Towards a Methodology for Rule-Based Programming

Carlos Castro, Claude Kirchner

Towards a Methodology for Rule-Based Programming

Carlos Castro
Universidad Técnica Federico Santa María
Av. España 1680, Valparaíso, Chile
e-mail: Carlos.Castro@inf.utfsm.cl
&
Claude Kirchner
INRIA Lorraine & LORIA
615, rue du jardin botanique, BP 101, 54602 Villers-lès-Nancy CEDEX, France
e-mail: Claude.Kirchner@loria.fr

January 7, 2003

Abstract

In this paper, we propose general guidelines that could be considered to define transformation rules when programming using a rule-based approach. We apply the proposed steps for solving some typical problems in Computer Science. Through these examples, we also show how clear and easy it is to prove properties such as correctness, completeness and termination following these guidelines when a rule-based paradigm is used.

1 Introduction

The notion of rules is in the essence of computer programming. This paradigm has been used since long time ago in Computer Science to represent operations on data structures. Transformation rules are widely used in several domains under different denominations: Inference Rules in Logic, Rewrite Rules in Rewriting Logic, Transition Rules in Automata Theory, Production Rules in Artificial Intelligence, and so on. In the last years, the development of rule-based programming languages such as ELAN [4], Maude [9], Claire [5], ILOG RULES [15], ASF+SDF [13], and CHR [14], has allowed the application of these concepts in solving real-life problems.

The basic idea of a transformation rule is very natural. When a simple operation is applied on an input producing an output an obvious way to express that it is just to link input and output by an arrow. We say that the input is transformated into the output and the arrow must express the direction of

this transformation. This simple idea expressing what is carried out represents an interesting alternative with respect to traditional procedural programming where we have to specify how the operation is carried out. However, when using a rule-based approach to develop large-scale applications that involve many transformations, we need a framework to support the design and programming tasks and no such a guide is available up to now.

A lot of work has been carried out for developing methodologies in other paradigms: structured programming [12], structured analysis [21], structured design [22], and object-oriented programming [1] are examples of methods to develop software using a procedural approach. The work presented in this paper can be considered as a first attempt to see rule-based programming from a software engineering point of view.

Providing a general methodology to deal with all aspects of rule-based programming seems to be too ambitious. However, in order to advance in this direction, we think that one of the first things that we should consider is the definition of the rules themselves. That is why the main motivation of this work is to provide a framework which could be considered to define transformation rules when a rule-based approach is used for programming. The main contribution of this work is two-folds: on one hand, we propose a general framework for rule-based programming, on the other hand, we explain the basic concepts behind the use of rules as a programming tool.

This paper is organized as follows: section 2 presents two simple examples of sorting and search problems in order to informally introduce the notions of transformation and control that can be clearly separated in rule-based programming. Section 3 details all concepts involved in the application of a rule in the framework of term rewriting. Section 4 strengh the role that control plays when developing programms using a rule-based approach. Section 5 presents the five steps defining the methodology that we propose for programming in this paradigm. Section 6 presents the application of the methodology for solving some simple examples. Finally, in section 7, we conlude the paper and give some perspectives for further research.

2 Transformations and Control

In this section, we present two examples introducing two fundamental concepts in our framework. The first one mainly concerns the concept of transformation, a key notion in rule-based programming. The second one discusses the role of control that allows to guide the application of the transformations.

2.1 A First Example: Sorting

Sorting is a basic operation that everybody uses daily: think of sorting your 52 playing cards, of sorting by dates the file of your bank statements or to sort by size the elements of a Russian doll. We sort tennis players, favorite songs, football teams, etc. ...

How can we describe the sorting operation in a simple and concise way? Let us take for example the list of numbers

$$12\ \ 4\ \ 34\ \ 5\ \ 7\ \ 1\ \ 23\ \ 45\ \ 3\ \ 5\ \ 2\ \ 54\ \ 6\ \ 87$$

and assume we would like to sort them in increasing order (from left to rigth). A simple way to describe this task is the following:

— *choose* 2 *different places in the list*: These could be, for example, the ones determined by the arrows:

$$12\ \ 4\ \ 34\ \ \overset{\downarrow}{5}\ \ 7\ \ 1\ \ 23\ \ 45\ \ 3\ \ 5\ \ 2\ \ \overset{\downarrow}{54}\ \ 6\ \ 87$$

— *if the chosen elements are in increasing order, do nothing*: We are exactly in this situation, so we do nothing.

— *if the chosen elements are in decreasing order, exchange them*: Assuming we selected:

$$12\ \ 4\ \ 34\ \ \overset{\downarrow}{5}\ \ 7\ \ 1\ \ 23\ \ 45\ \ 3\ \ 5\ \ \overset{\downarrow}{2}\ \ 54\ \ 6\ \ 87$$

we will transform this list into

$$12\ \ 4\ \ 34\ \ \overset{\downarrow}{2}\ \ 7\ \ 1\ \ 23\ \ 45\ \ 3\ \ 5\ \ \overset{\downarrow}{5}\ \ 54\ \ 6\ \ 87$$

Now, we have to decide when to stop making these "try to exchange" instructions, so we give the last order:

— *choose again two places and perform the previous instructions, until no more exchange is possible.*

We would like to describe these simple instructions in a formal enough way to have a computer mechanically executing them.

The first main abstract notion needed to design a model of the sorting algorithm is the one of *variable*. A variable is a *name* that allows to designate *any* element of a given set. For example, when we talk about a "boy" in a classroom "boy" designates anyone of the boys in the given classroom. It is classical to set "let x be a variable in N the set of natural numbers" to designate one of the natural numbers.

In our sorting example, we can introduce 3 variables, L_1, L_2, and L_3, ranging on lists and 2 variables, E_1 and E_2, ranging on natural numbers.

Now we can express the fact that we have chosen 2 different elements in a list in the following way:

$$\overbrace{12\ \ 4\ \ 34}^{L_1}\ \ \overbrace{5}^{E_1}\ \ \overbrace{7\ \ 1\ \ 23\ \ 45\ \ 3\ \ 5\ \ 2}^{L_2}\ \ \overbrace{54}^{E_2}\ \ \overbrace{6}^{L_3}\ \ 87$$

and the transformation above can be expressed as

for all lists L_1, L_2, L_3 *and elements* E_1, E_2,
transform any list of the form $L_1\ E_1\ L_2\ E_2\ L_3$ *into the list* $L_1\ E_2\ L_2\ E_1\ L_3$,
provided $E_1 > E_2$.

Now instead of using this verbose description of the transformation, we use an horizontal arrow to denote the fact that we transform a list into another one, i.e.:

$$L_1\ E_1\ L_2\ E_2\ L_3 \quad \rightarrow \quad L_1\ E_2\ L_2\ E_1\ L_3 \quad \textbf{if } E_1 > E_2$$

We have just written a so called *rewrite rule*. In this example, the rewrite rule transforms a list into another list, its left-hand side is $L_1\ E_1\ L_2\ E_2\ L_3$, its right-hand side is $L_1\ E_2\ L_2\ E_1\ L_3$ and, its firing *condition* is $E_1 > E_2$. Carrying out a transformation is what we call the application of a rewrite rule.

When defining rewrite rules we are just expressing the transformations that are carried out, we do not specify the way they are applied. In the next section we discuss the notion of control.

2.2 How to search?

Sliding-tile puzzles have been widely used by the Artificial Intelligence community as examples to show different search heuristics. This kind of puzzle consists of a $n \times n$ frame holding $n^2 - 1$ distinct movable tiles, and a blank space, as presented in figure 1.

3	7	6
4		1
2	5	8

Figure 1: A sliding-tile puzzle.

Any tile that is horizontally or vertically adjacent to the blank may move into the blank position. An operator is any such legal move. Given an initial state, for example, the one in figure 1, the goal is to find the sequence of moves that transforms the initial state into some given final state, for example, the one presented in figure 2.

Considering the general rules of this game, the four valid operators available to solve this puzzle can be graphically defined as presented in figure 3 where X is a variable that can take any integer value in $[1, \ldots, 8]$.

These operators define the transformations that can be applied on a given frame: the right move represents a tile moving to the blank position on its right, the left move represents a tile moving on the blank position on its left, and so

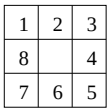

Figure 2: A final state for the sliding-tile puzzle.

Figure 3: The four valid operators for the sliding-tile puzzle.

on. It is important to note that the four rules express general transformations, we are note defining specific operations. This is in the essence of rule-based programming, we are interested in what are the operations, we are not representing how the operations are carried out.

States that can be reached by the application of all available transformations are called neighborg states. Applying left, up, down, and right moves on the initial state of figure 1 we obtain the four neighborg states in the second level of the search tree presented in figure 4.

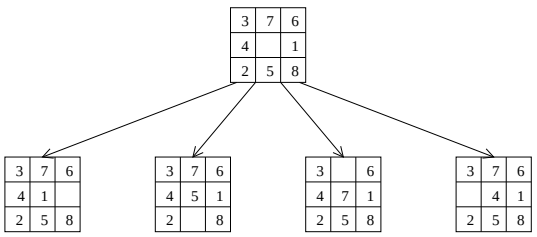

Figure 4: All neighborgs of the initial state.

Search heuristics are defined based on the criteria used to choose the neighborg state that will be considered as the next current state to be analysed. Once a new current state is determined the process is repeatedly applied until a final state, if any, is reached. Search heuristics express the order in which the search tree is explored. In other words, search heuristics express the order in which transformation rules are applied to explore the search tree. This is the notion of control, the second fundamental concept in rule-based programming.

For example, if we were interested in visiting all possible neighborg states that can be reached in one step from all states in the second level of the search

tree in figure 4 we should apply all transformations on every state to obtain the search tree presented in figure 5.

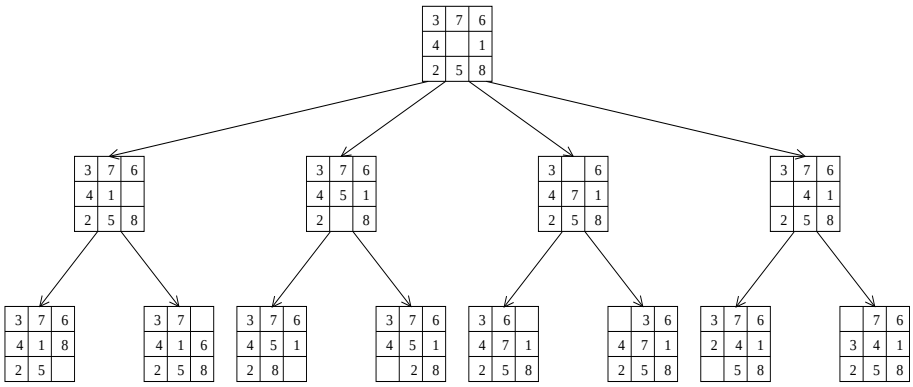

Figure 5: Breadth-First Search tree for the sliding tile puzzle.

In this way we are following the Breadth-First Search (BFS) heuristics that visits all states in a level before going down to states in the next level.As usual in this kind of heuristics, we do not show neighborg states that have already been visited in previous levels of the search tree. It is easy to note that strictly speaking each state of the second level should have three neighborgs in the third level: if the state in the second level was created using a left move on the initial state, the same initial state could be reached applying the right move on the state in the second level, and so on. This shows a common situation in programming: non termination. In order to avoid this behaviour a memory is generally used to record the states that have already been visited and that have no interest in being visited again.

When we are interested in obtaining just the first neighborg that can be reached from the current state using all possible transformations we are talking about the Depth-First Search (DFS) strategy. The first three levels of the search tree following this heuristics is presented in figure 6.

In order to control the application of the transformations we need tools that allow to express the desired search heuristics. In the context of rewriting, the tools that control the application of a set of rewrite rules are called strategy languages, a notion that was introduced in the rewriting systems giving origin to the Computational Systems [17, 18]. In the last ten years, a lot of research has been done on the definition of strategy languages. The ELAN system, the first one implementing these ideas, provides operators for rule selection, non-determinism, iteration, and cooperation of several rewrite systems [3, 2]. The notion of strategy is also used in others systems based on rewriting logic. Strategies of Maude are based on reflexivity aspects of rewriting logic [8, 10]. The system ASF+SDF offers several built-in term-traversal strategies with a recursion operator that allow to define strategies used in various program transformation

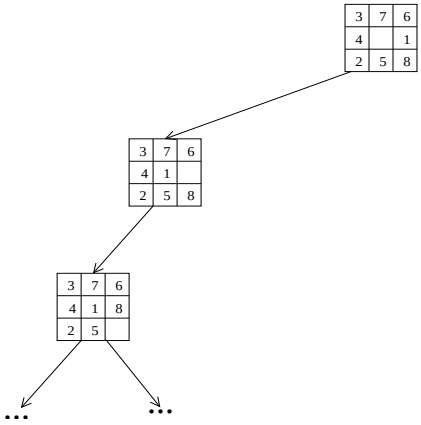

Figure 6: Depth-First Search for the sliding-tile puzzle.

techniques [20, 19].

The very simple example presented in this section about search heuristics in Artificial Intelligence shows the relation between two fundamental concepts that are always used in Computer Science: transformation and control. In this example, transformations are represented by the four rules presented in figure 3 and the notion of control refers to the order of the application of these rules. Depending on the paradigm used to program computers these notions are implemented by different data structures and sentences available in the host language. However, the difference between transformation and control is not always explicit in all these frameworks.

The need for a clear distinction between transformation and control has been well recognized in the last years. The rule-based paradigm seems to be a good framework in order to make this difference. Because of their simplicity and clarity to express transformations, they provide strong advantages over procedural approaches. Specifically, when we are interested in proving properties, such as completeness, correctness and termination of a program, they appear as an unavoidable tool to be considered. These advantages are mainly due to the explicit separation between transformations and control that is inherent to the rule-based approach.

3 Transformations by Term Rewriting

An elementary term rewriting step can be seen as involving three basic ideas: selection of the rule to be applied, verification of the applicability of the selected rule, and actions derived by the application of the rule. In this section, we will first introduce the basic ideas of term rewriting through an example and then we will formally define these concepts.

Let us consider the well-known factorial function:

$$n! = 1 \times \ldots \times n$$

This function can be formally described by the following set of rules:

$$\begin{array}{lcl} fac(0) & \rightarrow & 1 \\ fac(n) & \rightarrow & fac(n-1) \times n \\ & & \textbf{if} \;\; n >= 1 \end{array}$$

Now, suppose we are interested in computing 3!, so we should specify the following query:

$$fac(3)$$

Trying to apply the set of rules we verify:

- The first rule $fac(0)$ cannot be applied on the input $fac(3)$ because the parameters of the symbol fac are different integer numbers.
- The second rule $fac(n)$ could be applied because $fac(3)$ can be equated to $fac(n)$ for $n = 3$ and the condition $3 \geq 1$ is true.

In this very simple example, we can identify the three fundamental concepts:

Rule selection When the set of available rules contains more than one element a strategy has to be specified to select the rules that will be tried and, eventually, applied. In this example, we just tried them in the order in which they are specified, but of curse we could be interested in tools to control their application for specifying other orders.

Matching Once a rule has been selected, we have to verify its applicability with respect to the input. This second basic idea of term rewriting is called matching. Verify the applicability of the rule $fac(0)$ on the input $fac(3)$ is quite easy because both ground formulas, i.e., formulas without variables, are different. Verifying the applicability of the rule $fac(n)$ on the input $fac(3)$ is a little bit more complicated. Due to the fact that $fac(n)$ contains the variable symbol n, we have to verify if this variable can be instantiated to the appropiate value. In this case, it is possible to instantiate n to 3 and so we say that there exists a substitution that solves the matching problem. We also say that the query matches the second rule. So, the fact that the second rule can be applied is due to the substitution.

Replacement Once a substitution is found, and the matching between a rule and the input is possible, we can go ahead and proceed to replace the input by the right-hand side of the second rule where n is replaced by 3:

$$3 \times fac(3-1)$$

This third basic idea, where the query is replaced by the right-hand side of a rule, is the action expressed by the rewrite rule. After simplifications we obtain the following reduced expresion [1]:

$$3 \times fac(2)$$

This is the application of a rule, this is also called a rewrite step or a one step rewriting. Of course, we can continue reducing the query because we can also apply rewrite rules in sub expresions.

$$\begin{array}{l} fac(3) \rightarrow\ 3 \times fac(3-1) \\ \qquad\qquad\ \downarrow \\ \qquad\quad 3 \times fac(2) \quad \rightarrow\ 3 \times 2 \times fac(2-1) \\ \qquad\qquad\qquad\qquad\ \downarrow \\ \qquad\qquad\qquad\quad 6 \times fac(1) \quad \rightarrow\ 6 \times 1 \times fac(1-1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\ \downarrow \\ \qquad\qquad\qquad\qquad\qquad\quad 6 \times fac(0) \quad \rightarrow\ 6 \times 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 6 \end{array}$$

This very simple example also shows the role that control play concerning efficiency: if we always try first the rule $fac(0)$ we will fail three times, each one after applying the rule $fac(n)$.

In the following, we formalize the notions of term, rewrite rule, substitution, matching, and rewrite step.

Terms First-order terms are built on a vocabulary of function symbols and variable symbols. Let $\mathcal{F} = \cup_{n \geq 0} \mathcal{F}_n$ be a set of symbols called function symbols, each symbol f in $\mathcal{F}_n$ has an arity which is the index of the set $\mathcal{F}_n$ it belongs to, it is denoted $arity(f)$. Elements of arity zero are called constants, and it is always assumed that there is at least one constant. Given a denumerable set $\mathcal{X}$ of variable symbols, the first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the smallest set containing $\mathcal{X}$ and such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $arity(f) = n$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $i \in [1 \ldots n]$.

Rewrite Rule A rewrite rule is an ordered pair of terms denoted by

$$l \quad \rightarrow \quad r$$

The terms l and r are called the left-hand side and the right-hand side of the rule, respectively. The direction of the arrow express the fact that the term on the left-hand side is transformed into the term on the right-hand side when the rule is applied. Rewrite rules can also be applied under satisfiability of conditions. The rule

$$\begin{array}{lll} l & \rightarrow & r \\ & & \textbf{if} \quad c \end{array}$$

[1] We could also specify rules to simplify arithmetics expressions such as $3 - 1 = 2$.

transforms the term on the left-hand side into the term on the right-hand side if the condition c is satisfied. When one needs to refer rules a label can be associated to each one. The labelled rule

$$[\ell] \quad l \quad \rightarrow \quad r$$

represents a rewrite rule with label ℓ. Of course, in general, we can talk about labelled conditional rewrite rules:

$$\begin{array}{llll} [\ell] & l & \rightarrow & r \\ & & & \textbf{if} \ \ c \end{array}$$

Finally, a rewrite system is a set of such rewrite rules.

Matching and Substitution A substitution is an application on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ uniquely determined by its image of variables. It is thus written out as $\{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ when there are only finitely many variables not mapped to themselves. The application of a substitution $\sigma = \{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ to a term is recursively defined as follows:

1. if t is a variable x_i for some $i = 1, \ldots, n$ then $\sigma(t) = t_i$,
2. if t is a variable $x \neq x_i$ for all $i = 1, \ldots, n$ then $\sigma(t) = t$,
3. if t is a term $f(u_1, \ldots, u_k)$ with $u_1, \ldots, u_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $f \in \mathcal{F}$, then $\sigma(t) = f(\sigma(u_1), \ldots, \sigma(u_k))$.

Finding a substitution σ such that $\sigma(l) = t$ is called the matching problem from l to t.

Rewrite Step A rewrite step consists in finding a subterm of the input term that matches the left-hand side of the rule and replace that subterm by the right-hand side of the rule instantiated by the match. Formally, given a rewrite system R and an input term t, if there exists

- a rule $l \rightarrow r$ in R,
- a position ω in t, and
- a substitution σ satisfying $t_{|\omega} = \sigma(l)$

we say that t rewrites to t' where $t' = t[\omega \leftarrow \sigma(r)]$. This is called a rewrite step and it is denoted by $t \rightarrow_R t'$. When either the rule, the substitution or the position need to be specified, a rewrite step is denoted by

$$t \quad \rightarrow_R^{\omega, \sigma, l \rightarrow r} \quad t'$$

When considering conditional rules of the form

$$\begin{array}{lcl} l & \rightarrow & r \\ & & \textbf{if } c \end{array}$$

the rule is applicable on a term t provided there exists a matching substitution σ of l on $t_{|\omega}$ such that

$$\sigma(c) \quad \rightarrow^* \quad true$$

In the example:

$$fac(3) \quad \rightarrow_1^{fac(n) \ \rightarrow \ n \times fac(n-1)} \quad 3 \times fac(3-1)$$

4 The Notion of Control

Let us consider the problem of finding the minimum value in a list of different integer numbers. Four cases have to be taken into account:

- The list is empty.
- The list contains only one element.
- The head is less than the second element of the list.
- The head is greater than the second element of the list.

In order to detect these four cases we could define, respectively, the following rules:

$$\begin{array}{rcl} min(nil) & \rightarrow & nil \\ min(x1.nil) & \rightarrow & x1 \\ min(x1.x2.l) & \rightarrow & min(x1.l) \\ & & \textbf{if } x1 < x2 \\ min(x1.x2.l) & \rightarrow & min(x2.l) \\ & & \textbf{if } x1 > x2 \end{array}$$

Applying the *min* operator on a given list of integers $3.2.7.nil$, the basic mechanism of recursive programming carries out all the work using the fourth, then the third, and finally the second rule as follows:

$$min(3.2.7.nil) \ \rightarrow \ min(2.7.nil) \ \rightarrow \ min(2.nil) \ \rightarrow \ 2$$

When using unlabelled rules we are just computing applying all rules as much as possible. The user has not control on the application of the rules. It is a fixed strategy, expressed in the implementation, that guides the computation. We can see that recursive programming is a concept widely used in this paradigm.

However, if we want to be able to control the application of the transformations we should make use of labelled rules:

$$
\begin{array}{rrcl}
\textbf{[MinNil]} & min(nil) & \rightarrow & nil \\
\textbf{[MinSingleton]} & min(x1.nil) & \rightarrow & x1 \\
\textbf{[MinHead]} & min(x1.x2.l) & \rightarrow & min(x1.l) \\
 & & & \textbf{if } x1 < x2 \\
\textbf{[MinSecond]} & min(x1.x2.l) & \rightarrow & min(x2.l) \\
 & & & \textbf{if } x1 > x2
\end{array}
$$

In this case, given a list we need to specify the order in which we want to apply this set of rules. To carry out this specification we need a set of strategy operators, that is what is called a strategy language. In the domain of term rewriting, a very well-known environment is the ELAN system whose strategy language involves operators to deal with rule selection, non-determinism, and iteration [4]. Among these operators we can mention the operators: $\textbf{first}(S_1, \ldots, S_n)$ that returns all results of the application of the first strategy S_i that can be applied, and $\textbf{repeat}(S)$ that applies repeatedly the strategy S until it cannot be applied anymore. Using these strategy operators we could specify an algorithm for searching the minimum value in a list in the following way:

$$\textbf{repeat}(\textbf{first}(\textbf{MinNil}, \ \textbf{MinSingleton}, \ \textbf{MinHead}, \ \textbf{MinSecond}))$$

As in this example, we do not always know *a priori* the best strategy to solve a given problem. However, some times specific knowledge is available about the problem. Considering again the factorial function presented at the begining of section 3, we could also define labelled rules as follows:

$$
\begin{array}{rrcl}
\textbf{[Factorial0]} & fac(0) & \rightarrow & 1 \\
\textbf{[FactorialN]} & fac(n) & \rightarrow & n \times fac(n-1) \\
 & & & \textbf{if } n >= 1
\end{array}
$$

Using the fact that the rule **Factorial0** is applied just once, when $n = 0$, and in all other cases we must apply the rule **FactorialN**, we could specify the following strategy in order to be more efficient:

$$\textbf{repeat}(\textbf{first}(\textbf{FactorialN}, \ \textbf{Factorial0}))$$

When using labelled rules we have the possibility of controling the application of the rules. However, even if this flexibility seems to be interesting, we are not always interested in specifying all transformations. Applying two times the rule **FactorialN** on $fac(3)$ we will obtain the following derivation:

$$
\begin{array}{lll}
fac(3) \rightarrow 3 \times fac(3-1) & & \\
\qquad\qquad \downarrow & & \\
\qquad 3 \times fac(2) & \rightarrow 3 \times 2 \times fac(2-1) & \\
\qquad\qquad & \qquad\qquad \downarrow & \\
\qquad\qquad & \qquad 6 \times fac(1) &
\end{array}
$$

It seems obvious that the simplification of arithmetics expressions such as $3 - 1 = 2$, $3 \times 2 = 6$, and $2 - 1 = 1$, are not interesting at all in the context of this problem. Transformation rules for treating these expressions could be defined but the application of these rules could be carried out without control by the programmer. This exemplifies the notion of deduction that consists in the application of a rewrite step under the control of the programmer, and the notion of computing that consists in the application of a set of rules without control by the programmer until a fixed point is achieved. The concepts of rewriting for deduction and rewriting for computing are implemented in the ELAN system through the distinction of labelled rules that have to be controled by the programmer and unlabelled rules that are applied by the system following a predefined strategy *left-most inner-most*.

In this section, we have shown that the approach of rule-based programming plus strategies allows the user to specify different strategies in order to carry out testing, prototyping, proving, and deduction. The definition of strategy languages, control languages, and coordination languages, is currently an active research topic in several domains of Computer Science.

5 Five Steps for Defining Transformation Rules

Based on the approach first proposed by Jouannaud and Kirchner for solving unification problems [16], the general idea of problem solving using a rule-based approach is to apply a set of transformation rules until no rule can be applied anymore. In order to understand how to define a set of transformation rules we first need to introduce some concepts:

- A *basic form* represents the syntax of all terms that we will transform.
- A *normal form* is a basic form that cannot be reduced by a set of transformation rules.
- A *property* is a predicate that has to be satisfied by a basic form.
- A *solved form* is a basic form satisfying a given property.
- A *set of solutions* to a given problem corresponds to the set of all solved form for the problem.

Now, we can say that solving a problem using a rule-based approach consists in transforming an initial basic form by the repeated application of a set of transformation rules until a normal form is obtained. The results so obtained are said to be normal with respect to the set of rules under consideration and they have to correspond to the solved form, the set of solutions, we are looking for.

In the following, we detail the five steps for defining transformation rules.

1. We first define a *basic form* for the problem. Considering all terms that can be built from the given signature, we specify those we are interested in and that will be treated by the set of transformations. This basic form represents the syntactical structure of the terms.

2. Then, we define a *solved form* that represents the result that we want to obtain. This solved form corresponds to a particular instance of the basic form plus the semantical characteristics that represent a solution. The solved form represents the set of terms that correspond to the solutions to the problem.

3. Once we have defined the input and output of the problem solving process, we start defining the transformations to be done on the set of terms. These transformations must be defined in such a way that, when applied to an initial term in basic form, they will allow us to obtain the desired result, a solved form, as shown in figure 7.

Figure 7: Input/Output.

In order to obtain such a set of transformation rules, for each term that is not in solved form we define a rule that transforms a term into another equivalent. The notion of equivalence has, on one hand, a syntactical meaning and, on the other hand, a semantical meaning. From the syntactical point of view, all rules have to maintain the term in basic form, i.e., they do not have to modify the syntactical form of the terms. From the semantical point of view, all rules have to maintain the set of solved forms, i.e., they do not have to modify the set of terms that is considered as solution to the problem. In this way, each rule that transforms a problem P into a problem P' has to comply with the following properties:

(a) Preserve the set of terms in basic form.
This proof is carried out just comparing the resulting term with respect to the defined basic form in order to verify that the sort of the resulting term is equivalent to the sort of the basic form.

(b) Preserve the set of solutions to the problem

$$Sol(P') = Sol(P)$$

This proof can be carried out verifying the correctness and completeness properties for each rule:

i. A rule transforming a problem P into a problem P' verifies the *correctness* property if

$$Sol(P') \subseteq Sol(P)$$

ii. A rule transforming a problem P into a problem P' verifies the *completeness* property if

$$Sol(P) \subseteq Sol(P')$$

A rule complying with the correctness and completeness properties means that it preserves the set of solutions to the problem.

4. After a set of correct and complete rules is defined, we verify that the repeated application of the set of rules will always terminate. This proof is carried out using either a particular sequence of application of the rules or a class of sequences. In other words, we prove that the application of the set of rules following a given sequence will always terminate. In order to verify this termination property, some ad-hoc complexity measurements have to be defined. If the application of the set of rules, following a given sequence, strictly reduces these measurements, the computation process will terminate each time the same sequence or class of sequences is followed.

5. Finally, we have to prove that the normal forms obtained with respect to the set of terms correspond to the solved forms. In other words, we have to prove that the output produced by the computation process semanticaly corresponds to the expected results. This proof can be carried out in two steps:

   (a) We prove that every solved form is a normal form with respect to the set of terms, i.e., none of the rules can be applied. This proof can be easily verified because the rules had to be defined in such a way that they transform every set of terms that is not in solved form.

   (b) We prove that every set of terms that is not in solved form can be reduced by some rule. This proof is a little more complicated because rules are defined in order to transform the whole set of terms that is not in solved form into another equivalent. In case this property is not satisfied, the solution is obvious: we have to define a new rule that has to take in charge the transformation of the set of terms that is not in solved form.

The five steps that we have described can be graphically represented by the diagram in figure 8:

This diagram explicitly shows the nature of the transformation rule definition process. The approach described in this section was first developed by Jouannaud and Kirchner for solving unification problems [16], and, later on

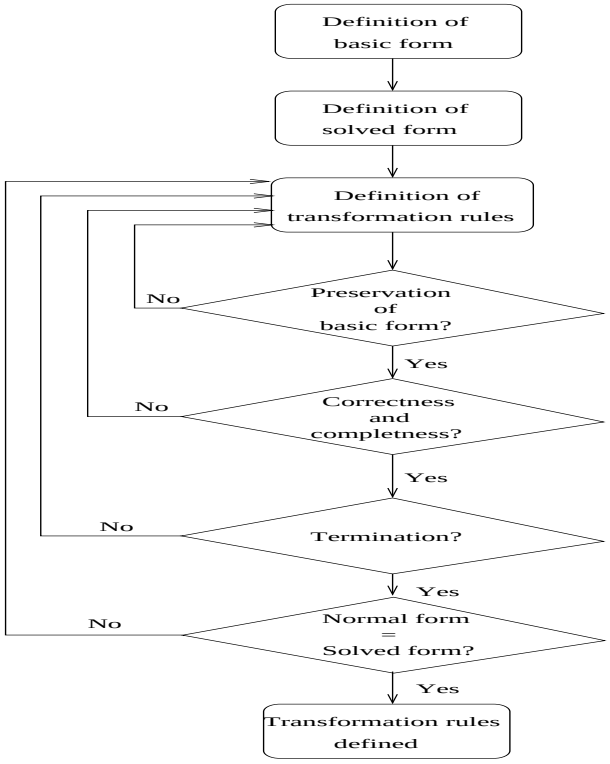

Figure 8: Transformation rule definition process.

it was also used for modeling constraint solving by Comon, Dincbas, Jouannaud, and Kirchner [11]. Based on these works, Castro proposed rewrite rules and strategies for solving Constraint Satisfaction Problems [6, 7]. All these works deal with modeling complex problems. In the next section, we apply this methodology step by step for solving some typical problems in mathematics and computer science that should be easier to understand.

6 Applications

In this section, we exemplify the use of the proposed methodology by its application on solving instances of three kinds of problems occuring in mathematics and computer science: computation, searching, and sorting.

Computation This kind of application involves the determination of values for mathematical expressions. Interest calculation, in the domain of finances, and square root computation, when solving second degree equations, are some basic examples where mathematical algorithms are expressed by simple sums

or recursive definitions. A typical example of such algorithms is the Fibonacci function defined as follows:

$$fibonacci(n) = 1,\ 1,\ 2,\ 3,\ 5,\ 8, \ldots,\ (n-1) + (n-2)$$

In general, for a natural number n, the value of $fibonacci(n)$ is determined by adding $fibonacci(n-1)$ and $fibonacci(n-2)$, and assuming that both $fibonacci(0) = 1$ and $fibonacci(1) = 1$.

Let us start defining the operator $fib : N \rightarrow N$ that takes an integer parameter and returns an integer value. A basic form for a term t is defined by

$$t ::= t + t \mid fib(t) \mid n$$

where $n \in N$ and $n > 1$.

For $n > 1$, and assuming that $fib(0) = 1$ and $fib(1) = 1$, computing $fib(n)$ means that we look for a value x that satisfies the following property $P(x)$:

$$P(x) : x = fib(n)$$

The set of solutions is so defined by

$$Sol = \{x \in N \mid x = fib(n)\}$$

A solved form is a basic form without fib operators and satisfying the property P.

We define the following set of labelled rules to compute this function:

$$\begin{array}{lllll} [\textbf{Fib0}] & fib(0) & \rightarrow & 1 & \\ [\textbf{Fib1}] & fib(1) & \rightarrow & 1 & \\ [\textbf{FibN}] & fib(n) & \rightarrow & fib(n-1) + fib(n-2) & \\ & & & \textbf{if}\ \ n > 1 & \end{array}$$

The first two rules just express the definition of the function for $n = 0$ and $n = 1$, and the third one expreses the recursive definition for $n > 1$ based on $n - 1$ and $n - 2$.

Now, we prove preservation of basic form, correctness and completeness properties, termination, and equivalence between normal forms and solved forms.

- The first two rules preserve the basic form because they replace a fib operator by an integer value, both of them corresponding to basic forms. The third rule replaces a fib operator by the addition of two fib operators and so the basic form is preserved too.

- We can easily verify that these rules are correct and complete: by definition $Sol(fib(0)) = \{1\}$ and $Sol(fib(1)) = \{1\}$, and these two cases are expressed by the first two rules, respectively. The third rule directly implements the recursive definition of the Fibonacci function, and so it does not eliminate any solution neither it add any solution.

- In order to prove termination we could define as complexity criterion the size of the parameter of the fib operators: the first two rules simply eliminate the fib operator, and the third rule replaces a fib operator by the addition of two fib operators, each one involving a parameter that is less than the input parameter, so the application of this set of rules will terminate.

- As a solved form is just an integer value and all rules are applied on a fib operator, no rule can be applied on a term in solved form. As the basic form involves fib operators with an integer parameter n, we easily realize that for $n = 0$, $n = 1$, and $n > 1$, the first, the second, and the third rules will be applied, respectively, and so the set of normal forms is equivalent to the set of solved forms.

Search The problem of finding the maximum value in a list of integer numbers is similar to the one presented in section 4 where we were interested in finding the minimum value in a list of integer numbers.

We define the operator $max : list \rightarrow N$ that takes a list as parameter and returns an integer value. A basic form for a term t is defined by

$$t ::= t.t \mid max(t) \mid n$$

where $n \in N$.

When computing $max(l)$ for a list l we look for a value x that satisfies the following property $P(x)$:

$$P(x) : \forall y \ (y \in l \rightarrow x > y)$$

The set of solutions is so defined by

$$Sol = \{x \in l \mid \forall y \ (y \in l \rightarrow x > y)\}$$

A solved form is a basic form that only contains an integer value satisfying the property P.

As we explain in section 4, four cases have to be taken into account when searching a minimum or a maximum value in a list:

- The list is empty.

- The list contains only one element.

- The head is less than the second element of the list.

- The head is greater than the second element of the list.

In order to detect these four cases we could define, respectively, the following labelled rules:

$$\begin{array}{rrcl}
[\textbf{MaxNil} & max(nil) & \rightarrow & nil \\
[\textbf{MaxSingleton}] & max(x1.nil) & \rightarrow & x1 \\
[\textbf{MaxHead}] & max(x1.x2.l) & \rightarrow & max(x1.l) \\
 & & & \textbf{if } x1 > x2 \\
[\textbf{MaxSecond}] & max(x1.x2.l) & \rightarrow & max(x2.l) \\
 & & & \textbf{if } x1 < x2
\end{array}$$

In the following, we present proofs for preservation of basic form, correctness and completeness properties, termination, and equivalence between normal forms and solved forms.

- Preservation of the basic form is verified by the second rule because it replaces a max operator by an integer value. The third and the fourth rules also preserve the basic form because they replace the max operator on the entry by another max operator.

- The rule **MaxNil** just considers an empty list where a maximum value does not exist. The rule **MaxSingleton** obviously verifies the completeness and correctness properties because $Sol(max(x1.nil)) = \{x1\}$ as $x1$ is the only element of the list. The rule **MaxHead** eliminates the second element of the list when this element is less than the first one, so this rule is complete because it does not eliminate any solution ($Sol(max(x1.x2.l)) \subseteq Sol(max(x1.l))$) and, of course, it is also correct because it does not add any one ($Sol(max(x1.l)) \subseteq Sol(max(x1.x2.l))$). Following the same reasoning we can prove correctnes and completness of the rule **MaxSecond** because it considers the opposite case when the first element is less than the second one and the elimination of the first one preserves the set of solutions.

- In order to prove termination we define as complexity criterion the number of elements in the list. This measurement is not modified by the rules **MaxNil** and **MaxSingleton**, and is strictly decreased by the rules **MaxHead** and **MaxSecond** because they eliminate one element each time they are applied. So the repeated application of these rules will terminate.

- In order to prove that the set of solved forms is equivalent to the set of solutions we can first note that a solved form is just an integer value and so no rule can be applied because all rules are applied on a max operator. On the other hand, the basic forms that are not in solved form correspond to a max operator with a list l as parameter. If the list is empty the rule **MaxNil** will be applied and if the list contains just one element the rule **MaxSingleton** will be applied. If the list contains two or more elements the rules **MaxHead** or **MaxSecond** will be applied depending on which of the first two elements is greater than the other. So, assuming that

the list contains different integer values, every term in basic form will be treated by this set of rules.

Sorting The last example we present in this paper concerns the problem of sorting a list of integer numbers. As an alternative to the very naive algorithm we use in section 2.1, we will implement a selection sort algorithm. When a list of n integer values has to be sorted in increasing order, this simple approach first visits the list looking for the maximum value and place it in the first position. Then, the same idea is applied on the list of the remaining $n-1$ elements, and so on.

As, in the previous example, we have already formally defined a set of rules for searching the maximum value in a list, we will assume that a *max* operator is available. Let us also assume that an operator $deletemax : list \rightarrow N$ is defined by the following set of rules:

$$\begin{array}{lcl} deletemax(nil) & \rightarrow & nil \\ deletemax(x1.nil) & \rightarrow & nil \\ deletemax(x1.x2.l) & \rightarrow & x2.deletemax(x1.l) \\ & & \textbf{if } \ x1 > x2 \\ deletemax(x1.x2.l) & \rightarrow & x1.deletemax(x2.l) \\ & & \textbf{if } \ x1 < x2 \end{array}$$

For sake of space, we will not prove all properties for this set of rules. However, following the definition of the set of rules for the *max* operator, these proofs are obvious. It is interesting to note that in this case transformations are carried out without control by the user because we only use unlabelled rules that are applied following a built-in strategy (for example, *left-most inner-most* in the case of the ELAN language).

Now we define the operator $sort : list \rightarrow list$ that takes a list as parameter and returns a list. So, the basic form in this case is just a list of integer values.

When computing $sort(l)$ for a list l we look for lists $(a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$ that satisfy the property $a_1 > a_2, a_2 > a_3, \ldots, a_{n-1} > a_n$.

The set of solutions is so defined by

$$Sol = \{(a_1, \ldots, a_n) \in \Pi\ l \mid \forall i, j\ (i \in 1 \ldots n, j \in i+1 \ldots n \ \rightarrow \ a_i > a_j)\}$$

where $\Pi\ l$ denotes the set of all permutations of the elements of l.

A solved form is a basic form $(a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$ that satisfies the property $a_1 > a_2, a_2 > a_3, \ldots, a_{n-1} > a_n$.

Now, using the operators *max* and *deletemax*, we can easily define the following rules to sort a list l:

$$\begin{array}{lrcl} [\textbf{SortList}] & sort(nil) & \rightarrow & nil \\ [\textbf{SortList}] & sort(l1.nil) & \rightarrow & max(l1).l2 \\ & & & \textbf{where } \ l2 := [\textbf{SortList}] \ \ sort(deletemax(l1)) \end{array}$$

The first rule is defined to treat an empty list and, of course, it would be applied just once. The second one considers lists with at least one element and it would also be applied just once, but its recursive definition will do all the work. This second rule places the maximum value in the input list in the first position of the output list and then it appends the rest of the elements of the list, but these elements are first sorted before the append operation is carried out. All this is done thanks to the **where** sentence.

Finally, we prove properties for this set of rules.

- Preservation of the basic form is verified by these rules because they replace the list on the entry by another list.
- The set of solutions to $sort(l)$ are lists of the form $(a_1, a_2, a_3, \ldots, a_{n_1}, a_n)$ where $a_1 > a_2, a_2 > a_3, \ldots, a_{n_1} > a_n$. The first rule, considering an empty list, does not modify the set of solutions. The first time that the second rule is applied on $l1$ it will change the order of $l1$ assigning the maximum value in $l1$ to the head. This assignment will be fixed and the rest of the list will be treated again by the recursive application of the same rule and so this recursive application of the second rule will not eliminate any solution.
- In order to prove the termination of this set of rules we can use as complexity measurement the number of elements in the list used as parameter of the *sort* operator. The first rule eliminates the *sort* operator, and the second rule call itself recursively with the *sort* operator involving the same list with all elements but one which has been eliminated, so the recursive application of this rule will terminate.
- A solved form corresponds to a list and so no rule can be applied on a solved form because both rules are applied on a *sort* operator. The basic forms that are not in solved form correspond to a *sort* operator with a list l as parameter. If the list is empty the first rule will be applied, and if the list contains at least one element the second rule will be applied. So, the set of solved forms is equivalent to the set of normal forms.

7 Conclusions

In this paper, we have proposed a general framework for defining transformation rules when a rule-based approach is used for computer programming. This work can be seen as a contribution to the development of a general methodology to guide the development of software using a rule-based approach.

As a further work, a wider domain of applications should be considered in order to validate the applicability of these ideas. Problems related to modularity that arrive when developing large-scale software should also be taken into account. Also, the definition of operators available in current strategy languages should be standard. Indeed, a lot of research has to be done in order to study the complexity of the application of a transformation rule.

Although rule-based programming provides several advantages over procedural programming, its use in practice is not yet wide enough. We strongly think that expanding the use of this technology depends on the effort that the scientific community carries out to develop theoretical and practical tools allowing programmers learn and apply it efficiently. This task involves changes in the way that most of people develop programms nowadays.

Aknowledgements. This work has been partially supported by a grant INRIA-CONICYT from the cooperation program between the French and Chilean governments. The first author has also been supported by grants from the French Ministery of Education and Research and the Chilean National Science Fund through the project FONDECYT 1010121.

References

[1] Grady Booch. *Object Oriented Analysis and Design with Applications.* The Benjamin/Cummnings Publishing Company, 2 edition, 1994.

[2] Peter Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, October 1998. Also available as Technical Report 98-T-326 of the Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA.

[3] Peter Borovanský and Carlos Castro. Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, pages 379–398, Pont-à-Mousson, France, September 1998. Electronic Notes in Theoretical Computer Science. Also available as Technical Report 98-R-305 of the Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA.

[4] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, Japan, April 1998. World Scientific.

[5] Yves Caseau and François Laburthe. *Introduction to the CLAIRE Programming Language.* Département Mathématiques et Informatique, Ecole Normale Supérieure, Paris, France, September 1996.

[6] Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, June 1998. Also available as Technical Report 98-R-308 of the Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA.

[7] Carlos Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, December 1998. Also available as Technical Report 98-T-315 of the Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA.

[8] M. Clavel. *Reflection in general logics, rewriting logic, and Maude.* PhD thesis, University of Navarre, Spain, 1998.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI International, Menlo Park, USA, March 1998.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Metalevel Computation in Maude. In C.Kirchner and H.Kirchner, editors, *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, pages 3 – 24. Electronic Notes in Theoretical Computer Science, September 1998.

[11] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A Methodological View of Constraint Solving. *Constraints*, 4(4):337–361, December 1999. Also available as Technical Report 99-R-306 of the Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA.

[12] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[13] A. Deursen, J. Heering, and P. Klint. *Language Prototyping.* World Scientific, 1996. ISBN 981-02-2732-9.

[14] Thom Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basic and Trends. Selected Papers of the 22nd Spring School in Theoretical Computer Sciences*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, Châtillon-sur-Seine, France, May 1994.

[15] Ilog. Business Rules: ILOG Rules White Paper, October 2002.

[16] J.-P. Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge, MA, USA, 1991.

[17] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Implementing Computational Systems with Constraints. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *Proceedings of The First Workshop on Principles and Practice of Constraint Programming, Providence, R.I., USA*, pages 166–175, 1993.

[18] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In Pascal Van Hentenryck and Vijay Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, chapter 8, pages 131–158. The MIT press, 1995.

[19] E. Visser and Z. Benaissa. A Core Language for Rewriting. In C.Kirchner and H.Kirchner, editors, *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, pages 25 – 44. Electronic Notes in Theoretical Computer Science, September 1998.

[20] E. Visser and B. Luttik. Specification of Rewriting Strategies. In *Proceedings of The Second International Workshop on the Theory and Practice of Algebraic Specifications, ASF+SDF'97*, September 1997.

[21] Edward Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

[22] Edward Yourdon and Larry L. Constantine. *Structured Design.* Computing Series. Yourdon Press, 1979.