A Rule-based System for Computation and Deduction in Mathematica*

Mircea Marin[1], Besik Dundua[2], and Temur Kutsia[3]

[1] Faculty of Mathematics and Informatics
Department of Computer Science
West University of Timişoara, Timişoara, Romania
mircea.marin@e-uvt.ro
[2] Ilia Vekua Institute of Applied Mathematics
Ivane Javakhishvili Tbilisi State University, Tbilisi, Georgia
bdundua@gmail.com
[3] Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
kutsia@risc.jku.at

Abstract. ρLog is a system for rule-based programming implemented in Mathematica, a state-of-the-art system for computer algebra. It is based on the usage of (1) conditional rewrite rules to express both computation and deduction, and of (2) patterns with sequence variables, context variables, ordinary variables, and function variables, which enable natural and concise specifications beyond the expressive power of first-order logic. Rules can be labeled with various kinds of strategies, which control their application. Our implementation is based on a rewriting-based calculus proposed by us, called ρLog too. We describe the capabilities of our system, the underlying ρLog calculus and its main properties, and indicate some applications.

Keywords: Rewriting-based calculi · Strategies · Constrained Rewriting.

1 Introduction

In this paper we present our main contributions to the design and implementation of a system for rewriting-based declarative programming with rewriting strategies. The system is called ρLog, and is implemented as an add-on package on top of the rewriting and constraint solving capabilities of Mathematica [16]. It provides (1) a logical framework to reason in theories whose deduction rules can be specified by conditional rewrite rules of a very general kind, and (2) a semantic framework where computations are sequences of state transitions modelled as rewrite steps controlled by strategies.

ρLog has some outstanding capabilities:

* This work was supported by Shota Rustaveli National Science Foundation of Georgia under the grant no. FR17 439 and by the Austrian Science Fund (FWF) under project 28789-N32.

1. It has a specification language which, in addition to term variables, allows the use of sequence variables, function variables, and context variables. Sequence variables are placeholders for sequences of terms; function variables are placeholders for function symbols; and context variables are placeholders for functions of the form $\lambda x.t$ where t is a term with a single occurrence of the term variable x. These new kinds of variables enable natural and concise specifications beyond the expressive power of first-order logic. For example, solving equations involving sequence variables has applications in AI, knowledge management, rewriting, XML processing, and theorem proving.
2. It is based on a rewrite-based calculus proposed by us [14, 15] which integrates novel matching algorithms for the kinds of variables mentioned above, and is sound and complete if we impose some reasonable syntactic restrictions [12].
3. It is seamlessly integrated with the constraint solving capabilities of Mathematica, a state of the art system with nearly 5000 built-in functions covering all areas of technical computing. As a result, we can use ρLog to tackle a wide range of applications.
4. It can generate human-readable traces of its computations and deductions. In particular, this capability can be turned into a tool to generate human-readable proof certificates for deduction.

The paper is structured as follows. Section 2 contains a brief description of ρLog and its core concepts: programs, strategies and queries. In Sect. 2.1 we present the rewriting calculus implemented by us. Section 3 indicates some applications. Section 4 concludes.

2 The ρLog system

A program consists of rule declarations

$\texttt{DeclareRules}[rule_1, \ldots, rule_m]$

where $rule_1, \ldots, rule_m$ are labeled conditional rewrite rules of the form

$$t \to_{stg} t' / ; cond_1 \wedge \ldots \wedge cond_n \tag{1}$$

with the intended reading "t reduces to t' with strategy stg (notation $t \to_{stg} t'$) whenever $cond_1$ and ... and $cond_n$ hold." Such a rule is a partial definition for strategy stg. In the special case when $n = 0$, rules become unconditional: $t \to_{stg} t'$ has the intended reading "t is reducible to t' with strategy stg." Thus, the rewrite rules of ρLog differ from the usual rewrite rules of a rewrite theory because we label them with terms which we call *strategies*.

To illustrate how reduction works, consider the problem of extracting the smallest number from a non-empty list of numbers. We can achieve this by repeated application of the labeled rule

$\{\texttt{x}\_, \texttt{a}\_\_\_, \texttt{y}\_, \texttt{b}\_\_\_\} \to_{\text{"}\texttt{swap}\text{"}} \{\texttt{y}, \texttt{a}, \texttt{x}, \texttt{b}\} / ; (\texttt{x} > \texttt{y})$

until the smallest element is moved at first position in the list, followed by one application of the labeled rule

$\{\mathtt{x\_}, \mathtt{\_\_\_}\} \to_{\text{"}\mathtt{first}\text{"}} \mathtt{x}$

We can declare these labeled rewrite rules as follows:

```
DeclareRules[{x_, a___, y_, b___} →"swap" {y, a, x, b}/; (x > y),
             {x_, ___} →"first" x]
```

If `L` is a nonempty list of numbers and we pose the query

```
ApplyRule[NF["swap"]∘"first",L]
```

then the interpreter of ρLog returns as answer the term `t` which satisfies the reducibility formula $\mathtt{L} \to_{\mathtt{NF}[\text{"}\mathtt{swap}\text{"}]\circ\text{"}\mathtt{first}\text{"}} \mathtt{t}$, and this term is the smallest number in list `L`. Note the following peculiarities of our specification language:

1. `a, b, x, y` are variables. They are identified by suffixing their first occurrences[4] in the rule with `_` or `___`. The variables suffixed by `_` are either term variables or function variables, whereas those suffixed by `___` are sequence variables. Thus, `a, b` are sequence variables, and `x, y` are term variables. Like in Prolog, we allow the use of anonymous variables: `_` is a nameless placeholder for an element (a function symbol or a term), and `___` is a nameless placeholder for a sequence of terms.
2. `(x>y)` is a boolean condition that is properly interpreted by the constraint logic programming component (CLP) of ρLog.
3. `'NF'` and `'∘'` are predefined general-purpose strategy combinators: $t \to_{\mathtt{NF}[stg]} t'$ holds if t' is a normal form produced by repeated applications of $\to_{stg}$-reduction steps starting from t; and $t \to_{stg_1 \circ stg_2} t'$ holds if $t \to_{stg_1} t'' \to_{stg_2} t'$ holds for some intermediary term t''.

Sequence variables introduce nondeterminism in the reduction process. For example, there are 2 ways to reduce $\{4, 1, 5, 2\}$ with the labeled rule for `"swap"`: $\{4,1,5,2\} \to_{\text{"}\mathtt{swap}\text{"}} \{1,4,5,2\}$ with matcher $\{\mathtt{x} \to 4, \mathtt{a} \to \ulcorner\urcorner, \mathtt{y} \to 1, \mathtt{b} \to \ulcorner 5, 2 \urcorner\}$, $\{4,1,5,2\} \to_{\text{"}\mathtt{swap}\text{"}} \{2,1,5,4\}$ with matcher $\{\mathtt{x} \to 4, \mathtt{a} \to \ulcorner 1, 5 \urcorner, \mathtt{y} \to 2, \mathtt{b} \to \ulcorner\urcorner\}$. Here, $\ulcorner t_1, \ldots, t_n \urcorner$ represents the sequence of terms $t_1, \ldots, t_n$, in this order.

Sequence variables provide a simple way to traverse and process terms of any width. In contrast, context variables allow to traverse terms of any depth. For example, the parametric strategy `"rw"` defined by the rule

$\mathtt{C\_}^{\circ}[\mathtt{s\_}] \to_{\text{"}\mathtt{rw}\text{"}[\mathtt{r\_}]} \mathtt{C}^{\circ}[\mathtt{t}]/; (\mathtt{s} \to_{\mathtt{r}} \mathtt{t\_})$

specifies term rewriting with rules corresponding to parameter `r`. Here, $\mathtt{C}^{\circ}$ is a context variable, and the pattern $\mathtt{C}^{\circ}[\mathtt{s\_}]$ matches a term t in all ways which bind s to a subterm t' of t, and $\mathtt{C}^{\circ}$ to the context in which t' occurs. If `"r"` is the strategy defined by the rule

[4] By 'first occurrence' in (1), we mean first occurrence in the sequence of expressions (stg, t), $cond_1, \ldots, cond_n, t'$.

$\mathtt{f\_[f\_[x\_\_\_]]} \to_{\text{"}\mathtt{r}\text{"}} \mathtt{f[x]}$

then there are two ways to reduce the term $\mathtt{t} = \mathtt{a[a[b[b[1,2]]]]}$ with the labeled rule for strategy `"rw"["r"]`:

- $t \to_{\text{"}\mathtt{rw}\text{"}[\text{"}\mathtt{r}\text{"}]} t_1 = \mathtt{a[b[b[1,2]]]}$ with matcher $\{\mathtt{C}^\circ \to \lambda\mathtt{x.x}, \mathtt{s} \to t, \mathtt{t} \to t_1\}$ because $t \to_{\text{"}\mathtt{r}\text{"}} t_1$ with matcher $\{\mathtt{f} \to \mathtt{a}, \mathtt{x} \to \ulcorner \mathtt{b[b[1,2]]} \urcorner\}$, and
- $t \to_{\text{"}\mathtt{rw}\text{"}[\text{"}\mathtt{r}\text{"}]} t_2 = \mathtt{a[a[b[1,2]]]}$ with matcher $\{\mathtt{C}^\circ \to \lambda\mathtt{x.a[a[x]]}, \mathtt{s} \to \mathtt{b[b[1,2]]}, \mathtt{t} \to \mathtt{b[1,2]}\}$ because $\mathtt{b[b[1,2]]} \to_{\text{"}\mathtt{r}\text{"}} \mathtt{b[1,2]}$ with matcher $\{\mathtt{f} \to \mathtt{b}, \mathtt{x} \to \ulcorner \mathtt{1,2} \urcorner\}$.

This strategy definition illustrates another feature of ρLog: variable `f` matches a function symbol. In first-order logic, variables are placeholders for terms only, but some functional programming languages, including Mathematica, go beyond this limitation and allow variables to match function symbols too.

Matching with sequence variables and context variables is finitary [9, 8]. Algorithms which enumerate all finitely many matchers with terms containing such variables are described in [11, 10], and are used by the interpreter of ρLog.

The constraints in the conditional part of a rule are of three kinds: (1) reducibility formulas $t \to_{stg} t'$, (2) irreducibility formulas $t \not\to_{stg} t'$, (3) any boolean formulas expressed in the host language of Mathematica.

ρLog is designed to work with three kinds of **strategies**:

1. Atomic strategies, designated by a string identifier `sId`, and defined by one or more labeled rules of the form

$$t \to_{\mathtt{sId}} t'/; cond_1 \wedge \ldots \wedge cond_n.$$

   The following atomic strategies are predefined:
   `"Id"`: $t \to_{\text{"}\mathtt{Id}\text{"}} t'$, abbreviated $t \equiv t'$, holds if and only if t' matches t.
   `"elem"`: $l \to_{\text{"}\mathtt{elem}\text{"}} e$ holds if and only if e matches an element of list l.
   `"subset"`: $t \to_{\text{"}\mathtt{subset}\text{"}} t'$ holds if and only if t' matches a subset of set t.
2. Parametric strategies, defined by rules of the form

$$t \to_{sId[s_1,\ldots,s_m]} t'/; cond_1 \wedge \ldots \wedge cond_n$$

   where sId is the strategy identifier (a string) and $s_1, \ldots, s_m$ are its parameters. The parameters provide syntactic material to be used in the conditional part and result of the rule application.
3. Composite strategies, built from other strategies with strategy combinators. In ρLog, the following combinators are predefined:
   composition: $t \to_{stg_1 \circ stg_2} t'$ holds if $t \to_{stg_1} t'' \to_{stg_2} t'$ holds for some intermediary term t''
   choice: $t \to_{stg_1 | stg_2} t'$ holds if either $t \to_{stg_1} t'$ holds or $t \to_{stg_2} t'$ holds.
   repetition: $t \to_{stg^*} t'$ holds if either $t \to_{\text{"}\mathtt{Id}\text{"}} t'$ holds or there exist $u_1, \ldots, u_n$ such that $t \to_{stg} u_1 \to_{stg} \ldots \to_{stg} u_n \to_{stg} t'$ holds.
   first choice: $t \to_{\mathtt{Fst}[stg_1,\ldots,stg_n]} t'$ holds if there exists $1 \leq i \leq n$ such that $t \to_{stg_i} t'$ and $t \to_{stg_j} t'$ hold for all $i < j \leq n$.
   normalization: $t \to_{\mathtt{NF}[stg]} t'$ holds if $t \to_{stg^*} t'$ and $t' \not\to_{stg} \_$ hold.

Queries are formulas of the form $cond_1 \wedge \ldots \wedge cond_n$ where $cond_i$ must be of the same kind as the formulas from the conditional parts of rules. For $n = 0$ we obtain the vacuously true query, which we denote by $\top$.

We can submit to ρLog requests of the form

`Request[`$cond_1 \wedge \ldots \wedge cond_n$`]` or `RequestAll[`$cond_1 \wedge \ldots \wedge cond_n$`]`

They instruct the system to compute one (resp. all) substitution(s) for the variables in the formula $cond_1 \wedge \ldots \wedge cond_n$ for which it holds with respect to the current program. For example, if the current program contains the previous definition of the atomic strategy `"swap"`, then the request

`Request[`$\{4, 1, 5, 2\} \rightarrow_{\texttt{"swap"}} \texttt{x\_}$`]`

computes the substitution $\{\texttt{x} \rightarrow \{1, 4, 5, 2\}\}$, whereas the request

`RequestAll[`$\{4, 1, 5, 2\} \rightarrow_{\texttt{"swap"}} \texttt{x\_}$`]`

computes the set of all substitutions $\{\{\texttt{x} \rightarrow \{1, 4, 5, 2\}\}, \{\texttt{x} \rightarrow \{2, 1, 5, 4\}\}\}$ for which the reducibility formula $\{4, 1, 5, 2\} \rightarrow_{\texttt{"swap"}} \texttt{x}$ holds.

Another use of ρLog is to compute a reduct of a term with respect to a strategy. The request

`ApplyRule[`stg, t`]`

instructs ρLog to compute one (if any) reduct of t with respect to strategy stg, that is, a term t' such that the reducibility formula $t \rightarrow_{stg} t'$ holds. ρLog reports `"no solution found."` if there is no reduct of t with stg. ρLog can also be instructed to find *all* reducts of a term with respect to a strategy, with

`ApplyRuleList[`stg, t`]`

More information about ρLog can be found at

`http://staff.fmi.uvt.ro/~mircea.marin/rholog/`

2.1 The ρLog calculus

The ρLog system is designed to solve problems of the following kind:

Given a rewrite theory represented by a program P, and a query Q,
Find one, or all, substitutions σ for which formula $\sigma(Q)$ holds in the rewrite theory represented by P.

There is no effective method to solve this problem in full generality because syntactic unification of terms with our kinds of variables is infinitary [8]. We can avoid this difficulty by imposing syntactic restrictions on the structure of programs and queries, that guarantee the possibility to use matching instead of unification (we already mentioned that matching with terms containing the kinds of variables recognized by ρLog is finitary). Therefore, the ρLog calculus is designed to solve the following restricted version of the previous problem:

Given a rewrite theory represented by a deterministic program P, and a deterministic query Q,

Find one, or all, substitutions σ for which formula $\sigma(Q)$ holds in the rewrite theory represented by P.

Here, the notion of determinism is defined as follows: if $vars(E)$ denotes the set of variables in a syntactic construct E then

- If X is a set of variables and *cond* is a component formula of a query or of the conditional part of a rule, then *cond* is X-*deterministic* if either
  - *cond* is $t \rightarrow_{stg} t'$ or $t \nrightarrow_{stg} t'$ with $vars(t) \cup vars(stg) \subseteq X$, or
  - *cond* is a formula in which all predicate symbols are predefined in Mathematica, and $vars(cond) \subseteq X$.
- a rule $t \rightarrow_{stg} t'/; cond_1 \wedge \ldots \wedge cond_n$ is *deterministic* if $vars(t') \subseteq vars(t) \cup vars(stg) \cup \bigcup_{i=1}^{n} vars(cond_i)$ and, for all $1 \leq i \leq n$, $cond_i$ is X_i-deterministic where $X_i = vars(t) \cup vars(stg) \cup \bigcup_{j=1}^{i-1} vars(cond_j)$.
- a query $cond_1 \wedge \ldots \wedge cond_n$ is *deterministic* if, for all $1 \leq i \leq n$, $cond_i$ is X_i-deterministic where $X_i = \bigcup_{j=1}^{i-1} vars(cond_j)$.

Our calculus is, in essence, SLDNF-resolution with leftmost literal selection: every rule $t \rightarrow_{stg} t'/; cond_1 \wedge \ldots \wedge cond_n$ is logically equivalent with the clause

$$t \rightarrow_{stg} \mathtt{x}/; cond_1 \wedge \ldots \wedge cond_n \wedge (t' \equiv \mathtt{x\_})$$

where `x` is a fresh term variable and we can use resolution with respect to this equivalent clauses. The main difference from resolution in first-order logic is that, instead of using the auxiliary function $mgu(t, t')$ to compute a most general unifier of two terms, we use the auxiliary function $mcsm(t, t')$ which computes the finitely many matchers between a term t and a ground term t'.

Inference rules. The calculus has inference rules for the judgment $Q \rightsquigarrow_{\sigma} Q'$ with intended reading "query Q is reducible to Q' if substitution σ is performed." We use the notation

$$\frac{H_1 \quad \ldots \quad H_n}{Q \rightsquigarrow_{\sigma} Q'}$$

for an inference rule that allows us to conclude that $Q \rightsquigarrow_{\sigma} Q'$ holds if the assumptions $H_1, \ldots, H_n$ hold. Also, we write

- $Q_0 \rightsquigarrow_{\sigma}^{*} Q_n$, or just $Q_0 \rightsquigarrow^{*} Q_n$ whenever we succeed to infer a sequence of judgments $Q_0 \rightsquigarrow_{\sigma_1} Q_1 \ldots \rightsquigarrow_{\sigma_n} Q_n$, and σ is the restriction of substitution $\sigma_1 \ldots \sigma_n$ to $vars(Q_0)$.
- $Q_0 \not\rightsquigarrow^{*} Q_n$ whenever we finitely fail to infer that $Q_0 \rightsquigarrow^{*} Q_n$ holds.

The inference rules of ρLog are shown in Fig. 1. They differ from those of the initial ρLog calculus in some important ways:

1. The current version allows to have unrestricted Mathematica constraints in the specification of queries and conditional parts of rules. In this way, we achieve full integration of the CLP component of our system with the constraint solving capabilities of Mathematica.
   The inference rule of ρLog for such constraints is the last one from Fig. 1.
2. We introduced parametric strategies, that is, strategies with arguments that get instantiated during the reduction process. They enable natural and concise specifications of many kinds of rules, like those for strict and lazy evaluation. The inference rule of ρLog for this feature is the first one from Fig. 1: it matches both the left side and the strategy of the selected rule with the left side and strategy of the reducibility formula selected from the query.

$$\frac{(t_1' \to_{stg'} t_2'/; \bigwedge_{i=1}^{n} cond_i) \in P \quad \sigma \in mcsm((t_1', stg'), (t_1, stg))}{(t_1 \to_{stg} t_2) \wedge Q \rightsquigarrow_\sigma (\bigwedge_{i=1}^{n} cond_i \wedge (t_2' \equiv t_2) \wedge Q)}$$

$$\frac{\sigma \in mcsm(t', t)}{(t \equiv t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)} \qquad \frac{\sigma \in mcsm(t', t)}{(t \to_{\texttt{"Id"}} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{(t \to_{stg} t') \not\rightsquigarrow^* \top}{(t \not\to_{stg} t') \wedge Q \rightsquigarrow_{\{\}} Q}$$

$$\frac{(t \to_{stg} t') \rightsquigarrow_\sigma^* \top}{(t \to_{\texttt{Fst}[stg,\ldots]} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{(t \to_{stg_1} t') \not\rightsquigarrow^* \top}{(t \to_{\texttt{Fst}[stg_1, stg_2 \ldots, stg_n]} t') \wedge Q \rightsquigarrow_{\{\}} (t \to_{\texttt{Fst}[stg_2, \ldots, stg_n]} t') \wedge Q}$$

$$\frac{(t \to_{stg} \_) \rightsquigarrow^* \top}{(t \to_{\texttt{NF}[stg]} t') \wedge Q \rightsquigarrow_{\{\}} (t \to_{stg \circ \texttt{NF}[stg]} t') \wedge Q}$$

$$\frac{(t \to_{stg} \_) \not\rightsquigarrow^* \top \quad \sigma \in mcsm(t', t)}{(t \to_{\texttt{NF}[stg]} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{cond \text{ is a valid Mathematica formula}}{cond \wedge Q \rightsquigarrow_{\{\}} Q}$$

Fig. 1. The inference rules of the ρLog calculus

The proper interpretation of the other composite strategies and predefined parametric strategies is guaranteed by assuming that P contains defining rules for them. For example, the rules for the strategy combinators of ρLog are:

$\texttt{x\_} \to_{\texttt{s1\_}\circ\texttt{s2\_}} \texttt{z}/; (\texttt{x} \to_{\texttt{s1}} \texttt{y\_}) \wedge (\texttt{y} \to_{\texttt{s2}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{s1\_|s2\_}} \texttt{z}/; (\texttt{x} \to_{\texttt{s1}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{s1\_|s2\_}} \texttt{z}/; (\texttt{x} \to_{\texttt{s2}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{[s\_*]}} \texttt{y}/; (\texttt{x} \to_{\texttt{"Id"}|(\texttt{s}\circ\texttt{s*})} \texttt{y\_})$.

The set of answers computed by ρLog for Q in a rewrite theory represented by a program P is $Ans_P(Q) := \{\sigma \mid \text{there is an inference derivation } Q \rightsquigarrow_\sigma^* \top\}$.

Properties. ρLog is a sound calculus: for every $\sigma \in Ans_P(Q)$, the formula $\sigma(Q)$ holds in the rewrite theory presented by P. Unfortunately, ρLog is not complete: some substitutions that satisfy Q w.r.t. P may not be found because the leftmost selection strategy of query components is not fair, and some attempts to compute a derivation $Q \rightsquigarrow^* \top$ may run forever. The same phenomenon happens in most implementations of Prolog: SLDNF resolution with leftmost literal selection is incomplete for the same reason.

Implementations of logic programming languages usually adopt SLDNF resolution with leftmost literal selection, mainly for efficiency reasons; there are other, less efficient literal selection strategies, which preserve completeness. But for ρLog we have no other selection strategies, because this is the only way to avoid the problem of infinitary unification: by preserving determinism of queries.

3 Applications

3.1 Evaluation strategies

Suppose $P = \{t_i \to t'_i/; cond_{i,1} \wedge \ldots \wedge cond_{i,p_i} \mid 1 \leq i \leq n\}$ is a set of conditional rewrite rules that represent a functional program, and we wish to evaluate a term t with respect to P. A straightforward way to encode the rules of P in ρLog is to assign to all of them a common strategy identifier, say `"P"`, to indicate that they all belong to the same program. Thus, a ρLog program for P could be declared as follows:

`DeclareRules["P",`
$t_1 \to t'_1/; cond_{1,1} \wedge \ldots \wedge cond_{1,p_1}$,
...
$t_n \to t'_n/; cond_{n,1} \wedge \ldots \wedge cond_{n,p_n}$`]`

The main evaluation strategies in programming language theory are: eager (or strict) and lazy. When confined to expressions consisting of nested function calls, strict evaluation corresponds to innermost rewriting, and lazy evaluation corresponds to outermost rewriting with the conditional rewrite rules of P. If we adopt the small-step operational style, the value of a term is the normal form produced by derivations consisting of innermost (resp. outermost) rewrite steps.

In ρLog, the small-step operational specification of these evaluation strategies is straightforward:

strict evaluation: $t \to_{\text{"strict"[s]}} t'$ holds if if t' is obtained by reducing with s an innermost subterm of t. It can be defined by mutual recursion as follows:

```
DeclareRules[
    x_ →"strict"[s_] y/; (x →Fst["sAux"[s],s] y_),
    f_[as___, x_, bs___] →"sAux"[s_] f[as, y, bs]/; (x →"strict"[s] y_)]
```

lazy evaluation: $t \to_{\text{"lazy"[s]}} t'$ holds if t' is obtained by reducing with s an outermost subterm of t. It can be defined by mutual recursion as follows:

```
DeclareRules[
   x_ →"lazy"[s_] y/; (x →Fst[s,"lAux"[s]] y_),
   f_[as___, x_, bs___] →"lAux"[s_] f[as, y, bs]/; (x →"lazy"[s] y_)]
```

The strict value of a term t is returned by `ApplyRule[`t`,NF["strict"["P"]]]`, and the lazy value of term t is returned by `ApplyRule[`t`,NF["lazy"["P"]]]`.

3.2 Natural deduction

In logic and proof theory, natural deduction is a proof system whose inference rules are closely related to the "natural" way of reasoning. The general form of an inference rule is

$$\frac{J_1 \quad \ldots \quad J_n}{J}\ (name)$$

where $J_1, \ldots, J_n, J$ are judgments (that is, representations of something that is knowable), and *name* is the name of the inference rule. The judgments above the line are called *premise*s and that below the line is called *conclusion*. For example, Gentzen's proof system LK for natural deduction has inference rules for judgments of the form $L \vdash R$ where L and R are finite (possibly empty) sequences of formulas in first-order logic. Such a judgment is called sequent, and its intended reading is "If all formulas in L hold then at least one formula in R holds." For example, the inference rules of system LK that pertain to the propositional fragment of first-order logic are those shown in Fig. 2, where the metavariables L, L_1, L_2, R, R_1, R_2 denote sequences of formulas, and A, B denote formulas.

$$\frac{}{L_1, A, L_2 \vdash R_1, A, R_2}\ (I)$$

$$\frac{L_1, A, B, L_2 \vdash R}{L_1, A \wedge B, L_2 \vdash R}\ (\wedge L) \qquad \frac{L \vdash R_1, A, R_2 \quad L \vdash R_1, B, R_2}{L \vdash R_1, A \wedge B, R_2}\ (\wedge R)$$

$$\frac{L_1, A, L_2 \vdash R \quad L_1, B, L_2 \vdash R}{L_1, A \vee B, L_2 \vdash R}\ (\vee L) \qquad \frac{L \vdash R_1, A, B, R_2}{L \vdash R_1, A \vee B, R_2}\ (\vee R)$$

$$\frac{L_1, L_2 \vdash A, R \quad B, L_1, L_2 \vdash R}{L, A \Rightarrow B, L_2 \vdash R}\ (\Rightarrow L) \qquad \frac{A, L \vdash R_1, B, R_2}{L \vdash R_1, A \Rightarrow B, R_2}\ (\Rightarrow R)$$

$$\frac{L_1, L_2 \vdash A, R}{L_1, \neg A, L_2 \vdash R}\ (\neg L) \qquad \frac{A, L \vdash R_1, R_2}{L \vdash R_1, \neg A, R_2}\ (\neg R)$$

Fig. 2. System LK: Inference rules for propositional formulas.

System LK is sound and complete. This implies that $L \vdash R$ holds iff it can be derived from the above rules.

Often, we can translate an inference rule $\frac{J_1 \quad \ldots \quad J_n}{J}\ (name)$ into a corresponding rule of ρLog, as follows:

$J \rightarrow_{name}$ `True`$/; (J_1 \rightarrow_{stg}$ `True`$) \wedge \ldots \wedge (J_n \rightarrow_{stg}$ `True`$)$.

where strategy *stg* can be defined such that

> $J \rightarrow_{stg} \texttt{True}$ holds iff J can be derived using the inference rules of the proof system under consideration.

This translation technique works well for Gentzen's proof system illustrated in Fig. 2: We obtain the program consisting of the following nine rule declarations:

$$
\begin{aligned}
&\texttt{DeclareRules}[\\
&\quad \{\_\_\_, \texttt{A}\_, \_\_\_\} \vdash \{\_\_\_, \texttt{A}\_, \_\_\_\} \rightarrow_{\texttt{"I"}} \texttt{True},\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \wedge \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \texttt{R}\_ \rightarrow_{\texttt{"}\wedge\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{A}, \texttt{B}, \texttt{L2}\} \vdash \texttt{R} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \texttt{L}\_ \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \wedge \texttt{B}\_, \texttt{R2}\_\_\_\} \rightarrow_{\texttt{"}\wedge\texttt{R"}} \texttt{True}/; (\texttt{L} \vdash \{\texttt{R1}, \texttt{A}, \texttt{R2}\} \rightarrow_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{L} \vdash \{\texttt{R1}, \texttt{B}, \texttt{R2}\} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \vee \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \texttt{R}\_ \rightarrow_{\texttt{"}\vee\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{A}, \texttt{L2}\} \vdash \texttt{R} \rightarrow_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\{\texttt{L1}, \texttt{B}, \texttt{L2}\} \vdash \texttt{R} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \texttt{L}\_ \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \vee \texttt{B}\_, \texttt{R2}\_\_\_\} \rightarrow_{\texttt{"}\vee\texttt{R"}} \texttt{True}/; (\texttt{L} \vdash \{\texttt{R1}, \texttt{A}, \texttt{B}, \texttt{R2}\} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \Rightarrow \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \{\texttt{R}\_\_\_\} \rightarrow_{\texttt{"}\Rightarrow\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{L2}\} \vdash \{\texttt{A}, \texttt{R}\} \rightarrow_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\{\texttt{B}, \texttt{L1}, \texttt{L2}\} \vdash \{\texttt{R}\} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L}\_\_\_\} \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \Rightarrow \texttt{B}\_, \texttt{R2}\_\_\_\} \rightarrow_{\texttt{"}\Rightarrow\texttt{R"}} \texttt{True}/; (\{\texttt{A}, \texttt{L}\} \vdash \{\texttt{B}, \texttt{R1}, \texttt{R2}\} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \neg\texttt{A}\_, \texttt{L2}\_\_\_\} \vdash \{\texttt{R}\_\_\_\} \rightarrow_{\texttt{"}\neg\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{L2}\} \vdash \{\texttt{A}, \texttt{R}\} \rightarrow_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L}\_\_\_\} \vdash \{\texttt{R1}\_\_\_, \neg\texttt{A}\_, \texttt{R2}\_\_\_\} \rightarrow_{\texttt{"}\neg\texttt{R"}} \texttt{True}/; (\{\texttt{L}, \texttt{A}\} \vdash \{\texttt{R1}, \texttt{R2}\} \rightarrow_{\texttt{s}} \texttt{True})]
\end{aligned}
$$

where `s` is an identifier which should be instantiated with strategy *stg*. It could be the choice

$$\texttt{"}\wedge\texttt{L"} \mid \texttt{"}\wedge\texttt{R"} \mid \texttt{"}\vee\texttt{L"} \mid \texttt{"}\vee\texttt{R"} \mid \texttt{"}\Rightarrow\texttt{L"} \mid \texttt{"}\Rightarrow\texttt{R"} \mid \texttt{"}\neg\texttt{L"} \mid \texttt{"}\neg\texttt{R"} \mid \texttt{"I"}$$

but we can do better than that: We can use heuristics in the definition of `s` to reduce the search space for a proof. For example

$$\texttt{Fst}[\texttt{"I"}, \texttt{"}\wedge\texttt{L"} \mid \texttt{"}\wedge\texttt{R"} \mid \texttt{"}\vee\texttt{L"} \mid \texttt{"}\vee\texttt{R"} \mid \texttt{"}\Rightarrow\texttt{L"} \mid \texttt{"}\Rightarrow\texttt{R"} \mid \texttt{"}\neg\texttt{L"} \mid \texttt{"}\neg\texttt{R"}]$$

would be a better specification for `s` because it gives highest priority to rule `"I"` which, if applicable, detects immediately a proof for a sequent.

ρLog can generate a trace of its computation:

`Request[`Q`,Trace`$\rightarrow$`True]`

instructs ρLog to generate and open a Mathematica notebook with human-readable explanations of the rule-based computations that produced an answer or ended with failure. For example

```
s = Fst["I","∧L"|"∧R"|"∨L"|"∨R"|"⇒L"|"⇒R"|"¬L"|"¬R"];
Request[{} ⊢ {(P ⇒ Q) ⇒ ((¬Q) ⇒ ¬P)} →s True,Trace→True]
```

generates a notebook with explanations that certify that the sequent

$$\{\} \vdash \{(\texttt{P} \Rightarrow \texttt{Q}) \Rightarrow ((\neg\texttt{Q}) \Rightarrow \neg\texttt{P})$$

can be derived with the inference rules of system LK. A snapshot of this trace is shown in Fig. 3.

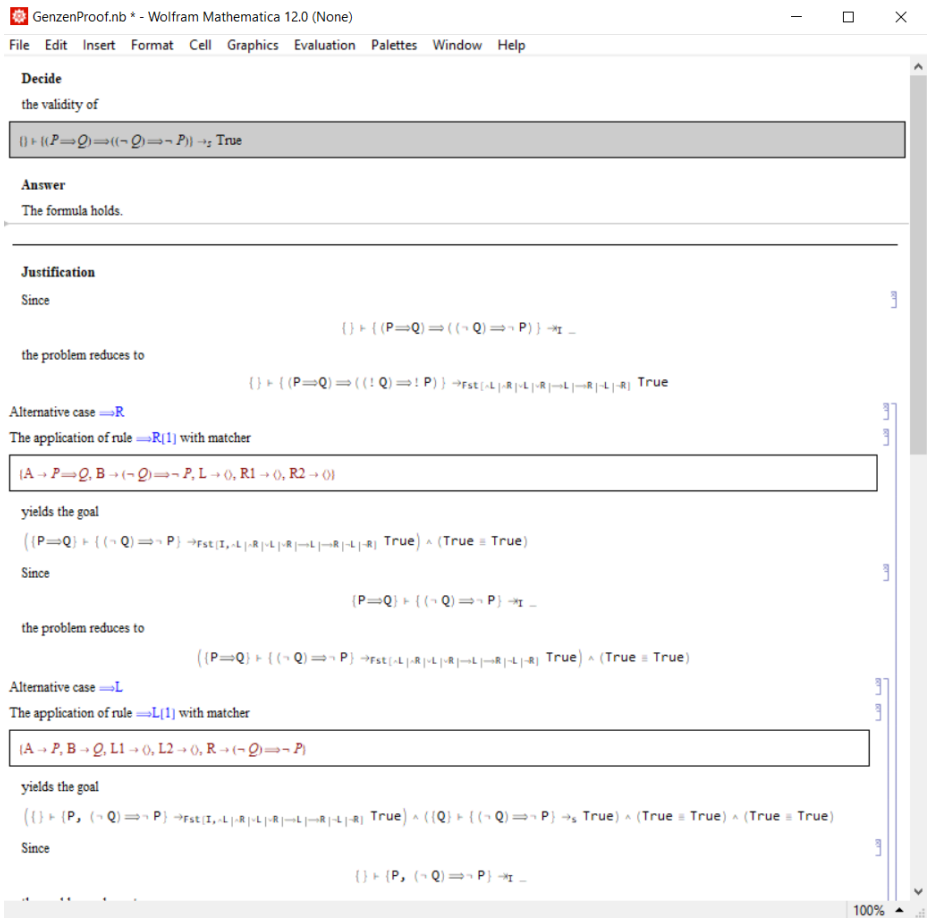

Fig. 3. A human-readable trace of a rule-based deduction.

3.3 Specification and safety-check of access-control models

A fundamental security problem in computing environments is to control the access of subjects to system abstractions (e.g., processes, files, or ports) or resources (e.g., printers). Attribute-based access control (ABAC) is the most recent authorization model, where access rights are granted to users through the use of policies which combine together the attributes of the participating entities (user/subject or subject/object), operations, and the environment relevant to a request. To better understand the power and limitations of this logical access control paradigm, several foundational models have been proposed [7, 6]: They provide minimal set of capabilities that suffice to configure many access control models of practical interest.

Of particular importance is ABAC$_\alpha$ [7], a foundational model for ABAC with a minimal set of capabilities to configure the dominant traditional access control models: discretionary (DAC), mandatory (MAC), and role-based (RBAC).

A system with an ABAC$_\alpha$ access control model can be viewed as a state transition system whose states are triples $\{U, S, O\}$ consisting of the existing users (U), subjects (S), and objects (O), and whose transitions correspond to the six operations from the functional specification of ABAC$_\alpha$: subject creation/deletion/modification, object creation/deletion, and authorized access.

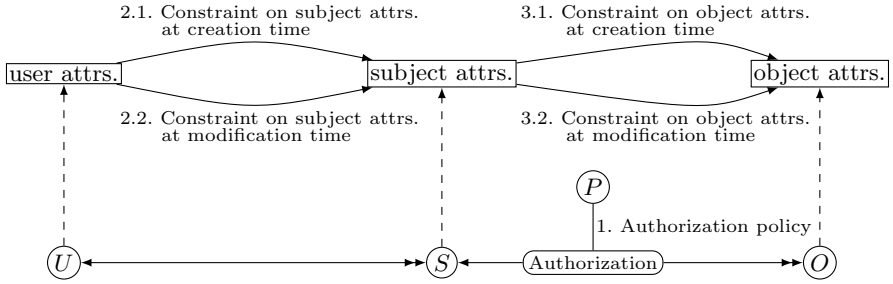

Fig. 4. The structure of ABAC$_\alpha$ model (adapted from [6])

In [13] we have shown that ρLog is a suitable framework to specify the operational model of ABAC$_\alpha$, because:

1. With parametric strategies, we can separate the declaration of configuration-specific policies from the declaration of policies characteristic to ABAC$_\alpha$. For example, the configuration point which grants to users the right to create a subject has a policy which depends on the specific configuration type (e.g., DAC, MAC, or RBAC).
2. The constraint logic programming component of ρLog is based on the constraint solving capabilities of Mathematica and can interpret correctly all boolean constraints expressed in the CPL instances of ABAC$_\alpha$.
3. The transitions of the operational model of ABAC$_\alpha$ can be defined as parametric strategies whose parameters get bound to configuration-specific identifiers, and can be used to enforce the application of the rule-based specifications of the configuration points.

In the remainder of this section, we describe a software tool developed by us in Mathematica, which relies on the rule-based programming capabilities of ρLog to specify any possible configuration of ABAC$_\alpha$ and its operational model, and to check safety properties. The tool can be downloaded from the website of our system:

`http://staff.fmi.uvt.ro/~mircea.marin/rholog/`

Representation of ABAC$_\alpha$ entitites. The entities of ABAC$_\alpha$ are users, subjects, or objects. Each user, subject, object is associated with a finite set of user attributes (UA), subject attributes (SA) and object attributes (OA) respectively. Every attribute *att* has a type, scope, and finite range of possible values. The sets of attributes specific to each kind of entity, together with their corresponding type, scope, and range, are specified in a *configuration type*.

We represent entities as flat terms of the form

$$E[at_1[val_1], \ldots, at_n[val_n]]$$

where the term constructor $E \in \{\bullet\mathtt{U}, \bullet\mathtt{S}, \bullet\mathtt{O}\}$ indicates the entity type (user, subject, or object), every at_i is a string literal that indicates an attribute name, and val_i is a Mathematica literal that indicates the value of the corresponding attribute.

Configuration types and configurations. In ABAC$_\alpha$, every attribute *at* has (1) a scope SCOPE[*at*] which is a finite set of values, and (2) a type which is either atomic or set. The range of values of *at* is SCOPE[*at*] if the type of *at* is atomic, and $2^{\text{SCOPE}[at]}$ if the type of *at* is set. A *configuration type* specifies the admissible attributes for every kind of entity, their scope and their type.

Our tool provides the command `DeclareCfgType` to declare ABAC$_\alpha$ configuration types. The syntax of this command is

`DeclareCfgType[`*`typeId`*`,`
$\{\mathtt{UA}\rightarrow\{uAt_1,\ldots,uAt_m\},\mathtt{SA}\rightarrow\{sAt_1,\ldots,sAt_n\},\mathtt{OA}\rightarrow\{oAt_1,\ldots,oAt_p\},$
$\mathtt{Scope}\rightarrow\{at_1\rightarrow\{sId_1,\tau_1\},\ldots,at_r\rightarrow\{sId_r,\tau_r\}\}\}]$

where *`typeId`* is the newly declared ID of a configuration type; $\{sAt_1,\ldots,sAt_n\}$ is the set of attributes for users; $\{uAt_1,\ldots,uAt_m\}$ is the set of attributes for subjects; $\{oAt_1,\ldots,oAt_p\}$ is the set of attributes for objects; and

> the scope of every attribute at_i is the set bound to identifier sId_i in a particular configuration (see below), and its type is $\tau_i \in \{\texttt{"elem"},\texttt{"subset"}\}$, where "`elem`" stands for atomic and `"subset"` for set.

A *configuration* is an instance of a configuration type which indicates (1) the sets of values for the identifiers sId_i from the specification of a configuration type, and (3) the sets U, S, and O of entities (users, subjects, objects) in the initial configuration of the system.

Our tool provides the command `DeclareConfiguration` to declare a configuration *`cId`* for a configuration type *`typeId`*. It has the syntax

`DeclareConfiguration[`*`cId`*`, {CfgType`$\rightarrow$*`typeId`*`,`
$\mathtt{Users}\rightarrow\{uId_1\rightarrow u_1,\ldots,uId_m\rightarrow u_m\},$
$\mathtt{Range}\rightarrow\{\mathtt{UId}\rightarrow\{uId_1,\ldots,uId_m\},$
$sId_2\rightarrow\text{SCOPE}(at_2),\ldots,sId_r\rightarrow\text{SCOPE}(at_r)\},$
$\mathtt{Subjects}\rightarrow\{s_1,\ldots,s_n\},$
$\mathtt{Objects}\rightarrow\{o_1,\ldots,o_q\}\}]$

The side effect of this command is to instantiate some globally visible entries:

`CfgType[`*`cId`*`]` with *`typeId`*,
`Users[`*`cId`*`]` with the set $\{u_1, \ldots, u_m\}$ of terms for users,
every `User[`*`cId`*`,uId`$_i$`]` with the term u_i,
`Subjects[`*`cId`*`]` with the set $\{s_1, \ldots, s_n\}$ of terms for subjects, and
`Objects[`*`cId`*`]` with the set $\{o_1, \ldots, o_q\}$ of terms for objects.

To illustrate, consider the mandatory access control model (MAC). Users and subjects have a clearance attribute of type elem, whose value is a number from a finite set of integers $L = \{1, 2, \ldots, N\}$ which indicates the security level of the corresponding entity. Objects have a sensitivity attribute of type `"elem"` whose value is also from L, and represents the sensitivity degree of the information in that object. When read and write are the only permissions on objects, we can assume the set of permissions P to be {`"read","write"`}. A configuration type for MAC can be defined as follows:

```
DeclareCfgType["MAC",
 {UA→{"id","clearance"},
  SA→{"id","clearance"},
  OA→{"sensitivity"},
  Scope→{"id"→{"uId","elem"},"clearance"→{"level","elem"},
                          "sensitivity"→{"level","elem"}}}]
```

and a particular MAC configuration can be defined by

```
DeclareConfiguration["MAC-Cfg01",
 {CfgType→"MAC",
  Users→{"u1"→•U["id"["u"],"clearance"[3]],
         "u2"→•U["id"["u2"],"clearance"[4]]},
  Range→{uId→{"u1","u2"},"level"→{1,2,3,4,5}},
  Subjects→{•S["id"["u1"],"clearance"[3]],
            •S["id"["u2"],"clearance"[2]]},
  Objects→{•O["sensitivity"[1],•O["sensitivity"[4]]}}]
```

Specification of the policy configuration points. The operational model depicted in Fig. 4 has five policy configuration points: for subject creation, object creation, modification of the attribute values of a subject, modification of attribute value of an object, and granting a permission to a subject on an object. With our tool, all configuration points of a particular configuration type can be specified by declaring instances of parametric strategies for every kind of operation.

For a configuration type *`cfgTypeId`*, the configuration point for subject creation is specified with a declaration

`DeclareRules[ConstrSub[`$uPatt, sPatt$`]`$\rightarrow_{cfgTypeId}$ `True/;`$\langle$*CPL-formula*$\rangle$`]`

where $\langle$*CPL-formula*$\rangle$ is a logical formula written in the CPL-language of ABAC$_\alpha$. For example, subject creation in MAC can be specified by

```
DeclareRules[
  ConstrS[{•U[x_,"clearance"[y_]],•S[x_,"clearance"[z_]]}
       →"MAC" True/;(z≤y)]
```

The other four configuration points are specified in a similar way, with declarations of the form

```
DeclareRules[
   ConstrModS[uPatt, sPatt1, sPatt2] →cfgTypeId True/;⟨CPL-formula2⟩,
   ConstrO[sPatt, oPatt] →cfgTypeId True/;⟨CPL-formula3⟩,
   ConstrModO[sPatt, oPatt1, oPatt2] →cfgTypeId True/;⟨CPL-formula4⟩,
   Auth[p1, sPatt, oPatt] →cfgTypeId True/;⟨CPL-formula5,1⟩,
   ...
   Auth[pn, sPatt, oPatt] →cfgTypeId True/;⟨CPL-formula5,n⟩
]
```

For example, the rule-based specifications of MAC are

```
DeclareRules[
  ConstrModS[_,_,_] →"MAC" False,
  ConstrO[{•S[_,"clearance"[x_]],•O[x_,"sensitivity"[y_]]
       →"MAC" True/;(x≤y),
  ConstrModO[_,_,_] →"MAC" False,
  Auth["read",•S[_,"clearance"[x_],•O["sensitivity"[y_]]
       →"MAC" True/;(y≤x),
  Auth["write",•S[_,"clearance"[x_],•O["sensitivity"[y_]]
       →"MAC" True/;(x≤y)
]
```

Specification of the operational model. The state transitions in the operational model of ABAC$_\alpha$ of a particular configuration `cId` of ABAC$_\alpha$ can be specified as instances of parametric strategies:

1. `"createSubj"[cId]`: it acts on a state $\{U, S, O\}$ by extending, if possible, S with a newly created subject,
2. `"deleteSubj"[cId]`: it acts on a state $\{U, S, O\}$ by deleting, if possible, a subject from S,
3. `"createObj"[cId]`: it acts on a state $\{U, S, O\}$ by extending, if possible, O with a newly created object,
4. `"modifySubj"[cId]`: it acts on a state $\{U, S, O\}$ by modifying, if possible, the attributes of a subject from S,
5. `"modifyObj"[cId]`: it acts on a state $\{U, S, O\}$ by modifying, if possible, the attributes of an object from O,
6. `"auth?"[p,cfgTypeId]`: it acts on a pair $\{S, O\}$ and yields True if there is a subject $s \in S$ with permission `p` on an object $o \in O$.

The definition of these strategies in terms of the parametric strategies of the policy configuration points is quite straightforward, and can be found in [13].

Safety analysis of ABAC$_\alpha$ configurations. A fundamental problem for any access control model is safety analysis. For ABAC$_\alpha$, the safety problem can be stated as follows:

Given an initial state $St_0 = \{U, S_0, O_0\}$ for a particular configuration of ABAC$_\alpha$, a subject $s \in S_0$, an object $o \in O_0$, and a permission p

Decide if there is a scenario, that is, a sequence of ABAC$_\alpha$ state transitions from St_0 to a state $St_n = \{U, S_n, O_n\}$ where s and o have attribute values that authorize subject s to exercise permission p on object o.

This problem was recently shown to be decidable [1], but no practical algorithm was proposed. In [13], we described a rule-based implementation in ρLog of such an algorithm. Our algorithm decides the safety problem in two steps:

1. First, we compute all possible combinations of attribute values that s may get, if the initial state is St_0. There are finitely many such combinations, which we call descendants of s, and collect them incrementally in a set $sDesc$.
   - We interleave the computation of $sDesc$ with the application of the labeled conditional rewrite rule
     $\texttt{Auth}[p, sPatt, oPatt] \rightarrow_{cfgTypeId}$ `True/;`$\langle$*CPL-formula*$\rangle$
     to early detect if any descendant of s can exercise permission p on o. If yes, the configuration of ABAC$_\alpha$ is unsafe.
2. Next, we compute all possible combinations of attribute values that o may get by the operations exercised by all subjects that can be created and can modify their attribute values. There are finitely many such combinations, which we call *descendants* of o, and collect them incrementally in a set $oDesc$.
   - We interleave the computation of $oDesc$ with the application of the labeled conditional rewrite rule
     $\texttt{Auth}[p, sPatt, oPatt] \rightarrow_{cfgTypeId}$ `True/;`$\langle$*CPL-formula*$\rangle$
     to early detect if any descendant of s can exercise permission p on and descendant of o. If yes, the configuration of ABAC$_\alpha$ is unsafe. If no such situation is detected, the configuration of ABAC$_\alpha$ is reported to be safe.

To run this algorithm, we provide the method `CheckSafety[`*`cfgId`*`,`s, o, p`]` where *`cfgId`* is the identifier of the initial configuration under consideration. The method returns `"UNSAFE"` as soon as it detects that a descendant of s can get permission p on a descendant of object o, and `"SAFE"` otherwise.

4 Conclusion

ρLog is a system for rule-based programming with strategies and labeled conditional rewrite rules. It is based on a rewriting calculus designed by us, which has the following characteristics:

1. It accepts specifications with term variables, function variables, sequence variables, and context variables. As a result, the specifications are more natural and concise. For instance, sequence variables and context variables

permit matching to descend to arbitrary depth and width in a term represented as a tree. The ability to explore terms in two orthogonal directions in a uniform turned out be useful for querying data available as a large term, like XML documents [10].
2. It is based on sound and complete matching algorithms developed by us.
3. Its range of applications is enlarged significantly by its access to the constraint solving capabilities of Mathematica, its host language.

The ρLog calculus was also used to implement PρLog [4], an experimental tool that extends logic programming with strategic conditional transformation rules. PρLog combines Prolog with the ρLog calculus, and is available for free download from

`https://www3.risc.jku.at/people/tkutsia/software/prholog`

We already mentioned that ρLog can trace the computation of a derivation $Q \rightsquigarrow^*_\sigma \top$ as a human-readable proof that substitution σ satisfies query Q in the rewrite theory represented by a program P [14, 12]. Failed proof attempts $Q \not\rightsquigarrow^* \top$ can be traced too. This capability was inspired from Theorema [3, 5], a theorem prover which influenced the development of ρLog and with which our system shares many features.

We are currently investigating the possibility to extend our calculus with capabilities for approximate reasoning, by solving constraints over several similarity relations [2]. Similarity relations are reflexive, symmetric, and transitive fuzzy relations. They help to make approximate inferences, but pose challenges to constraint solving, since we can not rely on the transitivity property anymore.

References

1. Ahmed, T., Sandhu, R.: Safety of ABAC$_\alpha$ Is Decidable. In: Yan, Z., Molva, R., Mazurczyk, W., Kantola, R. (eds.) Network and System Security. pp. 257–272. Springer International Publishing, Cham (2017)
2. Ait-Kaci, H., Pasi, G.: Fuzzy unification and generalization of first-order terms over similar signatures. In: Fioravanti, F., Gallagher, J.P. (eds.) Logic-Based Program Synthesis and Transformation – 27th International Symposium, LOPSTR 2017. LNCS, vol. 10855, pp. 218–234. Springer, Namur, Belgium (2017)
3. Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuţa, E., Văsaru, D.: A Survey of the Theorema Project. In: Proceedings of ISSAC'97. pp. 384–391. Maui, Hawaii, USA (1997)
4. Dundua, B., Kutsia, T., Reisenberger-Hagmayer, K.: An Overview of PρLog. In: Y. Lierler and W. Taha (ed.) Proceedings of PADL 2017. LNCS, vol. 10137, pp. 34–49. Springer, Paris, France (2017)
5. Jebelean, T., Drămnesc, I.: Synthesis of list algorithms by mechanical proving. JSC **69**, 61–92 (2015)
6. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) Data and Applications Security and Privacy XXVI. LNCS, vol. 7371, pp. 41–55. Springer, Berlin, Heidelberg (2012)

7. Jin, X.: Attribute-Based Access Control Models and Implementation in Cloud Infrastructure as a Service. Ph.D. thesis, University of Texas at San Antonio (2014)
8. Kutsia, T.: Solving equations with sequence variables and sequence functions. JSC **42**(3), 352–388 (2007)
9. Kutsia, T.: Solving equations involving sequence variables and sequence functions. In: Buchberger, B., Campbell, J.A. (eds.) Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings. LNCS, vol. 3249, pp. 157–170. Springer (2004). https://doi.org/10.1007/978-3-540-30210-0_14, https://doi.org/10.1007/978-3-540-30210-0_14
10. Kutsia, T., Marin, M.: Can Context Sequence Matching be Used for Querying XML? In: Vigneron, L. (ed.) Proceedings of the 19th International Workshop on Unification (UNIF'05). pp. 77–92. Nara, Japan (2005)
11. Kutsia, T., Marin, M.: Matching with regular constraints. In: Sutcliffe, G., Voronkov, A. (eds.) Proceedings of LPAR 2005. LNAI, vol. 3835, pp. 215–229. Montego Bay, Jamaica (2005)
12. Marin, M., Kutsia, T.: Foundations of the rule-based system ρLog. Journal of Applied Non-Classical Logics **16**(1-2), 151–168 (2006)
13. Marin, M., Kutsia, T., Dundua, B.: A Rule-Based Approach to the Decidability of Safety of $ABAC_\alpha$. In: Proceedings of the 24th ACM Symposium on Access Control Models and Technologies. pp. 173–178. SACMAT 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3322431.3325416
14. Marin, M., Piroi, F.: Deduction and Presentation in ρLog. In: Proceedings of MKM. ENTCS, vol. 93, pp. 161–182 (2004)
15. Marin, M., Piroi, F.: Rule-Based Programming with Mathematica. In: Proceedings of International Mathematica Symposium (IMS 2004). Banff, Canada (2004)
16. Wolfram, S.: The Mathematica Book. Wolfram Media, 5 edn. (2003)