_{\texttt{"PP"}} \texttt{True}$$

by always looking to the syntactic structure of the leftmost formula of the current query. We recall from Example 5 that the request to solve this problem was submitted to ρLog by calling

$$\texttt{Query}[\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\} \rightarrow_{\texttt{"PP"}} \texttt{True}, \texttt{TraceStyle} \rightarrow \texttt{"Notebook"}]$$

The outcome of this call is a notebook with the following contents.

Decide
the validity of $\{\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\}\} \rightarrow_{\texttt{"PP"}} \texttt{True}$

Answer.
The formula holds.

Applied rules

$$\begin{aligned}
\textbf{Ax} :&= \{___, A_, ___\} \vdash \{___, A_, ___\} \rightarrow_{\texttt{"Ax"}} \texttt{True} \\
\textbf{PP} :&= \texttt{"Ax"} \mid \texttt{"}\neg\texttt{-L"} \mid \texttt{"}\neg\texttt{-R"} \mid \texttt{"}\wedge\texttt{-L"} \mid \texttt{"}\vee\texttt{-R"} \mid \texttt{"}\Rightarrow\texttt{-R"} \mid \texttt{"}\wedge\texttt{-R"} \mid \texttt{"}\vee\texttt{-L"} \mid \texttt{"}\Rightarrow\texttt{-L"} \\
\Rightarrow\textbf{-L} :&= \{\Gamma___, A_ \Rightarrow B_, \Delta___\} \vdash \{\Lambda\} \rightarrow_{\texttt{"}\Rightarrow\texttt{-L"}} \\
&\quad \texttt{True}/; ((\{\Gamma, \Delta\} \vdash \{A, \Lambda\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\{B, \Gamma, \Delta\} \vdash \{\Lambda\} \rightarrow_{\texttt{"PP"}} \texttt{True})); \\
\Rightarrow\textbf{-R} :&= \{\Gamma\} \vdash \{\Delta___, A_ \Rightarrow B_, \Lambda___\} \rightarrow_{\texttt{"}\Rightarrow\texttt{-R"}} \texttt{True}/; (\{A, \Gamma\} \vdash \{B, \Delta, \Lambda\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \\
\neg\textbf{-L} :&= \{\Gamma___, \neg A_, \Delta___\} \vdash \{\Lambda___\} \rightarrow_{\texttt{"}\neg\texttt{-L"}} \texttt{True}/; (\{\Gamma, \Delta\} \vdash \{A, \Lambda\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \\
\neg\textbf{-R} :&= \{\Gamma___\} \vdash \{\Delta___, \neg A_, \Lambda___\} \rightarrow_{\texttt{"}\neg\texttt{-R"}} \texttt{True}/; (\{A, \Gamma\} \vdash \{\Delta, \Lambda\} \rightarrow_{\texttt{"PP"}} \texttt{True})
\end{aligned}$$

Justification.
$\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\} \rightarrow_{\texttt{"PP"}} \texttt{True}$

By the definition of "PP", the problem reduces to

$\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\} \rightarrow_{\texttt{"Ax"}\mid\texttt{"}\neg\texttt{-L"}\mid\texttt{"}\neg\texttt{-R"}\mid\texttt{"}\wedge\texttt{-L"}\mid\texttt{"}\vee\texttt{-R"}\mid\texttt{"}\Rightarrow\texttt{-R"}\mid\texttt{"}\wedge\texttt{-R"}\mid\texttt{"}\vee\texttt{-L"}\mid\texttt{"}\Rightarrow\texttt{-L"}} \texttt{True}$

$\{\} \vdash \{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\} \rightarrow_{\texttt{"}\Rightarrow\texttt{R"}} \texttt{True}$

In the definition of the rule "⇒-R" we instantiate
$\langle\Gamma\rangle \rightarrow \langle\rangle$
$\langle\Delta\rangle \rightarrow \langle\rangle$
$\langle A\rangle \rightarrow \langle P \Rightarrow Q\rangle$
$\langle B\rangle \rightarrow \langle\neg Q \Rightarrow \neg P\rangle$
$\langle\Lambda\rangle \rightarrow \langle\rangle$

and the problem reduces to

$(\{P \Rightarrow Q\} \vdash \{\neg Q \Rightarrow \neg P\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge \texttt{True} \equiv \texttt{True})$

By the definition of "PP", the problem reduces to

$(\{P \Rightarrow Q\} \vdash \{\neg Q \Rightarrow \neg P\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{P \Rightarrow Q\} \vdash \{\neg Q \Rightarrow \neg P\} \rightarrow_{\texttt{"⇒-R"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"⇒-R"` we instantiate

$\langle \Gamma \rangle \rightarrow \langle P \Rightarrow Q \rangle$
$\langle \Delta \rangle \rightarrow \langle \rangle$
$\langle A \rangle \rightarrow \langle \neg Q \rangle$
$\langle B \rangle \rightarrow \langle \neg P \rangle$
$\langle \Lambda \rangle \rightarrow \langle \rangle$

and the problem reduces to

$(\{\neg Q, P \Rightarrow Q\} \vdash \{\neg P\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

By the definition of `"PP"`, the problem reduces to

$(\{\neg Q, P \Rightarrow Q\} \vdash \{\neg P\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{\neg Q, P \Rightarrow Q\} \vdash \{\neg P\} \rightarrow_{\texttt{"¬-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"¬-L"` we instantiate

$\langle \Gamma \rangle \rightarrow \langle \rangle$
$\langle A \rangle \rightarrow \langle Q \rangle$
$\langle \Delta \rangle \rightarrow \langle P \Rightarrow Q \rangle$
$\langle \Lambda \rangle \rightarrow \langle \neg P \rangle$

and the problem reduces to

$(\{P \Rightarrow Q\} \vdash \{Q, \neg P\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

By the definition of `"PP"`, the problem reduces to

$(\{P \Rightarrow Q\} \vdash \{Q, \neg P\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{P \Rightarrow Q\} \vdash \{Q, \neg P\} \rightarrow_{\texttt{"¬-R"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"¬-R"` we instantiate

$\langle \Gamma \rangle \rightarrow \langle P \Rightarrow Q \rangle$
$\langle \Delta \rangle \rightarrow \langle Q \rangle$
$\langle A \rangle \rightarrow \langle P \rangle$
$\langle \Lambda \rangle \rightarrow \langle \rangle$

and the problem reduces to

$(\{P, P \Rightarrow Q\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

By the definition of `"PP"`, the problem reduces to

$(\{P, P \Rightarrow Q\} \vdash \{Q\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{P, P \Rightarrow Q\} \vdash \{Q\} \rightarrow_{\texttt{"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"⇒-L"` we instantiate

$\langle \Gamma \rangle \rightarrow \langle P \rangle$
$\langle A \rangle \rightarrow \langle \neg P \rangle$
$\langle B \rangle \rightarrow \langle \neg Q \rangle$
$\langle \Delta \rangle \rightarrow \langle \rangle$
$\langle \Lambda \rangle \rightarrow \langle Q \rangle$

and the problem reduces to

$(\{P\} \vdash \{P, Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

By the definition of `"PP"`, the problem reduces to

$(\{P\} \vdash \{P, Q\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True})$
$\wedge(\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{P\} \vdash \{P, Q\} \rightarrow_{\texttt{"Ax"}} \texttt{True}) \wedge (\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"Ax"` we instantiate

$\langle A \rangle \rightarrow \langle P \rangle$ and the problem reduces to

$(\texttt{True} \equiv \texttt{True}) \wedge (\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

We eliminate the first trivial identity

and the problem reduces to

$(\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"PP"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

By the definition of `"PP"`, the problem reduces to

$(\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"Ax"|"¬-L"|"¬-R"|"∧-L"|"∨-R"|"⇒-R"|"∧-R"|"∨-L"|"⇒-L"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

$(\{Q, P\} \vdash \{Q\} \rightarrow_{\texttt{"Ax"}} \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

In the definition of the rule `"Ax"` we instantiate

$\langle A \rangle \rightarrow \langle Q \rangle$

and the problem reduces to

$(\texttt{True} \equiv \texttt{True}) \wedge (\texttt{True} \equiv \texttt{True})$

We eliminate the first trivial identity

and the problem reduces to

$\texttt{True} \equiv \texttt{True}$

which is valid because it has solution(s)

□