Electronic Notes in Theoretical Computer Science 44 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 20 pages

A Pattern-Matching Compiler

Pierre-Etienne Moreau, Christophe Ringeissen

LORIA-INRIA
BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
Email: moreau@loria.fr,ringeiss@loria.fr

Marian Vittek

Institut of Informatica
Mlynska dolina 842 15 Bratislava, Slovakia
Email: vittek@fmph.uniba.sk

Abstract

Implementation of a rule-based transformation engine consists of several tasks with various abstraction levels. We present a new tool called mtom for the efficient implementation of rule-based transformations. This engine should help to bridge the gap between rewriting implementations and practical applications. It aims at implementing well-identified parts of complex applications where the use of rewriting is natural or crucial. These parts are specified using rewrite rules and integrated with the rest of the application, which is kept in a classical imperative language such as C, C++ or Java. Our tool, which can be viewed as a Yacc-like pre-processor, does not depend on a given term representation, rather it accepts implementation of terms (or term like data-types) of yet existing applications and it permits to define and execute rewrite rules upon those types. From our experiences, this system is well-suited for industrial use as well as for implementations of rule-based languages. The paper introduces several features supported by mtom.

1 Introduction

Only few industrial applications are implemented using tools that perform rewrite rules. However, rewriting is of greatest interest for symbolic and algebraic computations, program transformations, compiler constructions, etc. A couple of existing rule-based programming languages (such as ASF+SDF [17], Cafe-OBJ [12], ELAN [5] or Maude [7]) have been yet used in the development of large applications but their connections with industrials are rather limited. This is probably not because of a lack of will on both sides (industrials as well as rule-based programming promoters). For example, several serious attempts have been made to use ELAN in industrial applications. But those attempts

failed on ordinary practical tasks, like implementing particular input/output requirements, improving the user-interface, reusing existing tools, linking to existing libraries, etc. In general it seems to us that academical rewriting environments are not well-suited for real-life programming.

To bridge the gap between rewriting implementations and practical applications, we propose a new tool called mtom (for Many-To-One Matching). Its design follows our industrial experiences and our works on the efficient compilation of rewriting [30,16]. From our point of view, this new tool is very useful for implementing well-identified parts of complex applications. These parts are specified using rewrite rules and integrated with the rest of the application, which is kept in a classical imperative language such as C, C++ or Java, called *goal language* in the rest of the document.

This tool aims at compiling functions defined in a declarative way into functions written in an imperative programming language. The latter will then become executable with well-known compilers. Hence, the programmer simply defines his function by using his favorite programming language but also with the help of rewrite rules which will be applied through the mechanism of pattern-matching. Our tool can be viewed as a Yacc-like pre-processor translating this *rule-based* imperative function into a *true* imperative function.

When trying to integrate a black-box tool into an existing system, one of the main bottlenecks comes from data conversion and the flexibility offered to the user. One of the main originality of our system is to allow a flexible term representation. The programmer can use (or re-use) his own data-structures for terms and then execute rule-based functions defined upon those data-structures. We propose to access terms using only a simple user-defined *Application Interface* (API). This allows the programmer to work on multiple term representations, including user-defined data-structures as well as existing built-in data-types. The proposed tool is also able to cooperate with a variety of memory management methods and does not impose any particular evaluation strategy. It is general enough to permit the user to implement his own rewriting strategies. The innermost normalisation remains the default strategy and we will use it in most of the cases.

The paper is organized as follows. After this introduction, we start with a brief overview of our approach that has led to the development of mtom (Section 2). The notations and definitions are given in Section 3. Then, we present in Section 4 how to encode innermost rewriting using mtom. The specification language of mtom is given in Section 5. Its main features are discussed in Section 6. Eventually, we compare our tool with existing works (Section 7), and we conclude with some final remarks (Section 8).

2 Overview

Our practical experiences show that the construction of right hand sides of rules can be written in standard programming languages in a straightforward

way and we believe that there is no need for a particular new syntax to represent them. Simply, let us suppose that for each n-ary function symbol occurring in the signature, we have implemented a *term constructing function* (n-ary) `f`, written in the goal language and whose evaluation returns the term $f(t_1, \ldots, t_n)$, when called with $t_1, \ldots, t_n$ as arguments. In this case, the construction of a given term (right hand side of a rule), can be directly implemented in the goal language by several nested function calls. For example, in order to construct the term $f(g(a))$ we just write the code: `f(g(a()))`.

As illustrated in the rest of the paper, it is possible to express rewriting strategies (innermost, outermost or user-defined) directly in the goal language. Concerning non-deterministic strategies, our experience showed us that it can also be expressed in the goal language in a straightforward and understandable way [30,20].

However, given a ground term, the test whether a rule is applicable or not is more complex. It usually requires term decomposition and some kind of indexing technique for efficiency reasons. When looking at a code generated by a rule-based language compiler, it is clear that the compilation of the (many-to-one) matching process makes the resulting code complex and difficult to understand. If one tries to code the rewriting directly in a goal language, this part of code would be hard to understand and hardly maintainable. For these reasons it is desirable that this part is not written directly in the goal language but is generated from a more compact and abstract representation.

Because of the above discussion, our tool will only process left hand sides of rules, which are called *patterns*. Given a set of patterns, the tool generates a many-to-one matching function in the goal language. Each pattern has a *semantic action*, which is executed if the given pattern matches the input term. The semantic action is directly written in the *goal language* and is supposed to construct the right hand side of the rewrite rule.

Our tool can be seen as a Yacc-like pre-processor. The input is a set of patterns together with their semantic actions (written in the goal language). The output is a function, say `mtom_main(input_term)`, providing many-to-one matching of the `input_term` against the given set of patterns. Similarly to Yacc, semantic actions are inserted directly into the generated function. In order to keep the tool general and simple, the `mtom_main` function performs matching only at the top position of the `input_term` (see Section 4 and Section 6.5 for full normalisation implementations). If a particular pattern matches the `input_term`, then the corresponding semantic action is executed.

This also explains why we call our tool mtom, since it provides basically a Many-To-One Matching compilation. In order to be as general as possible, we restrict our tool to this unique functionality. Hence, even elementary rewriting strategies must be implemented by the user. However, as we will see in Section 4 some support for innermost strategies is provided.

mtom is not designed for a particular goal language: the function implementing the pattern-matcher can be expressed in an abstract language and

then translated into a specific goal language such as C, C++ or Java. This code can then be compiled by a standard compiler and linked together with other files of the whole project. An important question is: what will be the form of rewrite rules given to mtom? We will give an answer to this question after explaining how to implement efficiently an innermost rewriting.

3 Notations and Definitions

Let us briefly introduce notations for concepts that will be used along this paper. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables, denoted by x, y, z, etc. Positions in a term are represented as sequences of integers and denoted by greek letters ϵ, ν. The empty sequence ϵ denotes the position associated to the root and so it is the position of the top symbol. The subterm of t at position ν is denoted by $t_{|\nu}$. The replacement at position ν of the subterm $t_{|\nu}$ by t' is written $t[\nu \hookleftarrow t']$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. A term t is said to be *linear* if no variable occurs more than once in t. A *substitution* is an assignment from a finite subset of $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. A rewrite rule is a pair of terms with variables, called respectively the left and right-hand sides. The *root* of a rewrite rule is the top symbol of its left hand side. A function symbol is *defined* if it occurs as a root of a rewrite rule, otherwise, it is a *constructor*.

Let us consider a set of rewrite rules R and a term t. Computing the normal form of a term t w.r.t. a rewrite system R consists in applying successively the rewrite rules of R, at any position, until no more applies. The existence and uniqueness of normal forms require the rewrite system R to be respectively terminating and confluent. An *innermost normalisation* is a process which consists in computing normal forms of subterms (using recursively innermost normalisation) before applying any rule to the top position of the current term. It corresponds to the intuition that from all the possible positions where a rewrite rule can be applied, an innermost one is chosen at each rewrite step. There may be several such innermost positions, further refinements are then possible by defining for example the left-most innermost normalisation.

In order to make things more readable, several fonts will be used along this paper: mathematical $font$ to express mathematical concepts, mtom `font` to write mtom expressions, and goal `font` to write expressions from the goal language.

4 Efficient Innermost Normalisation

In this section we will describe mtom's support for innermost term-normalisation strategy, especially we will demonstrate how this strategy can be efficiently implemented using the standard evaluation mechanism of imperative program-

ming languages. This compilation schema has been successfully used in several compilers such as ASF+SDF [28] and ELAN [30,16].

The main idea consists in generating *normalising term construction functions* for each defined functional symbol in the signature. A *normalising term construction function* for a n-ary symbol $f \in \mathcal{F}$ is a n-ary function `f` with the following property: let $t_1, \ldots, t_n$ be terms in normal forms, then `f` applied on terms $t_1, \ldots, t_n$ returns the normal form of the term $f(t_1, \ldots, t_n)$. Moreover, this normal form has to be computed by an innermost normalisation of $f(t_1, \ldots, t_n)$. Normalising term construction functions refine the construction seen in Section 2. As mentioned previously, with those functions, a term construction (for example, of $f(g(a))$) can be easily implemented as nested function calls (e.g. `f(g(a()))`). Except that, with normalising functions, the normal form of this term is directly constructed. Since the evaluation mechanism of the goal language is assumed to be a call-by-value evaluation mechanism, it ensures that arguments are evaluated before being passed to the calling function. Therefore, the nested function calls lead to an innermost normalisation.

As a special support for innermost normalisation, mtom generates a skeleton of normalising term construction function for each defined function symbol. There are two reasons that explain why mtom does not generate a term construction function for constructor symbols of the signature and is only restricted to defined symbols. The first main reason is that mtom does not construct terms, and a term construction function for a constructor symbol always results in the construction of a constructor-based term. Another reason is that we suppose that the description of a rewrite system can be organized into several files sharing the same signature. In this case, the generation of functions for all symbols in each of those files would generate conflicts at link time.

The whole concept is easy to understand on this classical example: let us consider the two following rewrite rules, where $Zero, Suc, Plus$ are function symbols and x, y are variables:

$$Plus(Zero, y) \rightarrow y \qquad Plus(Suc(x), y) \rightarrow Suc(Plus(x, y))$$

If we assume using the ATerm library [27], this rewrite system can be compiled into C language as follows:

```
ATerm zero()        { return(ATmake("Zero")); }
ATerm suc(ATerm x)  { return(ATmake("Suc(<term>)", x)); }
ATerm plus(ATerm u, ATerm y) {
      ATerm x;
      if(ATmatch(u, "Zero")) {
        return(y);
      } else if(ATmatch(u, "Suc(<term>)", &x)) {
        return(suc(plus(x,y)));
      }
}
```

The two first functions (`zero` and `suc`) use the `ATmake` function, which only consists in constructing a term for the pattern introduced as a string (first argument). The third function (`plus`) is more interesting since it decomposes its first argument `u`, using the `ATmatch` function [1], and returns either `y` (corresponding to the first rule), or `suc(plus(x,y))` (corresponding to the second rule). As we can see, the compilation of innermost normalisation of a term is quite easy and straightforward. If we need to get the normal form of a term, say $Plus(Suc(Zero()), Plus(Suc(Zero()), Zero))$, we just write in the goal language the expression `plus(suc(zero()), plus(suc(zero()),zero()))`. The result of its evaluation will be the term $Suc(Suc(Zero))$.

This example illustrates also that rules rooted by a common function symbol are compiled into the normalising term construction function for this symbol. According to this schema, and given a set of rules, mtom will decompose the many-to-one matching into several independent functions that are invoked from the main generated function `mtom_main`. Note that this example does not provide real "Many-To-One" matching, since rewrite rules are simply processed one after the other.

As we have noticed before, mtom does not construct terms, however it generates term construction functions for symbols occurring at the root of a rule. In such a function if the constructed term is reducible, then the reduction is provided and the resulting term will be constructed by a user's semantic action (as the right hand side of the corresponding rewrite rule). A problem arises if for such a symbol f, no rule is applicable. In this case the term construction function for symbol f is supposed to construct the term $f(t_1, \ldots, t_n)$ itself. As an example, let us consider a rewrite system which only contains the rule $Plus(Suc(x), y) \rightarrow Suc(Plus(x, y))$. The evaluation of `plus(zero(),zero())` should result in the term $Plus(Zero, Zero)$ as no rule is applicable in this case. In general, such a situation indicates a wrong design (at least in context of innermost evaluation). In order to avoid those cases, the user is invited to define a default rewrite rule $f(x_1, \ldots, x_n) \rightarrow \ldots$ whose semantic action is supposed to construct the appropriate term $f(x_1, \ldots, x_n)$ or to issue an error message. From an implementation point of view, if no such rule is specified, mtom will return a special pre-defined term (such as `NULL` or `null` for C or Java).

The generation of normalising term constructing functions does not restrict the user to an innermost normalisation strategy: there is no obligation for the user to construct terms with this set of functions generated by mtom. When implementing other evaluation strategy, the user can define its own term construction functions. The reduction mechanism is therefore kept fully under the control of the user.

[1] This function matches the first argument against the pattern string given as second argument, and modifies accordingly the values of variables occurring after the second argument.

5 mtom Specification Language

In this section, we introduce the concrete syntax of an input file processed by mtom. It is inspired by the syntax of the Yacc tool (namely, usage of the % sign). In order to support the intuition throughout this section, we show how mtom can be used to specify addition (*Plus*) on Peano integers (*Suc* and *Zero*), and we use the C programming language as the goal language.

5.1 Passing Text to the Goal Language

Similarly to Yacc, any text enclosed between %{ and %} operators is directly passed to the generated output file. It is supposed that the enclosed text contains headers, declarations and definitions of functions and data-types used in semantic actions. As an example, consider an input file which starts with the following text:

```
%{
#include "myheaders.h"
struct term {            // an example of a term representation
  int fsym;              // functional symbol code
  int arity;             // number of subterms
  struct term **subs; }; // pointer to array of subterms.
%}
```

This means that the generated file starts with the corresponding `include` directive and contains the declaration of `term`.

5.2 Specifying Term API

As mentioned before, mtom does not require any pre-defined term representation. Instead, it performs an access to terms through a user-defined API. Hence, mtom needs to know:

- how to get the top symbol of a term
- how to get the n-th subterm of a given term
- how to compare two terms (wrt. equality)

In order to be flexible, the user is not restricted to have only one uniform term representation (see Section 6.2 for more details): the API can be parametrized by the type name of the class of terms to be considered. Those types are enclosed between < and > signs.
Continuing our example, in mtom we can specify a term API:

```
%GET_FUN_SYM<struct term *>(t) (t->fsym)
%GET_SUBTERM<struct term *>(t,n) (t->subs[n])
```

In all cases, declarations behave as macros, it means that the body is inserted into appropriate places of the output file (with corresponding argument replacements). In particular this example states that for terms represented

by the "`struct term *`" type, top symbol of a term `t` can be obtained by the `(t->fsym)` text and `n`-th subterm of a term `t` can be obtained by the `(t->subs[n])` text.

Since we want to consider non-linear patterns, a term comparing function must be provided as follows:

```
%TERMS_EQUAL<struct term *>(t1,t2) (compareTerms(t1, t2))
```

5.3 Specifying Signature of Patterns

In order to parse patterns, one has first to define the signature on which patterns are expressed. mtom's signature must also establish a relation between function symbols occurring in patterns (parsed by mtom) and internal representation of function symbols in the goal language. For example, if the signature declares a symbol *suc*, it must also specify its code (the integer `1`): the number used to represent the *suc* symbol in the `fsym` field of the `term` structure.
Consider the following signature declaration:

```
%sym struct term *zero()                                % 0
%sym struct term *suc(struct term *)                    % 1
%sym struct term *plus(struct term *, struct term *) % 2
```

This piece of code defines a signature of three symbols: `zero`, `suc` and `plus`. All these symbols are defined on our simple term data-structure, and they are represented by integer constants 0, 1 and 2.

A signature item is introduced by the `%sym` keyword followed by a symbol declaration which is split into two parts. The part before the `%` sign defines the profile of the symbol. The concrete syntax is the same as in the corresponding goal language. This part is also used as declaration for the normalising term construction function (which will be generated if there is a pattern with this symbol at the top position). The part after `%` defines the code of the symbol to be used in the generated code. This relates the function symbol to its internal representation in user's data-structures. This second part may be omitted if the symbol only appears at the top position of patterns. This usually occurs when dealing with built-in data-types (see Section 6.1).

5.4 Specifying Variables

Patterns can contain variables. Those variables can be used inside corresponding semantic actions too. Variables are introduced by the `%var` keyword followed by declarations. Similarly to symbol definitions the concrete syntax comes from the goal language.
For example, the text:

```
%var struct term *x,*y,*z;
```

declares three variables `x`, `y` and `z`, each of type `struct term *`.

5.5 Specifying Rewrite Rules

Patterns and rewrite rules are introduced by the `%rule` keyword followed by a pattern, the `%-->` sign and by a semantic action. The semantic action will be executed if its corresponding pattern matches the input term. In this case, local variables (generated by mtom) record the matching substitution. If a `return` statement occurs in a semantic action it is supposed to return the right hand side of the rule.
Consider the following example:

```
%rule plus(zero, x)     %--> return(x);
%rule plus(suc(x),y)    %--> return(SUC(plus(x,y)));
%rule plus(x,y)         %--> printf("unexpected term\n"); exit(1);
```

It describes three rewrite rules implementing the *plus* operator on Peano arithmetics (the third rule should not be executed, but it ensures that at least one rule with *plus* at the top position of the pattern will always be applicable).

Note that the second semantic action calls recursively the normalising term constructing function `plus` (generated by mtom) and then a `SUC` function, which should be an external function constructing a term by adding one *suc* operator. This function should be user-defined, for example as follows:

```
%{
struct term *SUC(struct term *x) {
  struct term *res;
  res = (struct term *) malloc(sizeof(struct term));
  res->fsym = 1; res->arity = 1;
  res->subs = (struct term **) malloc(sizeof(struct term *));
  res->subs[0] = x;
  return(res); }
%}
```

In order to easily implement several variants of rewriting (conditional rewriting, rewriting with local assignments, strategy controlled rewriting, etc.), possible ambiguities are solved as follows: If there are several patterns applicable on the given `input_term` then the first applicable rule is taken, i.e. the semantic action corresponding to the **first** applicable pattern is executed. If a semantic action does not exit by the `return` statement, then the next possible match of this rule is taken and the semantic action is executed again for the new substitution (this applies only on possible extensions of matching, see Section 6.4). If there is no more different matches for the same pattern then the next applicable pattern (in order of appearance) is taken and its semantic action is executed. Those rules permit us in particular to implement conditional rewrite systems, where the construction of each right hand side will be enclosed in an `if` statement (this `if` statement comes from the goal language, it is not an mtom construction, of course). Let us take for example two rules (note that this example involves built-in data-types of the goal language, see Section 6.1):

```
%rule fib(x)   %--> if(x<=1) return(1);
%rule fib(x)   %--> return(fib(x-1)+fib(x-2));
```

Here both rules are always applicable on a term starting by the fib symbol. Due to the condition, the second rule is executed if and only if the variable x is instantiated on a subterm greater than 1.

6 mtom Features

6.1 Goal Language Built-ins

We expect that primitive data-types, such as integers, floating points and strings will be often used in mtom programs. A syntactic sugar is introduced to ease this usage. First, the API for those types is not required. Second, constant symbols of built-in data-types do not need to be declared. They can be directly written in patterns and they stand for the corresponding constant term of the given built-in type. Those constants, however, cannot appear at the top position of patterns, since they must be constructor symbols.
Let us take the following example:

```
%sym int fib(int)
%var int x;

%rule fib(0) %--> return(1);
%rule fib(1) %--> return(1);
%rule fib(x) %--> return(fib(x-2)+fib(x-1));
```

Here, the generated function fib tests if its argument is equal to 0 or 1, and applies the third rewrite rule if the argument is different.

6.2 Mixed Terms

In mtom, symbol definitions contain profiles of symbols, including types of each argument and of result. Those types are not necessarily identical. In consequence, terms with different representations of different subterms are allowed.
Let us take for example a symbol definition:

```
%sym struct term *cons(int, struct term *)  % 1
```

mtom parses the signature definition and knows the type of each subterm. It knows also that when decomposing a term with *cons* as top symbol, its first subterm will be of type `int` and the second of type `struct term*`. In consequence, the generated temporary variables will be declared of appropriate types. On the other hand, mtom expects that patterns are well-typed.

In general, this feature gives to mtom the possibility to work on several different term representations, namely to define functions transforming one data representation into another, to define functions working with built-in

values as if they were terms, etc. This is a very important feature for industrial use.
Let us take for example the following code defining a sorted list of integers, with no duplicate elements:

```
%{
struct sortedIntList {
  int value;
  struct sortedIntList *next; };
#define NIL  0
#define CONS 1
%}

%GET_FUN_SYM<struct sortedIntList *>(list) ((list==NULL)?NIL:CONS)
%GET_SUBTERM<struct sortedIntList *>(list,n)
          ((n==0)?list->value:list->next)
%sym struct sortedIntList *nil()                                  % NIL
%sym struct sortedIntList *cons(int, struct sortedIntList *)  % CONS
%var int x,y;
%var struct sortedIntList *z;

%rule cons(x, cons(y, z)) %--> if(x==y) return(cons(y,z));
%rule cons(x, cons(y, z)) %--> if(x>y)  return(cons(y,cons(x,z)));
%rule cons(x, z)          %-->          return(allocCons(x,z));
```

In this example, we suppose that the function `allocCons` allocates and fills a new `struct sortedIntList` item. Provided that any list is constructed only with the `cons` function, this makes that those integer lists in the program will always be sorted. The user can handle this data-structure as if it was written directly in C. Note that the expression `((n==0)?list->value:list->next)` may be wrong typed on some C implementations. In such a case, some casts should be added to the above example. On the other hand, mtom is supposed to generate "reverse" casts before assignments to its temporary variables.

6.3 Support for Non-Automatic Garbage Collectors

Until now, we did not take into account the question of memory management. This is because the user has to take care of this problem. However, unless an automatic garbage collector is used (like for instance with ATerms [27]), some kind of cooperation is desired. This concerns mainly accessing parts of a term matched by a pattern. Some parts should be freed after an application of rewrite rule. Semantic actions need to be able to access particular subterms of the term to be reduced. In mtom, a special mechanism is provided to handle this situation. By convention any subterm of the input term corresponds to a local variable denoted by its position in the pattern. For example, the local variable `_1` denotes the first subterm, whereas the variable `_2_1` denotes the first subterm of the second subterm. Note that when a particular semantic

action is executed, all positions were inspected, and so, those variables are instantiated.
For example, when using a reference counting garbage collector, a rule for left-associativity of *plus* can be:

```
%rule plus(x,plus(y,z)) %--> free(_2); return(plus(plus(x,y),z));
```

Here the function `free` is supposed to decrease the reference counter and, if the latter is equal to zero, to free the packet $plus(y, z)$ of the reduced term.

This convention gives users direct access to all subterms of the input term. Those subterms can also be used to improve the construction of the right hand side (for instance, if there are common subterms between left hand side and right hand side of the rule).

6.4 Pattern Matching

Non-Linear Patterns
A variable may have multiple occurrences in a pattern. Such non-linear patterns cause that mtom will need to compare two subterms of the input term. Because mtom is not supposed to know the entire term representation, the user needs to specify how to compare two terms in its API part.

Another problem concerning non-linear terms is when to check the equality of corresponding subterms. There are basically two possibilities to implement those checks.

First, we can automatically linearise patterns by indexing differently multiple occurrences of a variable, say x_1, x_2, x_3 for three occurrences of x. Then, matching is processed on linear patterns, and eventually we must test whether variables introduced during the previous linearisation step are or not equal to the same term.

Second, we could forbid in mtom the use of non-linear patterns. Hence, the linearisation is left to the user, who can postpone the equality test in the semantic action of the rule:

```
%rule f(x,y,b)  %--> if(equals(x,y)) {...}
```

In the current implementation, mtom allows non-linear patterns and performs the equality test during the matching process whenever a new occurrence of a variable is instantiated. But the user is free to consider a linear pattern together with an equality test occurring in the semantic action.

Equational Matching
Rule-based languages (ASF+SDF, Maude, Elan) support extensions of matching modulo some equational theory, like A (Associativity), AC (Associativity-Commutativity), and some variants like AC1 (AC plus the Identity). As we want to use mtom as an intermediate code for ELAN compiler, we would like to support in the future those extensions. Making decision whether mtom should support equational matching or not is not easy. In this section, we discuss

about those issues, even if the full treatment of the problem is beyond the scope of this paper.

Indeed, dealing with equational matching is much more complicated than syntactic matching. First, this would break the nice property that mtom does not construct terms, because for example in AC-matching we need to construct new terms in order to build the solutions of a matching problem. For example when matching the pattern $(a + x)$ against the term $((a + b) + c)$ (where $+$ is associative and commutative), then x has to be instantiated by the $(b + c)$ which did not exist before. It must be created during the matching process. Second, specific non-syntactic matching algorithms are usually working with their own internal term representations. In order to be efficient, this would oblige the users to adopt this term representation and we would loose another nice mtom property, which is to be independent on a particular term representation.

However, one can remark that, even if there is no specific support for equational matching, it is possible to implement equational rewriting using mtom. Since a user can write as semantic action anything he wants, he can also split pattern matching into two stages. The first will be the syntactic top part of patterns, matching of this part will be executed by mtom. The rest of the matching process, i.e. the full A or AC matching can be then implemented directly in the goal language inside semantic actions.

For those reasons, we prefer to see the integration of equational matching as an open problem, at least for early versions of mtom tool.

Many-To-One Matching

When starting our work on mtom we thought that a particularly good choice of the many-to-one matching algorithm would be essential. Now it seems that the tool design has its importance in its own. It offers a specification language for industrial use of rewriting. When a user practices with mtom, its application will not depend on a concrete implementation of the many-to-one matching algorithm. The user is free to switch among different mtom implementations with different pattern matching algorithms.

For the moment, the concrete choice of the many-to-one algorithm does not have so much importance. However, we feel that such an algorithm should have reasonable time and space complexities. In the future, we plan to make experiments with sophisticated pattern matching algorithms like the one introduced in [26].

The implementation will be much more complex when considering equational matching (see previous paragraph). The compilation of Many-To-One AC-matching is already investigated in [16].

6.5 Rewriting Strategies

User-Defined Evaluation Strategies

As we have mentioned previously, the user can define its own evaluation strategies when using mtom. If semantic actions are just building right hand sides of rules instead of reducing them, then mtom itself does not provide and does not impose any reduction strategy. It turns into a simple tool for many-to-one matching and the generated function `mtom_main` provides one reduction step at the top position of its input term. Using this function, the user can easily define its own evaluation strategies.

However note that this case is much more complicated. For example, the fact that no rule is applicable does not necessary mean a wrong design. It may mean that an argument is not sufficiently evaluated at the moment. Due to this fact, the default rules should evaluate their arguments before causing an error message. Also conditions have to explicitly call an evaluation function before failing in the application of a rule.

Let us take for example a rewrite system computing the (infinite) list of all prime numbers. This example defines a function `ilist` computing the (infinite) list of all natural numbers; a function `isprime` testing whether a number is prime or not by trying to divide it by previously computed prime numbers; a function `pfilter` getting the infinite list of natural numbers and filtering it by the `isprime` function which results into the list of all prime numbers. The `pfilter` function also keeps the list of already computed prime numbers in its second argument. Semantic actions use functions `Cons`, `Ilist`, `Isprime`, `Pfilter` which are supposed to construct corresponding terms without trying to reduce them. This example acts on Peano arithmetics and uses functions mul, mod and div implementing respectively the multiplication, module and division between two Peano integers. Those functions are supposed to return normal forms of the result (innermost normalisation), so the example demonstrates also a combination of different rewriting strategies in a single program.

```
%rule ilist(n)                %--> return(Cons(n, Ilist(suc(n))));

%rule isprime(i,nil)          %--> return(1);
%rule isprime(i,cons(j,x))    %--> if(less(i,mul(j,j))) return(1);
%rule isprime(i,cons(j,x))    %--> if((mod(i,j)==zero) return(0);
%rule isprime(i,cons(j,x))    %--> return(Isprime(i,x));
%rule pfilter(cons(i,x),p)    %--> if(lazynf(Isprime(i,p))) {
                    return(
                           Cons(i,Pfilter(x,append(p,Cons(i,nil))))
                           );
                                  } else {
                                    return(Pfilter(x,p));
                                  }
```

```
%rule primes(n)          %--> return(
                              Pfilter(Ilist(suc(suc(zero))), nil));
```

In this example we have to define two C functions controlling applications of rewrite rules: the function `lazystep` providing a single application of a rule and the function `lazynf` computing a normal form of a given term. Both functions act in a lazy manner meaning that they reduce an outermost position first. Those functions use global variable `mtom_reduction_occured`. This variable is cleared by the `mtom_main` function in case that it did not provide any reduction.

```
struct term *lazystep(struct term *t) {
  int i;
  mtom_reduction_occured = 1;
  t = mtom_main(t);
  for(i=0; mtom_reduction_occured==0 && i<t->arity; i++) {
    t->subs[i] = lazystep(t->subs[i]);
  }
  return(t);
}

struct term *lazynf(struct term *t) {
  do {
    while (t->fsym == CONS) {
      printf("%t .", lazynf(t->subs[0]));
      t = t->subs[1];
    }
    t=lazystep(t);
  } while (mtom_reduction_occured);
  return(t);
}
```

Non-Deterministic Computations

In some languages (ELAN for example), rewriting is performed in a non-deterministic way, meaning that several possible rewrite steps are explored successively by backtracking. Non-determinism occurs when several rewrite rules can be applied on a single term. In such a case, all the possibilities should be explored.

In ELAN, we used a backtracking library [20] containing two functions "setChoicePoint()" and "fail()". The behaviour of those two functions is similar to a pair of standard C functions "setjmp" and "longjmp" with no restriction on "longjmp" invocations. Moreover "fail" is not jumping to any of the "setChoicePoint()" calls, but always to the last one.

When using this pair of functions in semantic actions, one can implement backtracking by using the fact that, if a semantic action does not exit via the

`return` instruction, then another semantic action is tried. In the simple case where each semantic action returns just one result, it is sufficient that the whole semantic action is enclosed into

```
if(setChoicePoint()) { ... }
```

instruction and the default rewrite rules (applied when no regular pattern matches) execute the `fail()` function. This ensures that a choice point is set and then the right hand side is constructed and returned (the *true* branch of the `if` statement). In the case that a `fail` occurs somewhere in further computation, the control comes back to the last `if(setChoicePoint())` statement and pass throw the *false* branch at the end of the semantic action. This will cause that next match and the next applicable pattern will be examined.

In summary this means that backtracking can be implemented without any modification of the mtom tool. One has just to modify semantic actions, which are fully under the control of users.

7 Related Works

Pattern matching is a simple but expressive concept which is used in a wide variety of programming languages and in a variety of programming styles. One axis for classifying the expressiveness of pattern matching is how many bindings a pattern match yields:

- In the *single-match* style, a successful match yields just one binding. This style is usually taken in declarative programming languages. In particular, most functional programming languages [4] such as CAML [8,31], Clean [6,24], Erlang [2,1], Gofer [14], Haskell [15,23], or ML [9,18] allow defining functions by pattern matching.
- In the *all-match* style, a pattern match yields a set of bindings corresponding to all possible matches. This style is often used in query languages and in programming languages based on term rewriting. Let us cite for instance OBJ [13], ASF+SDF [17], Maude [7], Cafe-OBJ [12] and ELAN [5]. In these rule-based languages, given a term and a single rule, there may exist several ways to apply the rule when the matching is performed in an equational theory for example. Given a term and a set of rules, several rules can be potentially applied (corresponding to all possible matches). Selecting which rules have to be applied is usually done *a posteriori*, according to the conditions and the strategy under which the rewrite rules are applied.

In general, pattern matching constructions are well integrated into mentioned languages. But this is not the case in imperative languages such as C, C++ or Java, in which are written most of industrial applications. In order to add pattern matching facilities to imperative languages such as C, C++ or Java, one can imagine using sub-routines written in one of the previously mentioned language. The main drawback of this approach concerns data rep-

resentation: pattern matching usually works on internal data-structures which are language dependent. Thus, when integrating it with other yet existing code many data conversion are needed and the memory cannot be managed in a uniform way.

Several systems have been developed in order to integrate pattern matching and transformation facilities into imperative languages. Let us mention for instance Rigal [3,11], R++ [10,25], App [22] and Prop [19]. Each of these systems has its own specificity. Rigal is presented as a compiler construction language based on advanced pattern matching. R++ and App are preprocessors for C++: the first one adds production rule constructs to C++, whereas the second one extends C++ with a match construct. Finally, Prop is a multi-paradigm extension of C++, including pattern matching constructs. In all these existing systems, the pattern matching process can only act on internal data-structures. Consequently, these systems cannot be used to add pattern matching facilities in an existing project written for example in C, C++ or Java: it is not easy to convince an industrial to use rule-based programming style if the first thing to do consists in translating the existing main data-structures.

8 Conclusion

In this paper we have presented a tool for many-to-one matching which seems to be particularly well-suited to provide rewriting in an industrial context. In our opinion, mtom is a key component for the implementation of rule-based language compilers, as well as for the design of program transformation tools, provided that programs are represented by terms (using for instance `ATerms` or `XML` representations). This tool is very general, depending on semantic actions which can be used to provide variety of rewriting strategies including the most usual leftmost-innermost and leftmost-outermost normalisations. mtom does not depend on a particular implementation of terms, as it gives no term manipulations, it does not construct or destruct terms, it only "observes" terms through a user definable API, all term modifications are made by users in semantic actions. This gives the user a large freedom in the choice of a term representation to consider. mtom even does not impose a homogeneous term representation, different data-structures can be mixed inside terms (Section 6.2). Conditional rewrite rules can be implemented directly as well as several extensions of term rewrite rules.

The examples presented in the paper are all operational and have been tested with the mtom tool. This first prototype is a monolithic piece of software which is rather hard to maintain. For instance, the mtom syntax is "hard-wired" in the implementation of the parser. We are currently working on a new design and implementation that improves the first prototype in the following directions:

- The new mtom parser is a separate piece of software that generates an

abstract syntax tree, represented by an ATerm [27].

- The compilation scheme consists of two phases: (a) The abstract syntax tree is transformed (using mtom itself) into another tree representation where each group of `rule` constructs is replaced by a matching automaton. (b) The latter is pretty-printed according to the chosen goal language. If one wants to enrich the generator with another goal language, it is sufficient to parameterize or to extend this phase according to the goal language constructs (How to declare a variable? How to assign a variable? How to perform an `if-then-else`? etc.)
- A new `match` primitive is added. Contrary to the existing `rule` construct, the `match` construct can be used everywhere a goal language instruction is allowed (in blocks or functions for example). Furthermore, this construct is more powerful since it can be used to group any set of patterns (even those with different top symbols). This feature is particularly useful when compiling strategy controlled rewriting [5,29] (i.e. a set of rules applied under a given strategy). It is clear that the current `rule` construct is implemented using this more atomic construct.

The future developments will be provided under the terms of the Gnu General Public License, and will be available on the ELAN homepage [21].

Acknowledgements: We would like to thank the referees for their pertinent remarks, Claude Kirchner and Eelco Visser for fruitful discussions on the mtom design and future extensions.

References

[1] Armstrong, J., *The development of Erlang*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 1997, pp. 196–203.

[2] Armstrong, J., R. Virding, C. Wikström and W. M., "Concurrent Programming in Erlang," Prentice-Hall International, 1996.

[3] Auguston, M., *Programming language RIGAL as a compiler writing tool*, ACM SIGPLAN Notices **25** (1990), pp. 61–69.

[4] Bird, R. and P. Wadler, "Introduction to Functional Programming," Prentice-Hall International Series in Computer Science, Prentice-Hall International, 1988, japanese translation, 1991. Dutch translation, 1991. German translation, 1992.

[5] Borovanský, P., C. Kirchner, H. Kirchner, P.-E. Moreau and C. Ringeissen, *An overview of* ELAN, in: C. Kirchner and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, Electronic Notes in Theoretical Computer Science, 1998.

[6] Brus, T. H., M. C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer and H. P. Barendregt, *CLEAN - A language for functional graph rewriting*, in: Kahn, editor, *In Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, number 274 in Lecture Notes in Computer Science (1987), pp. 364–384.

[7] Clavel, M., S. Eker, P. Lincoln and J. Meseguer, *Principles of Maude*, , **4** (1996).

[8] Cousineau, G. and M. Mauny, "The Functional Approach to Programming," Cambridge University Press, 1998.

[9] Cousineau, G., L. C. Paulson, G. Huet, R. Milner, M. Gordon and C. Wadsworth, "The ML Handbook," INRIA, Rocquencourt, 1985.

[10] Crawford, J. M., D. Dvorak, D. Litman, A. Mishra and P. F. Patel-Schneider, *Path-based rules in object-oriented programming*, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference* (1996), pp. 490–497.

[11] Engelson, V. and M. Auguston, *Rigal homepage:* `http://www.ida.liu.se/~vaden/rigal`.

[12] Futatsugi, K. and A. Nakagawa, *An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks*, in: *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[13] Goguen, J. A. and T. Winkler, *Introducing OBJ3*, Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025 (1988).

[14] Jones, M. P., *The implementation of the Gofer functional programming system*, Technical Report Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA (1994).

[15] Jones, M. P., *Hugs 1.3, The Haskell User's Gofer System: User Manual*, Technical Report Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, England (1996).

[16] Kirchner, H. and P.-E. Moreau, *Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories*, Journal of Functional Programming (2000), to appear.

[17] Klint, P., *A meta-environment for generating programming environments*, ACM Transactions on Software Engineering and Methodology **2** (1993), pp. 176–201.

[18] Leroy, X. and M. Mauny, *Dynamics in ML*, Journal of Functional Programming **3** (1993), pp. 431–463.

[19] Leung, A., *Prop homepage:* `http://cs1.cs.nyu.edu/phd_students/leunga/prop.html`.

[20] Moreau, P.-E., *A choice-point library for backtrack programming*, JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic (1998).

[21] Moreau, P.-E., C. Ringeissen and M. Vittek, *mtom homepage:* `http://www.loria.fr/ELAN/mtom`.

[22] Nelan, G., *App homepage:* `http://www.primenet.com/~georgen/app.html`.

[23] Peyton Jones, S., *Compiling Haskell by program transformation: a report from the trenches*, in: *Proceedings of the European Symposium on Programming (ESOP'96)*, Lecture Notes in Computer Science **1058** (1996).

[24] Plasmeijer, M. J. and M. C. J. D. van Eekelen, "Functional Programming and Parallel Graph Rewriting," Addison-Wesley, 1993.

[25] R++, *homepage:* `http://www.research.att.com/sw/tools/r++`.

[26] Sekar, R. C., R. Ramesh and I. V. Ramakrishnan, *Adaptive pattern maching*, in: W. Kuich, editor, *Proceedings of ICALP 92*, Lecture Notes in Computer Science **623** (1992), pp. 247–260.

[27] Van den Brand, M., H. de Jong, P. Klint and P. Olivier, *Efficient annotated terms*, Software-Practice and Experience **30** (2000), pp. 259–291.

[28] van den Brand, M. G. J., P. Klint and P. Olivier, *Compilation and Memory Management for ASF+SDF*, in: *Compiler Construction*, Lecture Notes in Computer Science **1575** (1999), pp. 198–213.

[29] Visser, E. and Z.-e.-A. Benaissa, *A core language for rewriting*, in: C. Kirchner and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, Electronic Notes in Theoretical Computer Science, 1998.

[30] Vittek, M., *A compiler for nondeterministic term rewriting systems*, in: H. Ganzinger, editor, *Proceedings of RTA'96*, Lecture Notes in Computer Science **1103** (1996), pp. 154–168.

[31] Weis, P. and X. Leroy, "Le langage Caml," InterEditions, 1993.