On the Implementation of a Rule-Based Programming System and Some of its Applications*

Mircea Marin[1] and Temur Kutsia[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-4232 Hagenberg, Austria
tkutsia@risc.uni-linz.ac.at

Abstract. We describe a rule-based programming system where rules specify nondeterministic computations. The system is called FunLog and has constructs for defining elementary rules, and to build up complex rules from simpler ones via operations akin to the standard operations from abstract rewriting. The system has been implemented in Mathematica and is, in particular, useful to program procedures which can be encoded as sequences of rule applications which follow a certain reduction strategy. In particular, the procedures for unification with sequence variables in free, flat, and restricted flat theories can be specified via a set of inference rules which should be applied in accordance with a certain strategy. We illustrate how these unification procedures can be expressed in our framework.

1 Introduction

In this paper we describe a rule-based programming language and illustrate its usefulness for implementing unification procedures with sequence variables in free, flat and restricted flat theories. We have designed and implemented a rule-based system called FunLog, which provides programming constructs to define elementary rules and to build up complex rules from simpler ones. Our programming constructs include primitives to compose rules, to group them into a specification of a nondeterministic rule, to compute with their transitive closure, and to define various evaluation strategies.

Such a programming style is very suitable for complex computations steps which can be expressed as sequences of computation steps with the following characteristics:

* Mircea Marin has been supported by the Austrian Academy of Sciences. Temur Kutsia has been supported by the Austrian Science Foundation (FWF) under Project SFB F1302.

1. Each computation step is driven by the application of a rule chosen (nondeterministically) from a finite set of alternative rules.
2. The sequence of steps must match a certain specification. In FunLog, such specifications can be built up via a number of operators which are similar to the operators of abstract term rewriting.

We used Funlog to implement unification procedures in free, flat, and restricted flat theories with sequence variables and flexible arity symbols. It was shown in [8, 9] that these procedures can be specified via a set of inference rules which should be applied in accordance with a certain strategy. Therefore, these procedures are a good example to be programmed in FunLog.

The rest of this paper is structured as follows. In Section 2 we give a brief description of the rule-based programming style supported by FunLog and describe the programming constructs of our system. Section 3 explains some details about the Mathematica implementation of FunLog. In Section 4 we describe our FunLog implementation of unification procedures with sequence variables. Section 5 concludes.

2 Programming with FunLog

FunLog is a rule-based system where

rule = specification of a partially defined, possibly nondeterministic computation.

This paradigm makes our notion of rule very similar to the notion of strategy as defined in the rule-based system Elan [4, 7]. There are, however, some notable exceptions:

1. all rules can be nondeterministic. The nondeterministic application of basic rules stems from the fact that we allow patterns which can match in more than one way with a given expression.
2. the application of rules is not driven by a built-in leftmost-innermost rewriting strategy. Instead, rules are always applied at the root position of a term. Rewriting behaviors can be attained by associating a new rule to a given rule, and imposing a certain strategy to look up for the subterm to be rewritten.

We believe that these features make our rule-based system more flexible, mainly because we can control the positions where rules should be applied.

In FunLog, each rule is characterized by a name and a code which describes a partially defined, possibly nondeterministic, computation. Formally, a rule is an expression

$$lbl :: patt \rightarrow rhs \tag{1}$$

where *lbl* is the rule name, *patt* is a pattern expression and *rhs* specifies the computation of a result from the bindings of the variables which occur in *patt*. The expression $patt :> rhs$ is called the *code* of the rule *lbl*.

The main usage of rules is to act with them them on various expressions. The attempt to apply a rule of type (1) on an expression *expr* proceeds as follows:

1. $\mathtt{M} :=$ enumeration of all matchers between *expr* and *patt*.
2. If $\mathtt{M} = \emptyset$ then **fail** else goto 3.
3. $\theta := \mathtt{first}(\mathtt{M})$, $\mathtt{M} := \mathtt{rest}(M)$, $\mathtt{V} :=$ enumeration of all values of $\theta(rhs)$.
4. If $\mathtt{V} = \emptyset$ then goto 3 else **return** $\mathtt{first}(\mathtt{V})$.

We write $expr \not\to_{lbl}$ if the application of rule *lbl* to *expr* fails, and $expr \to_{lbl}$ if it succeeds.

There are two sources of non-determinism in FunLog: non-unique matches and non-unique ways to evaluate a partially defined computation. Clearly, the result of an application $expr \to_{lbl}$ depends on the enumerations of matchers (M) and values (V) which are built into a particular implementation. These enumeration strategies are relevant to the programmer and are described in the specification of the operational semantics of our implementation.

In the sequel we write $expr_1 \to_{lbl} expr_2$ if we can identify two enumerations, for M and V, which render the result $expr_2$ for the application $expr_1 \to_{lbl}$.

Rules can be combined with various combinators into more complex rules. The implementation of these combinators is compositional, i.e., the meaning of each combination of rules can be defined in terms of the meanings of the component rules.

2.1 Main Combinators

A rule with name *lbl* is applied to an expression $expr_1$ via the call

$$\mathtt{ApplyRule}[expr_1, lbl] \tag{2}$$

which behaves as follows:

- If $expr_1 \not\to_{lbl}$ then **return** $expr_1$
- If $expr_1 \to_{lbl}$ then **return** the first $expr_2$ for which $expr_1 \to_{lbl} expr_2$.

The call

$$\mathtt{ApplyRuleList}[expr_1, lbl] \tag{3}$$

returns the list of values $\{expr_2 \mid expr_1 \to_{lbl} expr_2\}$.

FunLog provides a number of useful constructs to build up rules. These constructs are described in the remainder of this section.

Basic rules. A basic rule is a rule named by a string, whose code is given explicitly as a Mathematica transformation rule. A basic rule $lbl :: patt :> rhs$ is declared by

$$\mathtt{DeclareRule}[patt :> rhs, lbl] \tag{4}$$

We recall that $expr_1 \to_{lbl} expr_2$ iff there is a matcher θ between $expr_1$ and *patt* for which $\theta(rhs)$ evaluates to $expr_2$.

The enumeration strategy of basic rules depends only on the enumeration strategy of matches.

Example 1. The rule "split" introduced by the declaration

$$\texttt{DeclareRule}[\{x\_\_\_, y\_\_\_\}/;(\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}]) :> \{x\}, \text{“split”}]$$

takes as input a list L of elements and yields a prefix sublist of L whose length is larger than half the length of L. The outcome of the call

$$\texttt{ApplyRule}[\{a, b, c, d\}, \text{“split”}]$$
$$\{a, b, c\}$$

yields the instance $\{a, b, c\} = \theta(\{x\})$ corresponding to the matcher $\theta = \{x \mapsto \ulcorner a, b, c\urcorner, y \mapsto \ulcorner d\urcorner\}$. Note that this is the first matcher found by the enumeration strategy of the MATHEMATICA interpreter, for which $\theta(\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}])$ holds. □

Choice. $lbl_1 \mid \ldots \mid lbl_n$ denotes a rule whose applicative behavior is given by

$$expr_1 \rightarrow_{lbl_1|\ldots|lbl_n} expr_2 \text{ iff } expr_1 \rightarrow_{lbl_i} expr_2 \text{ for some } i \in \{1, \ldots, n\}. \tag{5}$$

The enumeration of the steps $expr_1 \rightarrow_{lbl_1|\ldots|lbl_n}$ starts with the enumeration of the steps $expr_1 \rightarrow_{lbl_1}$, followed by the enumeration of the steps $expr_1 \rightarrow_{lbl_2}$, and so on up to the enumeration of the steps $expr_1 \rightarrow_{lbl_n}$.

Example 2. Consider the declarations

$$\texttt{DeclareRule}[\{x\_\_\_m\_, y\_\_\_, n\_, z\_\_\_\} :> \texttt{False}/;(m > n), \text{“test”}];$$
$$\texttt{DeclareRule}[\_\texttt{List} :> \texttt{True}, \text{“else”}];$$

Here, `_List` is a pattern variable which matches any list structure. Then

$$\texttt{ApplyRule}[L, \text{“test”} \mid \text{“else”}]$$

yields `True` iff L is a list with elements in ascending order. This behavior is witnessed by the calls:

$$\texttt{ApplyRule}[\{1, 2, 4, 5, 3\}, \text{“test”} \mid \text{“else”}]$$
$$\texttt{False}$$
$$\texttt{ApplyRule}[\{1, 2, 3, 4, 5\}, \text{“test”} \mid \text{“else”}]$$
$$\texttt{True}$$

The first call yields `False` because $\{1, 2, 4, 5, 3\} \rightarrow_{\text{“test”}} \texttt{False}$ with the matcher $\theta = \{x \mapsto \ulcorner 1, 2\urcorner, m \mapsto 5, y \mapsto \ulcorner\urcorner, n \mapsto 4, z \mapsto \ulcorner 3\urcorner\}$. The second call yields `True` because $\{1, 2, 3, 4, 5\} \not\rightarrow_{\text{“test”}}$ and $\{1, 2, 3, 4, 5\} \rightarrow_{\text{“else”}} \texttt{True}$. □

Composition. $lbl_1 \circ lbl_2$ denotes a rule whose applicative behavior is given by

$$expr_1 \rightarrow_{lbl_1 \circ lbl_2} expr_2 \text{ iff } expr_1 \rightarrow_{lbl_1} expr \rightarrow_{lbl_2} expr_2 \text{ for some } expr. \tag{6}$$

The enumeration of values $expr_2$ for which the relation $expr_1 \rightarrow_{lbl_1 \circ lbl_2} expr_2$ holds, proceeds by enumerating all steps $expr \rightarrow_{lbl_2} expr_2$ during an enumeration of the steps $expr_1 \rightarrow_{lbl_1} expr$.

Example 3 (Oriented graphs). The following rule declarations

$$\begin{array}{l}\texttt{DeclareRule}[x\_ :> x, \text{“Id”}];\\ \texttt{DeclareRule}[a :> b, \text{“r1”}]; \texttt{DeclareRule}[a :> c, \text{“r2”}];\\ \texttt{DeclareRule}[c :> b, \text{“r3”}]; \texttt{DeclareRule}[b :> d, \text{“r4”}];\\ \texttt{DeclareRule}[b :> e, \text{“r5”}]; \texttt{DeclareRule}[c :> f, \text{“r6”}];\end{array}$$

define the edges of an oriented graph with nodes $\{a, b, c, d, e, f\}$. Then

$$a \rightarrow_{\texttt{Repeat}[\text{“r1”}|\text{“r2”}|\text{“r3”}|\text{“r4”}|\text{“r5”}|\text{“r6”},\text{“Id”}]} v$$

iff there exists a (possibly empty) path from a to v. To find such a v we can call

$$\texttt{ApplyRule}[a, \texttt{Repeat}[\text{“r1”} \mid \text{“r2”} \mid \text{“r3”} \mid \text{“r4”} \mid \text{“r5”} \mid \text{“r6”}, \text{“Id”}]]$$

and FunLog will yield the value d corresponding to the derivation

$$a \rightarrow_{\text{“r1”}} b \rightarrow_{\text{“r4”}} d \rightarrow_{\text{“Id”}} d.$$

The call

$$\texttt{ApplyRuleList}[c, \texttt{Repeat}[\text{“r1”} \mid \text{“r2”} \mid \text{“r3”} \mid \text{“r4”} \mid \text{“r5”} \mid \text{“r6”}, \text{“Id”}]]$$

will yield the list $\{d, e, b\}$ of all nodes reachable from b. □

Reflexive-transitive closures. If $lbl \in \{\texttt{Repeat}[lbl_1, lbl_2], \texttt{Until}[lbl_2, lbl_1]\}$ then

$$expr_1 \rightarrow_{lbl} expr_2 \text{ iff } expr_1 \rightarrow^*_{lbl_1} expr \rightarrow_{lbl_2} expr_2 \text{ for some } expr \tag{7}$$

where $\rightarrow^*_{lbl_1}$ denotes the reflexive-transitive closure of $\rightarrow_{lbl_1}$. These two constructs differ only with respect to the enumeration strategy of the possible reduction steps.

The enumeration of $expr_1 \rightarrow_{\texttt{Repeat}[lbl_1,lbl_2]} expr_2$ proceeds by unfolding the recursive definition:

$$\texttt{Repeat}[lbl_1, lbl_2] = lbl_1 \circ \texttt{Repeat}[lbl_1, lbl_2] \mid lbl_2,$$

whereas the enumeration of $expr_1 \rightarrow_{\texttt{Until}[lbl_1,lbl_2]} expr_2$ proceeds by unfolding the recursive definition:

$$\texttt{Until}[lbl_2, lbl_1] = lbl_2 \mid lbl_1 \circ \texttt{Until}[lbl_2, lbl_1].$$

Example 4 (Sorting). Consider the declarations of basic rules

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

Then the enumeration strategy of $\texttt{Repeat}[\text{“perm”}, \text{“Id”}]$ ensures that the application of rule $\texttt{Repeat}[\text{“perm”}, \text{“Id”}]$ to any list of integers yields the sorted version if that list. For example

$$\texttt{ApplyRule}[\{3,1,2\}, \texttt{Repeat}[\text{“perm”}, \text{“Id”}]]$$

yields $\{1,2,3\}$ via the following sequence of transformation steps

$$\{3,1,2\} \to_{\text{“perm”}} \{1,3,2\} \to_{\text{“perm”}} \{1,2,3\} \to_{\text{“Id”}} \{1,2,3\}.$$

□

Rewrite rules. FunLog provides the following mechanism to define a rule that rewrites with respect to a given rule:

$$\texttt{RWRule}[lbl_1, lbl, \texttt{Traversal} \to \ldots, \texttt{Prohibit} \to \ldots] \tag{8}$$

This call declares a new rule named *lbl* such that $expr_1 \to_{lbl} expr_2$ iff there exists a position p in $expr_1$ such that $expr_1|_p \to_{lbl_1} expr$ and $expr_2 = expr_1[expr]_p$. Here, $expr_1|_p$ is the subexpression of *expr* at position p, and $expr_1[expr]_p$ is the result of replacing the subexpression at position p by *expr* in $expr_1$. The option `Traversal` defines the enumeration ordering of rewrite steps (see below), whereas the option `Prohibit` restricts the set of positions allowed for rewriting. If the option $\texttt{Prohibit} \to \{f_1, \ldots, f_n\}$ is given, then rewriting is prohibited at positions below occurrences of symbols $f \in \{f_1, \ldots, f_n\}$. By default, the value of `Prohibit` is $\{\}$, i.e., rewriting can be performed everywhere.

The enumeration strategy of rewrite steps can be controlled via the option `Traversal` which has the default value ”LeftmostIn”. If the option $\texttt{Traversal} \to$ “LeftmostOut” is given, then the rewriting steps are enumerated by traversing the rewriting positions in leftmost-outermost order. If $\texttt{Traversal} \to$ “LeftmostIn” is given, then the rewriting steps are enumerated by traversing the rewriting positions in leftmost-innermost order.

Example 5 (Pure λ-calculus). In λ-calculus, a value is an expression which has no β-redexes outside λ-abstractions. We adopt the following syntax for λ-terms:

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

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

$$\texttt{DeclareRule}[\texttt{app}[\lambda[x\_, t1\_], t2\_] :> \texttt{repl}[t_1, \{x, t2\}], \text{“}\beta\text{”}]$$

where $\texttt{repl}[t_1, \{x, t2\}]$ replaces all free occurrences of x in $t1$ by $t2$. A straightforward implementation of repl in MATHEMATICA is shown below[1]:

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

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

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

The following calls illustrate the behavior of this rule:

$$
\begin{array}{l}
t := \texttt{app}[z, \texttt{app}[\lambda[x, \texttt{app}[x, \lambda[y, \texttt{app}[x, y]]]], \lambda[z, z]]];\\
t1 := \texttt{ApplyRule}[t, \text{“}\beta\text{-elim”}]\\
\;\texttt{app}[z, \texttt{app}[\lambda[z, z], \lambda[y, \texttt{app}[\lambda[z, z], y]]]]\\
t2 := \texttt{ApplyRule}[t1, \text{“}\beta\text{-elim”}]\\
\;\texttt{app}[z, \lambda[y, \texttt{app}[\lambda[z, z], y]]]\\
t3 := \texttt{ApplyRule}[t2, \text{“}\beta\text{-elim”}]\\
\;\texttt{app}[z, \lambda[y, \texttt{app}[\lambda[z, z], y]]]
\end{array}
$$

Thus, $t2$ is a value of t. To compute the value of t directly, we could call

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

□

Normal forms. $\texttt{NF}[lbl]$ denotes a rule whose applicative behavior is given by

$$expr \to_{\texttt{NF}[lbl]} expr_2 \text{ iff } expr_1 \to^*_{lbl} expr_2 \text{ and } expr_2 \not\to_{lbl}. \tag{9}$$

The enumeration strategy of $\texttt{NF}[lbl]$ is obtained by unfolding the recursive definition

$$\texttt{NF}[lbl] = \texttt{NFQ}[lbl] \mid lbl \circ \texttt{NF}[lbl] \tag{10}$$

where $\texttt{NFQ}[lbl] :: x\_ :> x/;(x \not\to_{lbl})$.

Abstraction. The abstraction principle is provided in FUNLOG via the construct

$$\texttt{SetAlias}[expr, lbl] \tag{11}$$

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

where *lbl* is a fresh rule name (a string identifier) and *expr* is an expression built from names of rules already declared with the operators $|$, $\circ$, `Repeat` and `Until` described before. For example, the call

$$\texttt{SetAlias}[\texttt{Repeat}[\text{“perm”}, \text{“Id”}], \text{“sort”}]$$

declares a rule named “sort” whose applicative behavior coincides with that of the construct $\texttt{Repeat}[\text{“perm”}, \text{“Id”}]$.

3 Notes on Implementation

We have implemented a MATHEMATICA package called FUNLOG which supports the programming style elaborated above. We decided to implement FUNLOG in MATHEMATICA because MATHEMATICA has very advanced pattern matching constructs for specifying transformation rules. Moreover, MATHEMATICA provides a powerful mechanism to control backtracking. More precisely, it allows to specify transformation rules of the form

$$\mathit{patt} :> \texttt{Block}[\{\texttt{result}, \ldots\}, \texttt{result}/; \mathit{test\_with\_side\_effect}]$$

which, when applied to an expression *expr* behave as follows:

1. If *patt* matches with *expr*, compute θ :=the first matcher and go to 2. Otherwise fail.
2. Evaluate the condition *test_with_side_effect*, in which we instantiate the pattern variables with the bindings of θ.
3. If the computation yields `True`, it also computes `result` as a side effect. This is possible because the `Block` construct makes the variable `result` visible inside the calls of *test_with_side_effect*.
4. If *test_with_side_effect* yields `False` then the interpreter of MATHEMATICA backtracks by computing $\theta :=$ next matcher and goto 2. If no matchers are left, then fail.

We have employed this construct to implement the backtracking mechanism of FUNLOG. For instance, the code for $lbl_1 \circ lbl_2$ is computed as follows:

– assume $\mathit{patt} :> \mathit{rhs}$ is the code of lbl_1. Then the code of $lbl_1 \circ lbl_2$ will be of the form

  $$\mathit{patt} :> \texttt{Block}[\{\texttt{result}, \ldots\}, \texttt{result}/; \texttt{CasesQ}[\mathit{rhs}, lbl_2]]$$

  where $\texttt{CasesQ}[\mathit{rhs}, lbl_2]$ does the following:
  - If $\mathit{rhs} \rightarrow_{lbl_2} \mathit{expr}_1$, then it binds `result` to expr_1 and yields `True`.
  - If $\mathit{rhs} \not\rightarrow_{lbl_2}$ it yields `False`.

  This decision step relies on the possibility to detect whether the code of rule lbl_2 accepts expr_1 as input. In our implementation, the code of lbl_2 is a MATHEMATICA transformation rule of the form $\mathit{patt}_2 :> \mathit{rhs}_2$. The piece of MATHEMATICA code

  $$\mathit{ok} = \texttt{True};\ \texttt{result} = \texttt{Replace}[\mathit{expr}_1, \{\mathit{patt}_2 \rightarrow \mathit{rhs}_2, \_ \rightarrow (\mathit{ok} = \texttt{False})\}]$$

  does the following:

- It sets the value of ok to `True`.
- It applies the rule $patt_2 \rightarrow rhs_2$ to $expr_1$ by assigning $expr_2$ to `result` if $expr_1 \rightarrow_{lbl_2} expr_2$.
- If $expr_1 \not\rightarrow_{lbl_2}$ then it applies the MATHEMATICA rule $\_ \rightarrow (ok = \texttt{False})$, which assigns `False` to ok.

In this way we can simultaneously check whether the code of lbl_2 is defined for $expr_1$, and assign $expr_2$ to `result` if $expr_1 \rightarrow_{lbl_2} expr_2$.
– variations of the same trick can be used for all the possible syntactic shapes of the code of lbl_1.

A similar trick can be used to implement the code for $\texttt{Repeat}[lbl_1, lbl_2]$.

In principle, our implementation of the combinators for rules is based on an agreed-upon standard on how to represent the code of non-basic rules, which enables to easily compute the code of the newly declared rule. We do not expose the details here because some of them require a deep understanding of the evaluation and backtracking principles of MATHEMATICA.

4 Applications – Implementing Unification Procedures with Sequence Variables

We have implemented a library of unification procedures for free, flat and restricted flat theories with sequence variables and flexible arity symbols [8] in FUNLOG. The common characteristic features of the procedures is that they are based on transformation rules, applied in a "don't know" non-deterministic manner. Each of the procedures is designed as a tree generation process.

A sequence variable is a variable that can be instantiated by an arbitrary finite sequence of terms. Below we use $\overline{x}, \overline{y}, \overline{z}$ and $\overline{w}$ to denote sequence variables, while x, y, z will denote individual variables. Sequence variables are used together with flexible arity function symbols. Terms and equalities are built in the usual way over individual and sequence variables, and fixed and flexible arity function symbols, with the following restriction: sequence variable can not be a direct argument of a fixed arity function, and sequence variable can not be a direct argument of equality. Substitutions map individual variables to single terms and sequence variables to finite, possible empty, sequences of terms. Application of a substitution is defined as usual. For example, applying the substitution $\{x \mapsto a, y \mapsto f(\overline{x}), \overline{x} \mapsto \ulcorner\urcorner, \overline{y} \mapsto \ulcorner a, f(\overline{x}), b\urcorner\}$ to the term $f(x, \overline{x}, g(y, y), \overline{y})$ gives $f(a, g(f(\overline{x}), f(\overline{x})), a, f(\overline{x}), b)$. The standard notions of unification theory [3] can be easily adapted for terms with sequence variables and flexible arity symbols.

4.1 Free Unification with Sequence Variables and Flexible Arity Symbols

In [9] a minimal complete unification procedure for a general free unification problem with sequence variables and flexible arity symbols was described. The procedure consists of five types of rules: projection, success, failure, elimination and split. They are given in Appendix. All three types of free unification with sequence variables (elementary, with constants and general) are decidable, but infinitary. We do not implement the decision procedure (it requires Makanin algorithm [10] and the combination method [2]). Instead, we put a bound on the unification tree depth and perform a depth-first search with backtracking. Optionally, if the user sets the tree depth bound to infinity, FunLog performs iterative deepening with the predefined depth (by default it is set to 20, but the user can change it), and reports the solutions as they are found.

Nodes in the unification tree are pairs (u, σ), where u is a unification problem together with its context and σ is the substitution computed so far. The successful nodes are labelled only with substitutions.

For instance, the third and fourth elimination rules can be encoded in FunLog as follows:

$$\begin{array}{ll}\texttt{DeclareRule} & [\{\{(\{\texttt{f\_[x\_?SVarQ, s1\_\_\_], f\_[t\_, s2\_\_\_]}\} \mid \\ & \{\texttt{f\_[t\_, s2\_\_\_], f\_[x\_?SVarQ, s1\_\_\_]}\})/;\texttt{Not[SVarQ[t]]}, \\ & \texttt{ctx\_}\}, \sigma\texttt{\_}\}/;\texttt{FreeQ[t, x]} :> \\ & \{\{\{\texttt{f[s1], f[s2]}\}, \texttt{ctx}\}/.\texttt{x} \to \texttt{t}, \\ & \texttt{ComposeSubst}[\sigma, \{\texttt{x} \to \texttt{t}\}], \text{“Elim-svar-nonsvar-1”}];\end{array}$$

$$\begin{array}{ll}\texttt{DeclareRule} & [\{\{(\{\texttt{f\_[x\_?SVarQ, s1\_\_\_], f\_[t\_, s2\_\_\_]}\} \mid \\ & \{\texttt{f\_[t\_, s2\_\_\_], f\_[x\_?SVarQ, s1\_\_\_]}\})/;\texttt{Not[SVarQ[t]]}, \\ & \texttt{ctx\_}\}, \sigma\texttt{\_}\}/;\texttt{FreeQ[t, x]} :> \\ & \{\{\{\texttt{f[x, Apply[Sequence, }\{\texttt{s1}\}/.\texttt{x} \to \texttt{seq[t, x]]]}, \\ & \texttt{f[s2]}/.\texttt{x} \to \texttt{seq[t, x]}\}, \texttt{ctx}/.\texttt{x} \to \texttt{seq[t, x]}\}, \\ & \texttt{ComposeSubst}[\sigma, \{\texttt{x} \to \texttt{seq[t, x]}\}], \text{“Elim-svar-nonsvar-2”}];\end{array}$$

“Elim-svar-nonsvar-1” rule corresponds to the cases with the substitution σ_1 of the third and fourth elimination rules in Fig. 1, and “Elim-svar-nonsvar-2” corresponds to the cases with σ_2 of the same rules. In this manner, we can declare a finite set of FunLog rules that cover all the situations shown in Fig. 1. The non-success rules can be grouped together via the FunLog construct

$$\texttt{SetAlias}[lbl_1 | \cdots | lbl_n, \text{“Non-success”}]$$

where $lbl_1, \ldots, lbl_n$ are labels of all non-success rules. Similarly, the success rules can be grouped into a rule

$$\texttt{SetAlias}[lbl_1 | \cdots | lbl_m, \text{“Success”}]$$

where $lbl_1, \ldots, lbl_m$ are the names of all success rules encoded with FunLog.

We encode computation of one unifier into the following FunLog rule:

$$\texttt{SetAlias}[\text{“Projection”} \circ \texttt{Repeat}[\text{“Non-success”}, \text{“Success”}], \text{“Unify”}].$$

After that, the computation of a unifier of a free unification problem Γ is achieved by the call

$$\texttt{ApplyRule}[\Gamma, \text{“Unify”}]$$

whereas the list of all unifiers is produced by the call

$$\texttt{ApplyRuleList}[\Gamma, \text{“Unify”}].$$

Example 6. For the free unification problem $f(x, b, \overline{y}, f(\overline{x})) \simeq^?_\emptyset f(a, \overline{x}, f(b, \overline{y}))$ the procedure computes the solution $\{\{x \mapsto a, \overline{x} \mapsto \ulcorner b, \overline{x}\urcorner, \overline{y} \mapsto \overline{x}\}, \{x \mapsto a, \overline{x} \mapsto b, \overline{y} \mapsto \ulcorner\urcorner\}\}$. □

Problems like word equations [1] or associative unification with unit element [14] can be encoded as a particular case of free unification with sequence variables and flexible arity symbols. Similarly, associative unification [13] can be translated into a particular case of free unification with sequence variables when projection rules are omitted. Thus, as a side effect, our implementation also provides unification procedures for those problems.

4.2 Flat Unification with Sequence Variables and Flexible Arity Symbols

Flat theory with sequence variables is axiomatized by the equality $f(\overline{x}, f(\overline{y}), \overline{z}) \simeq f(\overline{x}, \overline{y}, \overline{z})$. This theory gives a precise characterization of evaluation behavior of flat functions as it is implemented in the Mathematica system. It was the main motivation to study the flat theory, but then it turned out to have some interesting properties. Namely, it was shown that both matching and unification are infinitary but decidable, and the unification procedure was designed. It should be noted that flat pattern matching of Mathematica implements a restricted, finitary case of matching in the flat theory with sequence variables and flexible arity symbols.

The FunLog implementation of a general flat unification with sequence variables and flexible arity symbols goes along the procedure described in [8]. It combines the rules specific for the flat theory with those specific for the free theory. The implementation does not contain the decision procedure and uses the depth-first search with bounded depth. Since the procedure does not enumerate directly the minimal complete set of unifiers, after answer generation a certain minimization effort is required. As a result, the answer returned by the procedure represents a minimal subset of the complete set of solutions (and not a subset of minimal complete set of solutions).

Example 7. Let $f(\overline{x}) \simeq^?_F f(a)$ be a flat unification problem. It has infinitely many solutions. Our implementation computes the subset

$$\begin{array}{l}\{\{\overline{x} \mapsto a\}, \{\overline{x} \mapsto f(a)\}, \{\overline{x} \mapsto \ulcorner a, f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(a), f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), a\urcorner\},\\ \{\overline{x} \mapsto \ulcorner f(), f(a)\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), a, f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), f(a), f()\urcorner\},\\ \{\overline{x} \mapsto \ulcorner f(), f(), a\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), f(), f(a)\urcorner\}, \{\overline{x} \mapsto \ulcorner a, f(), f()\urcorner\},\\ \{\overline{x} \mapsto \ulcorner f(a), f(), f()\urcorner\}\}.\end{array}$$

of the complete set of unifiers of the problem, with the tree depth set to 4. □

Example 8. Let $f(\overline{x}, g(\overline{x})) \simeq^?_F f(a, b, g(a, f(), b))$ be a general flat unification problem, with f flat and g free. The the procedure computes the unique unifier $\{\overline{x} \mapsto \ulcorner a, f(), b\urcorner\}$. □

4.3 Restricted Flat Unification with Sequence Variables and Flexible Arity Symbols

The restricted flat theory with sequence variables is axiomatized by the equality $f(\overline{x}, f(\overline{y}, x, \overline{z}), \overline{w}) \simeq f(\overline{x}, \overline{y}, x, \overline{z}, \overline{w})$. In this theory only nested terms with at least one non-sequence variable argument can be flattened. Such a restriction makes matching finitary, while other properties of the flat theory are retained.

Example 9. The restricted flat unification problem $f(\overline{x}) \simeq^?_{RF} f(a)$ has two solutions: $\{\overline{x} \mapsto a\}$, $\{\overline{x} \mapsto f(a)\}$. $\square$

We have implemented in FunLog the restricted flat unification procedure described in [8].

5 Conclusion and Future Work

FunLog is intended to be used in areas where problems can be specified conveniently as combinations of abstract rewrite rules. In particular, the package turned out to be useful in implementations of procedures for E-unification.

Obviously, the range of problems which can be tackled with FunLog is very large. We expect to identify more interesting problems which can be easily programmed with transformation rules. But we also expect that our future attempts to solve new problems will reveal new programming constructs which are desirable for making our programming style more expressive.

Currently, we investigate how these programming constructs can be employed to implement provers in the Theorema system [5, 6]. We also believe that our programming constructs could underlie a convenient tool to write reasoners by Theorema users and developers.

Another direction of future work is to introduce control mechanisms for pattern matching with sequence variables. The current implementation of FunLog relies entirely on the enumeration strategy of matchers which is built into the Mathematica interpreter. However, there are many situations when this enumeration strategy is not desirable. We have already addressed this problem in [11, 12] and implemented the package Sequentica with language extensions which can overwrite the default enumeration strategy of the Mathematica interpreter. The integration of those language extensions in FunLog will certainly increase the expressive power of our rule-based system. We are currently working on integrating Sequentica with FunLog.

The current implementation of FunLog can be downloaded from

`http://www.score.is.tsukuba.ac.jp/~mmarin/FunLog/`

References

1. H. Abdulrab and J.-P. Pécuchet. Solving word equations. *J. of Symbolic Computation*, 8(5):499–522, 1990.

2. F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. of Symbolic Computation*, 21(2):211–244, 1996.
3. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
4. P. Borovanský, C. Kirchner, H. Kirchner, and Ch. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, 2001. Also available as Technical Report A01-R-388, LORIA, Nancy (France).
5. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema project: A progress report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning. Proc. of Calculemus'2000*, pages 98–113, St.Andrews, UK, 6–7 August 2000.
6. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the Theorema project. In W. Küchlin, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC'97*, pages 384–391, Maui, Hawaii, US, 21–23 July 1997. ACM Press.
7. The PROTHEO Group. http://www.loria.fr/equipes/protheo/softwares/elan/.
8. T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols.* PhD thesis, Institute RISC-Linz, Johannes Kepler University, Hagenberg, Austria, June 2002.
9. 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, editor, *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of Joint AICS'2002 - Calculemus'2002 Conference*, volume 2385 of *LNAI*, Marseille, France, 2002.
10. G. S. Makanin. The problem of solvability of equations on a free semigroup. *Math. USSR Sbornik*, 32(2), 1977.
11. M. Marin. Functional Programming with Sequence Variables: The Sequentica Package. In J. Levy, M. Kohlhase, J. Niehren, and M. Villaret, editors, *Proceedings of the 17th International Workshop on Unification (UNIF 2003)*, pages 65–78, Valencia, June 2003.
12. 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.
13. G. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 73–90, Edinburgh, UK, 1972. Edinburgh University Press.
14. A. F. Tiu. A1-unification. Technical Report WV-01-08, Knowledge Representation and Reasoning Group, Department of Computer Science, Dresden University of Technology, 2001.

A Rules for Free Unification

Projection:	$s \simeq^?_\emptyset t \rightsquigarrow \langle\langle s\pi_1 \simeq^?_\emptyset t\pi_1,\ \pi_1\rangle, \dots, \langle s\pi_k \simeq^?_\emptyset t\pi_k,\ \pi_k\rangle\rangle$	where $\{\pi_1,\dots,\pi_k\} = \Pi(s \simeq^?_\emptyset t)$.
Success:	$t \simeq^?_\emptyset t \rightsquigarrow \langle\langle \top,\ \varepsilon\rangle\rangle$.	
	$x \simeq^?_\emptyset t \rightsquigarrow \langle\langle \top,\ \{x \mapsto t\}\rangle\rangle$,	if $x \notin \mathit{vars}(t)$.
	$t \simeq^?_\emptyset x \rightsquigarrow \langle\langle \top,\ \{x \mapsto t\}\rangle\rangle$,	if $x \notin \mathit{vars}(t)$.
Failure:	$c_1 \simeq^?_\emptyset c_2 \rightsquigarrow \bot$,	if $c_1 \neq c_2$.
	$x \simeq^?_\emptyset t \rightsquigarrow \bot$,	if $t \neq x$ and $x \in \mathit{vars}(t)$.
	$t \simeq^?_\emptyset x \rightsquigarrow \bot$,	if $t \neq x$ and $x \in \mathit{vars}(t)$.
	$f_1(\tilde{t}) \simeq^?_\emptyset f_2(\tilde{s}) \rightsquigarrow \bot$,	if $f_1 \neq f_2$.
	$f() \simeq^?_\emptyset f(t_1,\tilde{t}) \rightsquigarrow \bot$.	
	$f(t_1,\tilde{t}) \simeq^?_\emptyset f() \rightsquigarrow \bot$.	
	$f(\overline{x},\tilde{t}) \simeq^?_\emptyset f(s_1,\tilde{s}) \rightsquigarrow \bot$,	if $s_1 \neq \overline{x}$ and $\overline{x} \in \mathit{svars}(s_1)$.
	$f(s_1,\tilde{s}) \simeq^?_\emptyset f(\overline{x},\tilde{t}) \rightsquigarrow \bot$,	if $s_1 \neq \overline{x}$ and $\overline{x} \in \mathit{svars}(s_1)$.
	$f(t_1,\tilde{t}) \simeq^?_\emptyset f(s_1,\tilde{s}) \rightsquigarrow \bot$,	if $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \bot$.
Eliminate:	$f(t_1,\tilde{t}) \simeq^?_\emptyset f(s_1,\tilde{s}) \rightsquigarrow \langle\langle g(\tilde{t}\sigma) \simeq^?_\emptyset g(\tilde{s}\sigma),\ \sigma\rangle\rangle$,	if $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \langle\langle \top,\ \sigma\rangle\rangle$.
	$f(\overline{x},\tilde{t}) \simeq^?_\emptyset f(\overline{x},\tilde{s}) \rightsquigarrow \langle\langle f(\tilde{t}) \simeq^?_\emptyset f(\tilde{s}),\ \varepsilon\rangle\rangle$.	
	$f(\overline{x},\tilde{t}) \simeq^?_\emptyset f(s_1,\tilde{s}) \rightsquigarrow$ $\langle\langle f(\tilde{t}\sigma_1) \simeq^?_\emptyset f(\tilde{s}\sigma_1),\ \sigma_1\rangle,$ $\langle f(\overline{x},\tilde{t}\sigma_2) \simeq^?_\emptyset f(\tilde{s}\sigma_2),\ \sigma_2\rangle\rangle$,	if $s_1 \notin \mathcal{V}_{\mathrm{Seq}}$ and $\overline{x} \notin \mathit{svars}(s_1)$, where $\sigma_1 = \{\overline{x} \mapsto s_1\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner s_1, \overline{x}\urcorner\}$.
	$f(s_1,\tilde{s}) \simeq^?_\emptyset f(\overline{x},\tilde{t}) \rightsquigarrow$ $\langle\langle f(\tilde{s}\sigma_1) \simeq^?_\emptyset f(\tilde{t}\sigma_1),\ \sigma_1\rangle,$ $\langle f(\tilde{s}\sigma_2) \simeq^?_\emptyset f(\overline{x},\tilde{t}\sigma_2),\ \sigma_2\rangle\rangle$,	if $s_1 \notin \mathcal{V}_{\mathrm{Seq}}$ and $\overline{x} \notin \mathit{svars}(s_1)$, where $\sigma_1 = \{\overline{x} \mapsto s_1\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner s_1, \overline{x}\urcorner\}$.
	$f(\overline{x},\tilde{t}) \simeq^?_\emptyset f(\overline{y},\tilde{s}) \rightsquigarrow$ $\langle\langle f(\tilde{t}\sigma_1) \simeq^?_\emptyset f(\tilde{s}\sigma_1),\ \sigma_1\rangle,$ $\langle f(\overline{x},\tilde{t}\sigma_2) \simeq^?_\emptyset f(\tilde{s}\sigma_2),\ \sigma_2\rangle,$ $\langle f(\tilde{t}\sigma_3) \simeq^?_\emptyset f(\overline{y},\tilde{s}\sigma_3),\ \sigma_3\rangle\rangle$,	where $\sigma_1 = \{\overline{x} \mapsto \overline{y}\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner \overline{y}, \overline{x}\urcorner\}$, $\sigma_3 = \{\overline{y} \mapsto \ulcorner \overline{x}, \overline{y}\urcorner\}$.
Split:	$f(t_1,\tilde{t}) \simeq^?_\emptyset f(s_1,\tilde{s}) \rightsquigarrow$ $\langle\langle f(r_1,\tilde{t}\sigma_1) \simeq^?_\emptyset f(q_1,\tilde{s}\sigma_1),\ \sigma_1\rangle, \dots,$ $\langle f(r_k,\tilde{t}\sigma_k) \simeq^?_\emptyset f(q_k,\tilde{s}\sigma_k),\ \sigma_k\rangle\rangle$	if $t_1, s_1 \notin \mathcal{V}_{\mathrm{Ind}} \cup \mathcal{V}_{\mathrm{Seq}}$ and $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \langle\langle r_1 \simeq^?_\emptyset q_1,\ \sigma_1\rangle, \dots, \langle r_k \simeq^?_\emptyset q_k,\ \sigma_k\rangle\rangle$.

Fig. 1. $\tilde{t}$ and $\tilde{s}$ are possibly empty sequences of terms; $\Pi(\Gamma)$ is the set of substitutions $\{\{\overline{x_1} \mapsto \ulcorner\urcorner, \dots, \overline{x_n} \mapsto \ulcorner\urcorner\} \mid \{\overline{x_1}, \dots, \overline{x_n}\} \subseteq \mathit{svars}(\Gamma)\}$; $\mathit{svars}(t)$ ($\mathit{vars}(t)$) is the set of all seq. variables (all variables) in t; f, f_1, f_2 are free (fixed or flexible) symbols; g is a new free flexible symbol, if in the same rule f is of the fixed arity, otherwise g is f.