Handling Relations over Finite Domains in the Rule-Based System ELAN

Christophe Ringeissen

LORIA-INRIA
615 rue du Jardin Botanique, BP 101, F-54602 Villers-lès-Nancy Cedex, France
Email: Christophe.Ringeissen@loria.fr

Abstract

We present a methodology for handling efficiently relations over small finite domains in the rule-based programming language ELAN. Usually, a relation is specified as a first-order formula (a constraint) interpreted in a given algebraic structure. The concept of rewriting allows us to implement an algebraic structure in a very elegant way, by using rules for defining operators and predicates. Hence, we can directly obtain a rule-based executable specification computing all tuples of a relation, but in most cases, the related computation is completely inefficient. Indeed, the specification of a relation involves conditional rules, and a lot of rewriting steps fail after being tried. In this paper, we use a constraint solver in finite algebras to transform a naive rule-based ELAN specification of a relation into an efficient rule-based ELAN program with only unconditional rules. Thus, the constraint solver enables us to improve the rule-based computation of a relation.

1 Introduction

In rule-based programming languages [16,4,12,14], programs are Term Rewrite Systems (TRS for short) involving functions defined by rules. Given a function defined by a confluent and terminating TRS, a rule-based system allows us to compute the unique image (result) of any element (query). In this work, we are also interested in rule-based programming with relations. Obviously, there is no real difference between functions and relations. Hence, a $m$-ary function $f : D^m \rightarrow D$ is a $(m+1)$-ary relation over $D$ (that is, a subset of $D^{m+1}$) such that a confluent and terminating TRS computing the result $f(\vec{d})$ for any query $\vec{d} \in D^m$, computes also one element $(\vec{d}, f(\vec{d}))$ of the relation. Conversely, an arbitrary $m$-ary relation $R$ over a domain $D$ may be viewed as a function $f_R : D^m \rightarrow \text{bool}$, such that $\vec{d} \in D^m$ is in $R$ if and only if $f_R(\vec{d})$ is true. Then, a confluent and terminating TRS computing $f_R$ permits to decide whether one tuple of $D^m$ is or not in the relation $R$.

©2000 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.
In this paper, we are not interested in computing one tuple of a relation, or in deciding if one tuple is or not in a relation. Instead, we are interested in computing (efficiently) all tuples of a relation. For this problem, confluent TRS are not enough, it is really interesting to be able to deal with non-confluent TRS, for facing the non-determinism induced by the generation of all tuples of a relation. In this context, the idea is to use a rule-based language integrating both deterministic and non-deterministic computations, using respectively confluent and non-confluent TRS. One possible instance is ELAN, in which one can program a non-confluent TRS, provided that a strategy is defined to control the rule application. The normal forms (in our case, the tuples of the relation) reached using this strategy are collected via a backtracking mechanism as in Prolog-like systems.

We report in this paper how to handle functions and relations (also called constraints) with a rule-based system like ELAN [17,4], when the domain $D$ is assumed to be finite. Especially, we are interested in using strategies for the generation of all tuples in a given relation. Our main contribution is to show the interest of an existing constraint solver in finite algebras [7,21]. Indeed, this solver is very useful for simplifying the relation we want to compute. This simplification is achieved by using the most general solution computed by the solver, which is in fact a parameterization of the relation. The parameterization is given by a tuple of functions defined by rewrite rules directly extracted from the internal data-structure [5] implemented in the constraint solver. Hence, the constraint solver can be used at compile-time as a way to obtain an efficient rule-based computation of a relation initially specified in an algebraic way. The rule-based description of this relation consists of a set of deterministic computation rules for the functions of the parameterization, plus a single non-deterministic rule for the parameterization itself. Then, this piece of code is useful in any rule-based program modelling a problem where the relation is involved in. Constraints and relations over finite domains are of greatest interest for prototyping and checking complex systems specified in multi-valued logics, such as electronic components [6,8].

This paper is organized as follows. In section 2, we give a brief overview of the rule-based system ELAN. Section 3 presents several ways to encode relations in ELAN, either by extension, or by intension using constraints expressed in an algebraic structure. We end this section with an example motivating our approach based on the parameterization of relations. In Section 4, we show how to construct a parameterization of a relation using terms obtained by a constraint solver in finite algebras. These terms are evaluated, after instantiation of variables, according to the rules specifying a functionally complete algebra. In Section 5, we show another better parameterization which uses functions instead of terms. The rule-based description of these functions are directly extracted from the internal data-structure implemented in the constraint solver. The different possible applications of this approach are discussed in Section 6. Finally, we conclude with future works in Section 7.
2 The Rule-Based System ELAN

We assume the reader familiar with rule-based programming languages, like for instance ELAN. In the rest of this paper, an ELAN-like syntax is used to present examples of rule-based programs. Each of these rule-based programs consists of a many-sorted signature and a set of rules involving terms built over this signature. In this syntax, a many-sorted operator declaration \( f : s_1 \times \cdots \times s_n \to s \) is given as follows: \( f(\@, \ldots, \@) : (s_1 \ldots s_n) s \), where \( s_1, \ldots, s_n, s \) are sort symbols and \( \@ \) denotes a placeholder. To an operator symbol \( f \), we may associate several operator declarations, which means that ELAN allows the overloading of operators. Given a set of operator declarations, it is now possible to define rewrite rules. A (global) rewrite rule \( l \to r \) between two terms \( l \) and \( r \) of the same sort \( s \) is written as follows:

```plaintext
rules for s
Var(r) \ Var(l)
local
[R] l => r end
end
```

Let us detail the different components of this rule. Local variables are declared in the `rules` body before the keyword `local` (or `global`) which is used to set the visibility of a rule/declaration. The identifier \( R \) enclosed by \([\] \) brackets is the (optional) rule label. The right-hand side \( r \) contains a term including possibly local variables plus a list of conditions (\( \text{if} \ \@ \)) and assignments (\( \text{where} \ \@ := \@ \)), which will be evaluated in the given order. The argument of `if` is a term of the built-in sort `bool`. If this term is not reducible to `true`, then the rule application fails. The `where` construction has two arguments:

- The first argument is a term, let say \( p \), containing local variables that will be assigned through matching.
- The second argument corresponds to a strategy applied to a term, which will return a (finite) set of terms. The term \( p \) will match successively by a backtracking mechanism each of these terms. If the set of terms is empty, then it is not possible to build a right-hand side for the rule, and the rule application fails.

Finally, local variables are mapped to ground terms, and so they can be substituted in the term of \( r \), in order to obtain a result of the rule application.

We denote by \(<s>\) the sort of a strategy expression processing terms of sort \( s \). A strategy expression of sort \(<s>\) can be defined by rules using a built-in strategy language. The unique built-in construction we use in the paper is `dk(R)`, which computes all terms resulting from the application (at the topmost position) of the rule labelled by \( R \). To perform deterministic computations, we also use the built-in empty strategy `{}`, which applies unlabelled rules by using a leftmost-innermost strategy. Following the order in
which unlabelled rules are given, ELAN tries each of these unlabelled rules and applies the first applicable one. Usually, the set of unlabelled rules is assumed to be a confluent and terminating TRS, so that the application order is not significant.

3 Rule-Based Encoding of Relations

3.1 The Membership Relation

Let us first consider the most elementary relation, which is the unary relation $X \in D$. For this relation, we define a constant operator, let say $D$ of sort $\text{dom}$, and a strategy $\text{inD}$ such that the normal forms of the application of $(\text{inD})$ to the term $D$ are all elements of $D$, of sort $\text{dom}$. This can be easily encoded in ELAN as follows:

```
operators  global
D: dom ;
d_1: dom ;
...
d_n: dom ;
end

rules for dom  global
[genD] D => d_1 end
...
[genD] D => d_n end
end

rules for <dom>  global
[] inD => dk(genD) end
end
```

Then, this relation can be used to define non-deterministic functions like this one:

Example 3.1

```
rules for dom
Y: dom ;
global
[addRule] add(X) => X + Y
    where Y := (inD) D end
end
```

With this definition, $(\text{dk(addRule)}) \ add(x)$ computes the set $\{x + d \mid d \in D\}$. 

4
3.2 Extensional Definition of Relations

For an arbitrary \( m \)-ary relation \( R \), we can use similarly to the previous case a constant operator \( R \) and a strategy operator \( \text{in} R \), such that \( (\text{in} R) R \) enumerates all terms \( R(\vec{d}) \) where \( \vec{d} \in R \). Now, we choose to encapsulate tuples of \( R \) into a \( m \)-ary operator \( R(...). \)

**Example 3.2** Consider for instance the domain \( D = \{0, 1, 2\} \) and the binary relation \( GT \) (“strictly Greater Than”) on \( D^2 \), which can be implemented as follows in ELAN:

```
operators global
0: dom ; [genGT] GT => GT(1,0) end
1: dom ; [genGT] GT => GT(2,0) end
2: dom ; [genGT] GT => GT(2,1) end
GT: rel ;
GT(0,0): (dom dom) rel ;
end
```

```
rules for rel global
[genGT] GT => GT(1,0) end
[genGT] GT => GT(2,0) end
[genGT] GT => GT(2,1) end

end
```

Then, the constant operator \( R \) and the strategy operator \( \text{in} R \) can be used in `if/where` parts of rules.

**Example 3.3** The relation \( GT \) allows us to define the following non-deterministic rule:

```
rules for dom
Y: dom ;
global
[gtRule] gt(X) => Y
  where GT(Y,X) := (inGT) GT
end
```

Normal forms of \( (\text{dk}(\text{gtRule})) \) \( gt(x) \) are elements of \( \{y \mid GT(y, x)\} \). It is very convenient to use relations as patterns (“left-hand sides”) of `where` parts. Hence, the non-deterministic function \( gt \) can be defined in a very natural way.

For now, we have only seen relations defined extensionally. With this approach, the drawback is that a labelled rule is needed for each tuple of the relation. For a relation of significant size, this leads to a program with too many labelled rules, which is difficult to handle by the rule-based system.

3.3 Intensional Definition of Relations

Usually, it is much more convenient to define the relation as a constraint expressed in a constraint language, in which variables are mapped to values and
operators and predicates are interpreted in a first-order algebraic structure. The interpretation of operators and predicates can be implemented by computation rules, whilst the assignment of variables can be performed thanks to the membership relation.

**Example 3.4** Consider the domain $D = \{0, 1, 2\}$ and the binary relation $GEQ$ ("Greater or Equal") on $D^2$, which can be implemented intensionally as follows in ELAN:

operators  global
0: dom ;
1: dom ;
2: dom ;
@>@: (dom dom) bool ;
GEQ: rel ;
GEQ(@,@): (dom dom) rel ;
end

// interpretation of >
rules for bool
X,Y:dom ;
global
  [] 1>0 => true end
  [] 2>0 => true end
  [] 2>1 => true end
  [] X>Y => false end // otherwise
end

rules for rel
X,Y: dom ;
global
[genGEQ] GEQ => GEQ(X,Y)
   where X:= (inD) D
   where Y:= (inD) D
   if X>Y or X==Y
      end
end

rules for <rel>  global
[] inGEQ => dk(genGEQ) end
end

The general form of an *intensional* definition is as follows:

rules for rel
$X_1,\ldots,X_m$: dom ;
global
\[ \text{[genR]} \quad R \Rightarrow R(X_1, \ldots, X_m) \]
where \( X_1 \leftarrow (\text{inD}) \ D \)
\[ \vdots \]
where \( X_m \leftarrow (\text{inD}) \ D \)
if \( R(X_1, \ldots, X_m) \)
end

rules for <rel> global
[] inR => dk(genR) end
end

Now, the main advantage is that we have only a single labelled rule. In counterpart, this rule is conditional. As a consequence, the construction of a right-hand side will be tried for any tuple in \( D^m \). The computation of this condition becomes really problematic if it involves a lot of local (auxiliary) variables.

3.4 Our approach: From Specification to Computation

To illustrate our approach, let us consider a relation defined intensionally by using a constraint in the 2-elements Boolean algebra, where \textit{and}, \textit{or}, \textit{not} operators are denoted respectively \(*, + \) and \(!\). This relation corresponds to an “electronic” gate [15] specified as a combination of \textit{and}/\textit{or}/\textit{not} elementary gates:

\[
\exists E, F, G, H, I, J, K, L, M, N, O, P: \\
!X = G \\
!Y = E \\
F = X * Y \\
I = G + F \\
H = E + F \\
K = !I \\
\]

\[
Gate(X, Y, A, B) \iff \begin{cases} \\
J = !H \\
M = K + F \\
L = J + F \\
O = L + M \\
P = !K \\
N = !J \\
A = P * O \\
B = N * O \\
\end{cases}
\]

The outputs of elementary gates are represented by auxiliary variables,
which are existentially quantified. This relation could be naïvely specified in ELAN as follows:

```
rules for rel
global
[genGate] Gate => Gate(X,Y,A,B)
  where X := (inD) D
  where Y := (inD) D
  where A := (inD) D
  where B := (inD) D
  where E := (inD) D
  where F := (inD) D
  where G := (inD) D
  where H := (inD) D
  where I := (inD) D
  where J := (inD) D
  where K := (inD) D
  where L := (inD) D
  where M := (inD) D
  where N := (inD) D
  where O := (inD) D
  where P := (inD) D
  if !X == G
    and !Y == E
    and F == X*Y
    and I == G+F
    and H == E+F
    and K == !I
    and J == !H
    and M == K+F
    and L == J+F
    and O == L+M
  end
end
rules for <rel> global
[[] inGate] => dk(genGate) end
```

This definition corresponds directly to the specification of Gate. However, this rule-based encoding is too naïve, and so completely inefficient. There are too many variables (16) and the ELAN system would try to compute $2^{16}$ right-hand sides, with of course a lot of redundancies. But the reader can remark that, in this particular case, the equational system specifying Gate can be transformed into a solved form by replacing existentially quantified variables. Moreover, except $X$ and $Y$, all other variables are uniquely defined. Therefore, Gate could be computed as follows:

```
rules for rel
global
[genGate] Gate => Gate(X,Y,A,B)
  where X := (inD) D
  where Y := (inD) D
  where G := () !X
  where E := () !Y
  where F := () X*Y
  where I := () G+F
  where H := () E+F
  where K := () !I
  where J := () !H
  where M := () K+F
  where L := () J+F
  where O := () L+M
  where P := () !K
  where N := () !J
end
```
where \( A := () \ P \ast O \)
where \( B := () \ N \ast O \)
end
end

Our approach leads the following equivalent program, which is even more attractive from a computational point of view.

rules for rel
Y1, Y2: dom;
global
[genGate] Gate => Gate(Y1, Y2, Y2, Y1)
where Y1 := (inD) D
where Y2 := (inD) D
end
end

The parameterization of \( Gate \) in the rule above is directly obtained from the most general solution returned by the constraint solver in finite algebras. It allows us to avoid the use of conditions. Hence, the construction of right-hand sides will never fail. Moreover, the number of auxiliary variables can be minimized.

The reader can check that the new rule generates the same relation as the one specified thanks to the equational constraint. It clearly appears now that \( Gate \) is a very simple “electronic component”, with two inputs \( X, Y \) and two outputs \( A, B \), where \( A \) and \( B \) are obtained by crossing the two inputs \( X \) and \( Y \). This was not easy to detect without solving the constraint specifying the gate.

4 Parameterization by Terms

Given a constraint \( R(X_1, \ldots, X_m) \) (a non-empty relation), the constraint solver in finite algebras computes a unique equational system

\[
X_1 = t_1 \land \cdots \land X_m = t_m
\]

such that

\[
R(X_1, \ldots, X_m) \iff X_1 = t_1 \land \cdots \land X_m = t_m
\]

Terms \( t_1, \ldots, t_m \) are used to build the following parametric definition of \( R \), which is equivalent to the intensional definition given in Section 3.3.

rules for rel
\( X_1, \ldots, X_m: \text{dom} \);
\( Y_1, \ldots, Y_r: \text{dom} \);
global
[genR] R => R(X_1, \ldots, X_m)
where \( Y_1 := (\text{inD}) D \)
\ldots
where \( Y_r := (\text{inD}) \ D \)
where \( X_1 := () \ t_1(Y_1, \ldots, Y_r) \)
...
where \( X_m := () \ t_m(Y_1, \ldots, Y_r) \)
end

rules for <rel> global
[ ] \text{inR} => \text{dk} (\text{genR}) \end
end

Terms \( t_1, \ldots, t_m \) contain some new (local) variables \( Y_1, \ldots, Y_r \) different from the set of variables occurring initially in the relation. The number \( r \) of new variables \( Y_1, \ldots, Y_r \) is the smallest integer such that \( n^r \geq |R| \). For sake of simplicity, we also consider variables \( X_1, \ldots, X_m \) whose values are computed deterministically by evaluating terms \( t_1, \ldots, t_m \) after instantiation of variables \( Y_1, \ldots, Y_r \) by values in \( D \). But, we could also get ride of variables \( X_1, \ldots, X_m \) by considering \( R(t_1, \ldots, t_m) \) as right-hand side. Let us now detail the signature used to build terms \( t_1, \ldots, t_m \). It is a signature of a functionally complete finite algebra as defined in [7,19] with two binary operators + and *, and a unary operator\(^1 \) \( C_d \) for each \( d \in D \):

operators global
Bot: dom alias \( d_1 \)
...// elements of \( D \)
Top: dom alias \( d_n \)
@+@: (dom dom) dom ;
@*@: (dom dom) dom ;
C_@(@): (dom dom) dom ;

rules for dom
\( X,Y: \text{dom} \)
global
[ ] \text{X}+\text{Y} => \text{X} \text{if X}=>\text{Y} \end
[ ] \text{X}+\text{Y} => \text{Y} \text{end} // otherwise (X=\leq Y)
[ ] \text{X}*\text{Y} => \text{X} \text{if X}<=\text{Y} \end
[ ] \text{X}*\text{Y} => \text{Y} \text{end} // otherwise (X=>Y)
[ ] C_{X}X(X) => \text{Top} \end
[ ] C_{Y}X(X) => \text{Bot} \end
end

The functionally complete algebra defined above enables us to represent any function \( F_k : D^r \rightarrow D \) by a term \( t_k \) including (at most) \( r \) variables.

\(^1\) The unary operators \( C_d \)’s are implemented as a binary operator where \( d \) is the first argument.
Example 4.1 Let us consider the following electronic gate [7]:

$$Xor(A, B, X) \iff \begin{cases} 
\exists K, S : \\
B = A \ast S + !A \ast B \\
S = B \ast A + !B \ast S \\
X = !S \\
K = !A \\
\end{cases}$$

The constraint solver computes the following most general solution:

$$\begin{cases} 
A = Y_1 \\
B = Y_2 \\
X = C_0(Y_1) \ast C_1(Y_2) + C_1(Y_1) \ast C_0(Y_2) \\
\end{cases}$$

Then, we use this most general solution to build the ELAN program generating the relation:

```elan
rules for rel
A, B, X: dom ;
Y_1, Y_2: dom ;
global
[genXor] Xor => Xor(A, B, X)
\hspace{1cm} where Y_1 := (inD) D
\hspace{1cm} where Y_2 := (inD) D
\hspace{1cm} where A := () Y_1
\hspace{1cm} where B := () Y_2
\hspace{1cm} where X := () C_0(Y_1) \ast C_1(Y_2) + C_1(Y_1) \ast C_0(Y_2)
end

rules for <rel> global
[] inXor => dk(genXor) end
end
```

With this approach, terms are in general huge and still need to be interpreted, when local variables are instantiated by values. This may be lead to a labelled rule with a very big right-hand side. Moreover, terms are automatically derived from the internal data-structure implementing functions. This data-structure, also called DAG, is a straightforward extension of Binary Decision Diagrams [5] (BDD’s) to the case where \( n \) is possibly greater than 2. The main advantage of DAG’s is the sharing, which is unfortunately lost at the syntactic level of terms.

The two drawbacks mentioned above disappear in the following approach.
5 Parameterization by Functions

Terms $t_1, \ldots, t_m$ used in Section 4 are syntactic representations of functions $F_1, \ldots, F_m$ implemented internally by DAG’s $\varnothing_1, \ldots, \varnothing_m$. These functions can be directly defined by computation rules using DAG’s. Together with computations rules for $F_1, \ldots, F_m$, we will consider the following generation rule:

\[
\text{rules for rel} \\
X_1, \ldots, X_m : \text{dom} ; \\
Y_1, \ldots, Y_r : \text{dom} ; \\
\text{global} \\
[\text{genR}] \ R \Rightarrow R(X_1, \ldots, X_m) \\
\text{where } Y_1 := (\text{inD}) \ D \\
\text{...} \\
\text{where } Y_r := (\text{inD}) \ D \\
\text{where } X_1 := () \ F_1(Y_1, \ldots, Y_r) \\
\text{...} \\
\text{where } X_m := () \ F_m(Y_1, \ldots, Y_r) \\
\text{end}
\]

To explain the construction of computation rules for $F_1, \ldots, F_m$, let us first consider some notations about DAG’s.

**Definition 5.1** A DAG $\varnothing$ is made of two kinds of nodes, namely vertices and leaves. The root of a DAG $\varnothing$ is denoted by $\text{Root}(\varnothing)$. The set of vertices (resp. leaves) is denoted by $\text{Ver}(\varnothing)$ (resp. $\text{Lea}(\varnothing)$). A vertex $v$ is identified by an integer denoted by $\text{id}(v)$, that is $\text{id} : \text{Ver}(\varnothing) \rightarrow \mathbb{N}$ is injective. A vertex $v$ has an arity $n$, which means that it has $n$ sons denoted by $v_1, \ldots, v_n$. A leaf has an arity 0, which means that it has no descendant. The arity of a leaf or a vertex $v$ is denoted by $\text{ar}(v)$. A leaf $v$ is labelled by a value of $D$, denoted $\text{lab}(v)$. A vertex $v$ is labelled by a variable, denoted by $\text{lab}(v)$. The ordered list of variables occurring as labels of vertices of the DAG rooted by $v$ is denoted by $\text{args}(v)$. This list is empty if $v$ is a leaf. A vertex representing a variable $Y$, is a vertex $v$ denoted by $\text{vertex}(Y)$ such that its label is $Y$, and for each $i = 1, \ldots, n$ the $i$-th son is the leaf labelled by $d_i$.

The rewrite rules associated to a DAG $\varnothing$ is:

\[
\text{GenRules}(\varnothing) := \bigcup_{v \in \text{Ver}(\varnothing)} \text{GenRules}_{\text{Ver}}(v)
\]

where $\text{GenRules}_{\text{Ver}}(v)$ are rules associated to a vertex $v$ of $\varnothing$:

\[
\text{GenRules}_{\text{Ver}}(v) := \bigcup_{i=1}^{n} \left\{ [ f_{\text{id}(v)}(d_i, \text{tail}(\text{args}(v))) \Rightarrow \text{Rhs}(v_i) \text{ end } } \right\}
\]
where \( \text{tail}(l) \) is the tail of the list \( l \), and \( \text{Rhs}(v) \) is the right-hand side associated to a vertex \( v \):

- \( \text{Rhs}(v) := \text{lab}(v), \) if \( \text{ar}(v) = 0; \)
- \( \text{Rhs}(v) := Y, \) if \( v = \text{vertex}(Y); \)
- \( \text{Rhs}(v) := f_{\text{id}(v)}(\text{args}(v)), \) otherwise.

**Remark 5.2** It is important to note that functions \( f_{\text{id}(v)} \) are defined by pattern-matching on their first argument.

For each \( k = 1, \ldots, m \), the function \( F_k \) corresponds to \( f_{\text{id}(\text{Root}(d_k))} \).

**Proposition 5.3** Given a DAG \( d \), the set \( \text{GenRules}(d) \) is a confluent and terminating TRS computing the function represented by \( d \). The computation of \( \text{GenRules}(d) \) is linear in the size of \( d \) (where the size of \( d \) is \( |\text{Ver}(d)| \)).

**Example 5.4** (Example 4.1 continued). The DAG’s of the most general solution of \( \text{Xor} \) can be depicted as follows:

\[
\begin{align*}
\varnothing(A) &= 0 \\
\varnothing(B) &= Y_2 \\
\varnothing(X) &= Y_1 \\
\end{align*}
\]

Given these DAG’s, we construct the following **ELAN** program:

```
operators global

F3(@,@): (dom dom) dom ;
F3-2(@): (dom) dom ;

rules for dom
Y_1, Y_2: dom ;
global
[] F3(0,Y_2) => Y_2 end
[] F3(1,Y_2) => F3-2(Y_2) end
[] F3-2(0) => 1 end
[] F3-2(1) => 0 end
end

rules for rel
A, B, X: dom ;
```

13
$Y_1, Y_2$: dom;

**global**

$[\text{genXor}] \ Xor \Rightarrow Xor(A, B, X)$

where $Y_1 := (\text{inD}) D$

where $Y_2 := (\text{inD}) D$

where $A := () Y_1$

where $B := () Y_2$

where $X := () F3(Y_1, Y_2)$

end

end

**rules for <rel> global**

[] inXor $\Rightarrow dk(\text{genXor})$ end

end

With this approach, we have only one labelled rule and many unlabelled rules. This is really interesting for efficiency matter, since the ELAN compiler \cite{23,20} is much more efficient with unlabelled rules than with labelled ones. Programs generated by this way are potentially good benchmarks for the ELAN compiler.

6 Applications

6.1 Pre-processing of Relations

On the top of the constraint solver, we have implemented GenRules which generates the rule-based definition of functions represented internally as DAG’s. Now, the constraint solver does not only compute the most general solution of a constraint, but it also computes automatically an ELAN module for the generation of the relation. For the moment, this new functionality is used as a pre-processing tool for the automatic generation of an efficient rule-based program (a module) from an algebraic specification of a relation $R$ we would like to integrate in ELAN. Then, the related constant $R$ and strategy $\text{inR}$ are useful in a more general ELAN program where we want use $R$, especially in a where with pattern, like in Example 3.3.

6.2 Transformation of Programs

Another possibility would be to use the parameterization (by functions) in order to transform a (naive) rule-based ELAN specification into an improved rule-based ELAN program. The specification defines the finite algebra of interest and the relation. This information will be used to initialize and then to run the constraint solver. The output of the constraint solver provides us the related rule-based ELAN computation.

The constraint solver is able to deal with rule-based ELAN specifications
of the following form.

**Input Specification**

Until now, we have only seen examples where variables are instantiated by values through the membership relation. It is interesting to consider that variables are constrained by more complex relations. To this aim, the relation we want to transform is defined according to \( q \) independent relations \( R_1, \ldots, R_q \), for which each \( R_i \) involves a tuple of variables \( \vec{X}_i = X^1_i, \ldots, X^a_{i(R)} \), such that the different tuples of variables \( \vec{X}_i \)'s are pairwise disjoint: \( \vec{X}_i \cap \vec{X}_{i'} = \emptyset \), for \( i, i' = 1, \ldots, q, i \neq i' \).

\[
\begin{align*}
\text{rules for rel} \\
\vec{X}_1, \ldots, \vec{X}_q: \text{dom} \\
\text{global} \\
[\text{genR}] R \Rightarrow R(\vec{X}_1, \ldots, \vec{X}_q) \\
&\quad \text{where } R_1(\vec{X}_1) := (\text{inR}_1) R_1 \\
&\quad \quad \ldots \\
&\quad \text{where } R_q(\vec{X}_q) := (\text{inR}_q) R_q \\
&\quad \quad \text{if } R(\vec{X}_1, \ldots, \vec{X}_q) \\
\end{align*}
\]

Relations \( R_i \)'s are defined similarly. For a \text{where} with simply a variable as left-hand side, its right-hand side is \( (\text{inD}) \ D \).

**Output Computation**

The conditional rule \text{genR} is transformed into an unconditional one. Each component \( X^j_i \) of the relation is generated by a function \( F^j_i \) computed by ELAN unlabelled rules. The constraint solver yields the following labelled rule \text{genR}:

\[
\begin{align*}
\text{rules for rel} \\
\vec{X}_1, \ldots, \vec{X}_q: \text{dom} \\
Y_1, \ldots, Y_r: \text{dom} \\
\text{global} \\
[\text{genR}] R \Rightarrow R(\vec{X}_1, \ldots, \vec{X}_q) \\
&\quad \text{where } Y_1 := (\text{inD}) \ D \\
&\quad \quad \ldots \\
&\quad \text{where } Y_r := (\text{inD}) \ D \\
&\quad \quad \text{where } X^1_1 := () \ F^1_1(Y_1, \ldots, Y_r) \\
&\quad \quad \quad \ldots \\
&\quad \quad \text{where } X^a_{i(R)} := () \ F^a_{i(R)}(Y_1, \ldots, Y_r) \\
&\quad \quad \quad \ldots \\
&\quad \text{where } X^1_q := () \ F^1_q(Y_1, \ldots, Y_r) \\
&\quad \quad \quad \ldots
\end{align*}
\]
where
\[ X_{q}^{\text{ar}(Rq)} : = \emptyset \quad F_{q}^{\text{ar}(Rq)}(Y_{1}, \ldots, Y_{r}) \]
end

6.3 Integration by using Reflection

The main drawback of the previous transformation is that it is a compile-time process. The best solution would be to generate at run-time the rules used to parameterize a relation (again, by using the constraint solver in finite algebras). Then, these rules could be directly executed via reflection [11,12] in the rule-based language. This is only possible in a rule-based language having both the ability of calling an external solver and of executing rules generated by this external solver.

There is no reflection mechanism yet available in ELAN, even some experiments have been already realized by using the current ELAN exchange format [3]. However, it is already possible in ELAN to call the constraint solver as an external process [18]. The (unique) result computed by the solver is a rule-based program which has to be executed by a new call to ELAN.

7 Conclusion

We have discussed in this paper several approaches for dealing with relations and constraints over finite domains in a rule-based programming language like ELAN in which both deterministic and non-deterministic computations are possible. On the one hand, we have deterministic computations for free using a confluent and terminating TRS given by unlabelled rules. On the other hand, the non-determinism is controlled thanks to strategies, like the one which collects all normal forms of a non-confluent terminating TRS. In this context, the most appropriate approach consists in implementing the relation as a non-deterministic (labelled) rule associated to a parameterization, together with a set of deterministic (unlabelled) rules for functions involved in the parameterization. This rule-based description can be obtained using a constraint solver in finite algebras. The output of this solver has been modified in order to produce this rule-based program generating the relation. Then, this program is executable in ELAN in a much more efficient way than the naïve implementation derived by specifying, with ELAN rules, the algebra and the relation. For the moment the solver is used as a separate tool. An interesting perspective would be to study the run-time integration of this tool in ELAN programs. We believe this would be a quite good application for the reflection mechanism (not yet available in ELAN), since the solver generates an executable ELAN program.

This work can be considered as a further step towards the integration of constraints and rules [13,18,1,10,9] in the same programming environment. Here, we are interested in the generation of tuples of a relation by using a
rule-based program obtained from a constraint solver in finite algebras. This solver is also helpful for the generation of propagation rules [22], which are of greatest interest to perform constraint solving and constraint propagation [2].

References


