# A Pattern Matching Compiler for Multiple Target Languages

Pierre-Etienne Moreau[1], Christophe Ringeissen[1], and Marian Vittek[2]

[1] LORIA-INRIA, 615, rue du Jardin Botanique,
BP 101, 54602 Villers-lès-Nancy Cedex France
{moreau,ringeiss}@loria.fr, url: elan.loria.fr/Toolkit

[2] Institut of Informatica Mlynska dolina,
842 15 Bratislava, Slovakia
vittek@fmph.uniba.sk

**Abstract.** Many processes can be seen as transformations of tree-like data structures. In compiler construction, for example, we continuously manipulate trees and perform tree transformations: parse trees, abstract syntax trees, tree transformations, etc. This paper introduces a pattern matching compiler (TOM): a set of primitives which add pattern matching facilities to imperative languages such as C, Java, or Eiffel. We show that this tool is extremely non-intrusive, lightweight and useful to implement tree transformations. It is also flexible enough to allow the reuse of existing data structures.

## 1 Introduction

For the compiler construction, there is an obvious need for programming transformation of structured documents like trees or terms: parse trees, abstract syntax trees (AST for short). In this paper, our aim is to present a tool which is particularly well-suited for programming various transformations on trees/terms. In the paper we will often talk about "term" instead of "tree" due to the one-to-one correspondence between these two notions. Our tool results from our experience on using existing programming languages and programming paradigms to implement transformations of terms.

In declarative (logic/functional) programming languages, we may find some built-in support to manipulate structured expressions or terms. For instance, in functional programming, a transformation can be conveniently implemented as a function declared by pattern matching [14, 26, 8, 3], where a set of patterns represents the different forms of terms we are interested in. A pattern may contain variables (or holes) to schematize arbitrary terms. Given a term to transform, the execution mechanism consists in finding a pattern that *matches* the term. When a match is found, variables are initialized and the code related to the pattern is executed. Thanks to the mechanism of pattern matching, one can implement a transformation in a declarative way, thus reducing the risk to implement it in the wrong way.

For efficiency reasons, it may be interesting to implement similar tree-like transformations using (low-level) imperative programming languages for which efficient compilers exist. Unfortunately, in such languages, there are no built-in facilities to manipulate term structures and to perform pattern matching. One possibility would be to enrich an existing imperative programming language with pattern matching facilities [17, 27, 16, 19]. This hard-wired approach makes the users, data-structures and implementation dependent on this new language. A simpler solution would be to develop a special library implementing pattern matching functionality. This approach

is followed for example in the ASF+SDF group [23] where a C library called ATERMS [22] has been developed. In this library, pattern matching is implemented via a function called `ATmatch`, which consists of matching a term against a single pattern represented by a string or a term.

Therefore, it is possible to define a transformation by pattern matching, thanks to a sequence of `if-then-else` instructions, where each condition is a call to the `ATmatch` function. But this approach has three drawbacks. First, matching is performed sequentially: patterns are tried one by one. This can be rather inefficient for a large number of patterns. Second, terms and patterns are untyped, and thus may be error prone. Third, the programming language and the data structure are imposed by the library, and so the programmer cannot use his favorite language as well as his own data structure to represent terms.

For sake of efficiency, we are interested in the *compilation* of pattern matching. By compilation we mean an approach where all patterns are compiled together producing a *matching automaton*. This automaton then performs matching against all patterns simultaneously. Our research on this topic is guided by the following concerns:

- How to efficiently compile different forms of pattern matching? We are concerned by simple syntactic matching but also by matching modulo an equational theory. In such case, for example a pattern `x+3` can match expression `3+7` thanks to commutativity of plus.
- How to implement compilation of pattern matching in a uniform way for a large class of imperative programming languages and for any representation of terms?

To tackle the above mentioned problems, we develop a non-intrusive pattern matching compiler called TOM. Its design follows our experiences on the efficient compilation of rule-based systems [25, 10]. Our tool can be viewed as a Yacc-like compiler translating patterns into executable pattern matching automata. Similar to Yacc, when a match is found, the corresponding "semantic action" (a sequence of instructions written in an imperative language) is triggered and executed. In a way, we can say that TOM translates a *declarative-imperative* function – defined by pattern matching and imperative instructions – into a *fully imperative* function. The resulting function can be integrated to an application written in a classical imperative language such as C, Java, or Eiffel, called the *target language* in the rest of the document. In this paper, we illustrate the different advantages of the approach implemented by TOM, namely:

**Efficiency** The gain of efficiency follows from the compilation of matching as implemented in TOM.

**Flexibility** When trying to integrate a black-box tool in an existing system, one of the main bottlenecks comes from data conversion and the flexibility offered to the user. One of the main originalities of our system is its independence on term representation. The programmer can use (or re-use) his own data structures for terms/trees and then execute matching upon those data structures. We propose to access terms using only a simple *Application Programming Interface* (API) defined by the user. This makes multiple term representations possible and allows to work on user defined data structures as well as on existing built-in data types.

**Generality** TOM is able to consider multiple target languages (C, Java, and Eiffel). TOM is implemented in TOM itself as successive transformations of AST. The code generation is performed at the very end, depending on the target language we are interested in. Hence, the target language is really a parameter of TOM.

**Expressivity** TOM supports non-linear patterns and equational matching like modern rule-based programming languages. Currently, we have implemented pattern matching with list operators. This form of associative matching with neutral element is very useful for practical applications. The main difference with standard (syntactic) matching is that variables have now multiple possible assignments.

The paper is organized as follows: Section 2 motivates the main features of TOM on a very simple example. In Section 3, we present the main language constructs and their precise meanings. The different envisioned applications are described in Section 4. Since TOM is non-intrusive, it can be used in the context of existing applications to implement in a declarative way some functionalities which can be naturally expressed as transformations of terms (Section 4.1). Furthermore, we show the interest of TOM in designing a compiler via some transformations of AST performed by pattern matching: this is exemplified with the current implementation of TOM itself (Section 4.2). Then, we briefly present how TOM can be used for the compilation of rule-based languages (Section 4.3), where the main difficulty is the compilation of pattern matching. Section 5 presents some related work and Section 6 concludes with final remarks and future work.

## 2 What is TOM?

In this section, we outline the main characteristics of TOM and we illustrate its usage on a very simple example specifying a well-known algebraic data type, namely the Naturals.

TOM does not really define a new language: it is rather a language extension which adds new matching primitives to an existing imperative language. From an implementation point of view, it is a compiler which accepts different *native languages*: C, Java, and Eiffel. The compilation process consists of translating introduced matching constructs into the underlying native language. Since the native language and the *target language* are identical and only introduced constructs are expanded, the presented tool can also be seen as a kind of preprocessor. On the other hand, the support of multiple target languages, and the fact that the input program has to be completely parsed before the transformation process can begin, make us consider TOM as a compiler.

To make the presentation simpler, we assume that TOM only adds one new construct: `%match`. This construct is similar to the `match` primitive of ML family languages: given a term (called subject) and a list of pairs: pattern-action, the `match` primitive selects a pattern that matches the subject and performs the associated action. This construct can seen as an extension of the classical `switch/case` construct. The main difference is that the discrimination occurs on a *term* and not on atomic values like characters or integers: the patterns are used to discriminate and retrieve information from an algebraic data structure.

To give a better intuition of what is TOM, let us consider a simple symbolic computation (the addition) defined on Peano integers represented by *zero* and *successor*. When using Java as native language, the sum of two integers can be described in the following way:

```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,zero   -> { return x; }
    x,suc(y) -> { return suc(plus(x,y)); }
  }
}
```

This example should be read as follows: given two terms $t_1$ and $t_2$ (that represent Peano integers), the evaluation of `plus` returns the sum of $t_1$ and $t_2$. This is implemented by pattern matching: $t_1$ is matched by $x$, $t_2$ is possibly matched by the two patterns *zero* and $suc(y)$. When *zero* matches $t_2$, the result of the addition is $x$ (with $x = t_1$, instantiated by matching). When $suc(y)$ matches $t_2$, this means that $t_2$ is rooted by a *suc* symbol: the subterm $y$ is added to $x$ and the successor of this number is returned. The definition of `plus` is given in a functional programming style, but

the `plus` function can be used in Java to perform computations. This first example illustrates how the `%match` construct can be used in conjunction with the considered native language.

In order to understand the choices we made when designing TOM, it is important to consider TOM as a *restricted* compiler: it is not necessary to parse the native language in detail in order to be able to replace the `%match` constructs by a sequence of native language instructions (Java in this example). This could be considered as a kind of island parsing, only the TOM constructs are parsed in detail. The first phase of the transformation process consists in *reading* the program: during this phase, the text is read and TOM constructs are recognized, whereas remaining parts are considered as target language constructs. The output of this first phase is a tree which contains two kinds of nodes: *target language nodes* and *TOM construct nodes*. When applied to the previous example, we get the following program with a unique TOM node, represented by a box as follows:

```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,zero   ->  { return x; }
    x,suc(y) ->  { return suc(plus(x,y)); }
  }
}
```

TOM never use any semantic information of the *target language nodes* during the compilation process, it does not inspect nor modify the source language part. It only replaces the TOM constructs by instructions of the native language. To support the intuition, the reader could easily imagine that the previous `%match` construct will be replaced by two nested `if-then-else` constructs.

At this point, it is interesting to note that `plus` is a function which takes two `Term` data structures as arguments, whereas the matching construct is defined on the algebraic data type `Nat`. This remark introduces the second generic aspect of TOM: the matching can be performed on any data structure. For this purpose, the user has to define its term representation and the mapping between the concrete representation and the algebraic data type used in matching constructs.

To make our example complete, we have to define the term representation `Term` and the algebraic data type which defines the sort $Nat$ and three operators:

$$\{zero :\mapsto Nat, \quad suc : Nat \mapsto Nat, \quad plus : Nat \times Nat \mapsto Nat\}$$

For simplicity, we consider in this example that the ATerm library [22] is used for term representation. This library is a concrete implementation of the Annotated Terms data type (ATerms). Among other possibilities, this library defines an interface to create and manipulate term data structures. It offers the possibility to represent function symbols, to get the arity of such a symbol, to get the root symbol of a term, to get a given subterm, *etc.* The main characteristic of this library is to provide a garbage collector [15] and to ensure maximal sharing of terms. Using this library, it becomes easy to give a concrete implementation of function symbols *zero* and *suc* (the second argument of `makeAFun` defines the arity of the operator):

```
AFun f_zero = makeAFun("zero", 0);
AFun f_suc  = makeAFun("suc", 1);
```

The representation of the constant *zero*, for example, is given by `makeAppl(f_zero)`. Similarly, given a Peano integer `t`, its successor can be built by `makeAppl(f_suc,t)`. Up to there, we know how to represent data using the ATerm library, and how defining matching with TOM, but, we do not know how to connect these two notions yet. Given an integer `t` of sort `Term`, we have to

define how to get its root symbol (using `getAFun` for example) and how to know if this symbol corresponds to the algebraic function symbol *suc*, intuitively `getAFun(t)==f_suc`.

This mapping from the algebraic data type to the concrete implementation is done via the introduction of new primitives, `%op` and `%typeterm`, which are described in Section 3.

## 3 The TOM Language: Main Constructs

In the previous section we introduced the `match` construct of TOM via an example. In this section, we give an exhaustive presentation of TOM by explaining all existing constructs and their behavior. As mentioned previously, TOM introduces a new construct (`%match`) which can be used by the programmer to decompose by pattern matching a tree-like data structure (an ATerm for example). TOM also introduces a second family of constructs which is used to define the mapping between the algebraic abstract data type and the concrete implementation. We distinguish four main constructs: `%typeterm` and `%typelist` are used to define respectively standard and associative algebraic sorts, whereas `%op` and `%oplist` are used to define the many-sorted signature of the algebraic constructors.

### 3.1 Sort definition

In TOM, terms, variables, and patterns are many-sorted. This means that algebraic sorts have to be introduced. The `%typeterm` primitive is usually used for this purpose. Furthermore, an important feature of TOM is that it supports equational matching, in list matching, also known as associative matching with neutral element. The `%typelist` primitive is used to define associative data structures.

In addition to this two basic primitives, the mapping from algebraic sorts to concrete sorts (the target language type, such as `Term`) has to be defined. This is done via the introduction of several sub-functions. To support the intuition, let us consider again the Naturals example where the *Nat* algebraic sort is implemented by ATerms. One possible mapping is the following:

```
%typeterm Nat {
  implement           { Term             }
  get_fun_sym(t)      { t.getAFun()      }
  cmp_fun_sym(s1,s2)  { s1 == s2         }
  get_subterm(t, n)   { t.getArgument(n) }
  equals(t1,t2)       { t1.isEqual(t2)  }
}
```

- The `implement` construct describes how the algebraic type is implemented. The target language part (written between braces: '{' and '}') is never parsed, it is only used by the compiler to declare some functions and variables.
  Since in this example we focus our attention on the ATerm library, we implement the algebraic data type using the "`implement { Term }`" construct. But, if we suppose that another data structure is used, "`struct myTerm*`" for example, the "`implement { struct myTerm* }`" construct should be used to define the mapping.
- `get_fun_sym(t)` denotes a function (parameterized by a term variable $t$) that should return the root symbol of the term referenced by $t$.
  As in the C preprocessor (`cpp`), the body of this definition is not parsed, but the formal parameter (`t`) can be used in the body (`t.getAfun()` in our example).

- `cmp_fun_sym(s1,s2)` denotes a predicate (parameterized by two symbol variables $s1$ and $s2$). This predicate should return `true` if the symbols $s1$ and $s2$ are "equal". The `true` value should correspond to the built-in `true` value of the considered target language. (`true` in Java, and something different from `0` in C for example).
  In our example, we can use "`{ s1 == s2 }`" because two `AFun` can be compared with the `==` predicate.
- `get_subterm(t,n)` denotes a function (parameterized by a term variable $t$ and an integer $n$). This function should return the $n$-th subterm of $t$. This function is only called with a value of $n$ between 0 and the arity of the root symbol of $t$ minus 1.
- `equals(t1,t2)` denotes a predicate (parameterized by two term variables $t1$ and $t2$). Similarly to `cmp_fun_sym(s1,s2)`, this predicate should return `true` if terms $t1$ and $t2$ are "equal". This last optional definition is only used to compile non-linear patterns. It is not required when the specification does not contain such patterns.
  When using the ATerm library, it is defined by "`{ t1.isEqual(t1,t2) }`"

As mentioned before, TOM also supports equational matching. This is performed via the definition of a list-sort data type, using the `%typelist` construct. When defining such a sort, several other access functions have to be defined. In order to correctly compile the list-matching algorithm, it has to know how to enumerate the elements that compose a list. For this purpose, three extra access functions are introduced: `get_head`, `get_tail` and `is_empty`:

- `get_head(l)` denotes a function parameterized by a list variable $l$ that should return the first element of the list $l$.
  When using the `TermList` data type, the definition is "`get_head(l) { l.getHead() }`".
- `get_tail(l)` denotes a function parameterized by a list variable $l$ that should return the tail of the list $l$. In our case, we use "`get_tail(l) { l.getTail() }`".
- `is_empty(l)` denotes a predicate parameterized by a list variable $l$. This predicate should return `true` if the list $l$ contains no element. One more time, the mapping to ATerms is straightforward: "`is_empty(l) { l.isEmtpy() }`".

To clarify the presentation we mainly used the ATerms data structure, but it should be clear that any other data structure could be used. TOM is a multi target language compiler that supports C, Java, and Eiffel.

### 3.2 Constructor definition

In TOM, the definition of a new operator is done via the `%op` construct. The many-sorted signature of the operator is given in a prefix notation. Let us consider the $suc : Nat \mapsto Nat$ operator for instance, its definition is: "`%op Nat suc(Nat)`". Because TOM has no knowledge of the concrete implementation, one more time, the user has to describe how to represent the newly introduced operator:

- `fsym` defines the concrete representation of the constructor. The expression that parameterizes `fsym` should correspond to the expression returned by the function `get_fun_sym` applied to a term rooted by the considered symbol.

In our setting the definition of *suc* and *zero* can be done as follows:

```
%op Nat zero {
    fsym  { f_zero }
}
%op Nat suc(Nat) {
    fsym { f_suc  }
}
```

When all needed operators are defined, it becomes possible to use them to define terms and patterns. Terms are written using standard prefix notation. By convention, all used constants which are undefined are considered as variables. Thus, the pattern `suc(y)` correspond to the term $suc(y)$ where $y$ is a variable. The pattern `suc(zero)` corresponds to the Peano integer *one*.

In addition to the `%op` construct, TOM provides the `%oplist` construct to define list operators. When using such a construct, the user has to specify how the symbol is implemented and how a list can be built. This is done via the two following functions:

- `make_empty()` should return an empty list. This objects correspond to the neutral element of the considered data structure.
- `make_add(l,e)` should return a new list $l'$ where the element $e$ is inserted at the head of the list $l$ (i.e. the two following expressions should be evaluated to `true`: `equals(get_head(l'),e)` and `equals(get_tail(l'),l)`).

### 3.3 The `%match` construct

The `%match` construct is parameterized by a subject (a list of terms) on which the discrimination should be performed, and a body. As for the `switch/case` construct of C and Java, the body is a list of pairs: *pattern-action*. The pattern is a list of terms (with free variables) which are matched against a list of terms that compose the subject. When the *pattern* matches the subject, the free variables are instantiated and the corresponding *action* is executed. The particularity of this construction is to mix two formalisms: the patterns are written in a pure algebraic specification style using constructors and variables, whereas the action parts are directly written in the native language, using the variables introduced by the patterns. Since TOM has no knowledge of what is done inside an action, the action part should be written in such way that the function has the desired behavior. In our Peano example, the `suc(plus(x,y))` expression corresponds to a recursive call of the `plus` function while the `suc` function is supposed to build a successor. Note that this part has nothing to do with TOM: it only depends on the considered target language. The semantic of a `%match` construct is the following:

***Matching.*** Given a list of terms (subject), the execution control is transferred to the first pattern that matches the subject. If no such pattern exists, the evaluation of the `%match` construct is finished.

***Selected pattern.*** Given a pattern which matches the subject, the associated action is executed, using the free variables instantiated during the matching phase. If the execution control is transferred outside the `%match` construct (by a `goto`, `break`, or `return` statement for example), the matching process ends. Otherwise, it continues in one of the two following cases:

- Syntactic case: the execution control is transferred to the next *pattern-action* whose pattern matches the subject.

– Equational case: since list-matching may return several matches, for each computed match, the list of free variables is instantiated and the same action part is executed one more time (remember that this behavior is only possible when the action part is not terminated by a `goto`, `break`, or `return` for example). When all matches have been computed, as in the syntactic case, the execution control is transferred to the next *pattern-action* whose pattern matches the subject.

***End.*** When no more pattern matches the subject, the `%match` construct ends, and the execution control is transferred to the next target language instruction.
This principle can be used to implement a conditional pattern matching function:

```
Term f(Term t) {
  %match(Nat t) {
    p_1 -> { if(cond) return /* expression related to p_1 */; }
    p_2 -> {          return /* expression related to p_2 */; }
  }
}
```

Let us suppose that patterns `p_1` and `p_2` match the subject `t`: as long as the condition `cond` is not satisfied, the pattern `p_1` is tried (all possible matches are computed). If the condition `cond` can definitively not be satisfied, the second pattern is tried.

## 4 Applications

The concept of TOM started from a very simple remark: "each time a parser is needed, Lex and Yacc[3] are used, but each time pattern matching is needed, a new matching algorithm is implemented". When designing TOM we tried to be as open and generic as possible, and we always had three main requirements in mind:

– TOM should be usable each time a matching operation is needed;
– TOM should be an interesting tool for implementing compilers and transformation tools;
– TOM should be a good support (back-end) for implementations of rule based programming languages.

In the following we illustrate how TOM could be used in three different representative contexts.

### 4.1 Implementing matching operations using TOM

When developing an application which manipulates tree-like data structures (such as symbolic computation, compilation, XML transformation, message exchanging, etc.), the needed for matching can rarely be avoided, simply because some information has to be retrieved. In principle, the extraction of this information can always be explicitly performed by accessing node information, performing some tests and accessing subterm information.

In order to illustrate the expressiveness of TOM we consider a small prototype of a coordination bus where exchanged messages are represented by terms. Components connected to this bus can send a *message* (containing data) to another connected component. We represent the bus as a *queue* that contains messages that have to be delivered. A message contains two fields: destination

[3] or a similar set of tools

and data. In our example, we represent a component which looks for a particular kind of message: a message addressed to a component `b` and whose data has a given `subject`.

In order to illustrate the flexibility of TOM we consider a C program where all data structures are internally defined (we no longer use the ATerm library). Thus we consider the `term` data structure to represent components, messages and data, and we consider the `list` data structure to represent queues.

```
struct term {
  int symbol;
  int arity;
  struct term **subterm;
};
```

```
struct list {
  struct term *head;
  struct list *tail;
};
```

The algebraic data type and the constructors are defined over this concrete implementation (`A`, `B`, `SUBJECT` and `MSG` are integers which encode function symbols):

```
%typeterm Term {
  implement { struct term* }
  get_fun_sym(t)     { t->symbol     }
  cmp_fun_sym(t1,t2) { t1 == t2      }
  get_subterm(t, n)  { t->subterm[n] }
}
```

```
%op Term a       { fsym { A } }
%op Term b       { fsym { B } }
%op Term subject(Term)
           { fsym { SUBJECT } }
%op Term msg(Term,Term)
           { fsym { MSG }     }
```

Since we want to manipulate and perform pattern matching on a list data structure we have to introduce algebraic data sort (using `%typelist`) and the corresponding `cons`-operator (using `%oplist`):

```
%typelist List {
  implement          { struct list*      }
  get_fun_sym(t)     { CONS              }
  cmp_fun_sym(t1,t2) { t1 == t2          }
  equals(l1,l2)      { list_equal(l1,l2) }
  get_head(l)        { l->head           }
  get_tail(l)        { l->tail           }
  is_empty(l)        { (l == NULL)       }
}
```

```
%oplist List cons( Term* ) {
  fsym          { CONS             }
  make_empty()  { NULL             }
  make_add(l,e) { build_list(e,l) }
}
```

In TOM, a list-operator accepts a sequence of elements as argument and returns a list structure: `List cons( Term* )`. In the following function, we use a list-matching pattern to search for a particular message in a given queue:

```
struct list *read_msg_for_b(struct list *queue, struct term *search_data) {
  %match(List queue) {
    cons(X1*,msg(b,subject(x)),X2*) -> {
      if(term_equal(x,search_data)) {
        print_term("read_msg: ",x);
        return concat(X1,X2);
      }
    }
    _ -> { /* msg not found */ return queue; }
  }
}
```

When defining a pattern, a variable under the `cons` operator may be annotated by a `"*"` to declare it as a list-variable. In this case, the variable should be instantiated by a (possibly empty) list. In our context, the pattern `cons(X1*,msg(b,subject(x)),X2*)` looks for a message addressed to component `b` whose data is rooted by the `subject` symbol. The condition checks that the found `x` corresponds to `search_data`. When a message addressed to `b` is found but does not correspond to `search_data`, another match is computed (all possible instances of `X1`, `x` and `X2` are tried). If no match satisfies this condition, the default case is executed.

### 4.2 Implementing compilers and transformation tools

The presented language extension has an implementation: `jtom`[4]. One characteristic of this implementation is that it is written in TOM itself (Java+TOM to be more precise).

Compiling a program consists in transforming this program (written in some *source language*) into another equivalent program written in some *target language*. This transformation can be seen as a textual or syntactic transformation, but in general, this transformation should be done at a more abstract level to ensure the equivalence of the two programs. A good and well-known approach consists in performing the AST that represents the program.

Representing an AST can be done in a "traditional way" by defining a data structure or a class (in an object oriented framework) for each kind of node. Another interesting approach consists in representing this tree by a term. Such an approach has several advantages.

- The manipulated information has a universal representation: a term.
- Since the AST is a term, and not a collection of spreaded objects in memory, it can be printed and exchanged with other tools at any stage of the transformation.
- All the information is always available in the term itself.

Thus, given a program, its compilation can be seen as the transformation of a term (the AST of the source language program) into another term (the AST of the target language program). Transformation rules are usually expressed by pattern matching, which is exactly what TOM is suited for.

The implementation of the TOM compiler is an application of this principle: it is composed of several phases that respectively transform a term into another one. The general layout of the

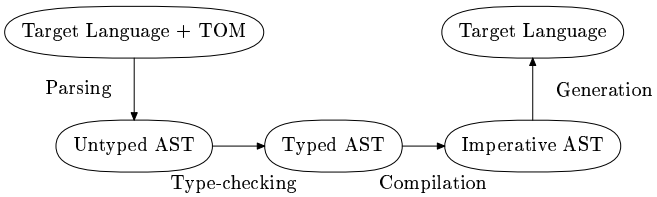


**Fig. 1.**

compiler is shown in Figure 1. As illustrated, four main compilation phases can be distinguished. Each phase corresponds with an abstract syntax whose signature is defined in TOM, using the signature definition formalism presented in sections 3.1 and 3.2.

[4] available at `http://elan.loria.fr/Toolkit`

*Parsing.* The TOM parser reads a program enriched by TOM constructs and generates an Abstract Syntax Tree. As mentioned previously, the source language is a superset of the target language, and we have the following particular equation: `source language = target language + TOM constructs`.

In order to be as general as possible, the current TOM parser is only slightly dependent on the supported target languages. In particular, it does not include a full native language parser: it should only be able to parse comments, strings, and should recognize the beginning and the end of a block ('{' and '}' in C or Java). Using this knowledge, the parser can detect and parse all TOM constructs. The resulting Abstract Syntax Tree is a list of ASTs of two kinds:

- A *Target Language Part* is a string that contains a piece of code written in the target language. This part does not contain any TOM construct;
- A *TOM Construct Part* is an AST that represents a TOM construct.

The role of the TOM compiler consists in replacing all *TOM Construct Parts* by new *Target Language Parts*, without modifying, and even parsing, the remaining *Target Language Parts*. When considering the Naturals example, after parsing, the pattern $suc(y)$ is represented by the following AST:

```
Term(Appl(Name("suc"),[Appl(Name("y"),[])]))
```

Informally, this means that an operator called `suc` is applied to a list of subterms. This list is composed of a unique term which corresponds to the application of `y` to the empty list. At this current stage, it is not yet possible to know whether `y` is a variable or a constant. We can also remark that there is no type information.

*Type-checking.* For sake of simplicity, no type information is needed when writing a matching construct. In particular, TOM variables do not need to be declared, and the definition of the signature can appear anywhere. Consequently, any constant not declared in the signature naturally becomes a variable. Unfortunately, it makes the compilation process harder. During this phase, the TOM type-checker tries to determine the type of each TOM construct and modifies the AST accordingly. The output formalism of this phase is a typed TOM AST as exemplified below:

```
Term(Appl(Symbol(Name("suc"),
     TypesToType([Type(TomType("Nat"),TLType("Term"))],
                 Type(TomType("Nat"),TLType("Term"))), TLCode("f_suc")),
     [Variable(Name("y"),Type(TomType("Nat"),TLType("Term")))]))
```

We can notice that the AST syntax (`Term`, `Appl`, `Name`, *etc.*) has been extended by several new constructors, such as `Symbol`, `TypesToType`, `Variable`, *etc.* A term corresponds now to the application of a *symbol* (and no longer a *name*)) to a list of subterms. A symbol is defined by its name, its profile and its implementation in the target language (`f_suc` in this example). We can also notice that the profile contains two kinds of type information: the algebraic specification type (`Nat`) and the implementation of this type in the target language (`Term`).

*Compilation.* This phase is the kernel of the TOM compiler: it replaces all TOM constructs by a sequence of imperative programming language instructions. To remain independent of the target language, this transformation is performed at the abstract level: instead of generating concrete target language instructions, the compilation phase generates abstract instructions such as `DeclareVariable`, `AssignVariable`, `IfThenElse`. The output formalism also contains some

abstract instructions to access the term data structure, such as `GetFunctionSymbol`, `GetSubterm`, *etc.* After compiling the previous term, we get the following AST (for a better readability, some parts have been removed and replaced by "..."):

```
CompiledMatch([
  Declaration(Variable(Position([match1,1]),Type(TomType("Nat"),TLType("Term")))),
  Assign(Variable(Position([match1,1]),...),
         Variable(Name("t2"),Type(TomType("Nat"),TLType("Term"))))
  ...
  IfThenElse(
    EqualFunctionSymbol(Variable(Position([match1,1]),...),
                        Appl(Symbol(Name("suc"),...))),
    Assign(Variable(Position([match1,1,1]),...),
           GetSubterm(Variable(Position([match1,1]),...),0))
    ...
    Action([TL("return suc(plus(x,y));")]),
    ...
    // else part
  )
  ...
)
```

The main advantage of this approach is that the algorithm for compiling pattern matching is independent from the target language, but also independent from the term data structure. During this phase, a `Match` construct is analyzed, and depending of its structure, abstract instructions are generated (a `Declaration` and an `Assignment` when a variable is encountered for example, or a `IfThenElse` when a constructor is found for example).

```
TermList genTermMatchingAutomata(Term term, TermList path, TermList actionList) {
  %match(Term term) {
    Variable(_, termType) -> {
      assign = makeAssign(term, makeVariable(makePosition(path),termType));
      action = makeAppl(afun_Action, actionList);
      return assign.append(action);
    }
    UnamedVariable(termType) -> {
      action = makeAppl(afun_Action, actionList);
      return action;
    }
    Appl(symbol:Symbol(...),termArgs) -> {
      // generate an Declarations, Assignments and an IfThenElse
      failureList = makeList();
      ...
      succesList  = declarationList.append(assignementList.append(automataList));
      cond = makeAppl(afun_EqualFunctionSymbol,subjectVariableAST,term);
      return result.append(makeIfThenElse(cond, succesList,failureList));
    }
  }
}
```

*Generation.* This phase corresponds to the back-end generator: it produces a program written in the target language. Currently, the back-end supports three target languages: C, Java and

Eiffel. The mapping between the abstract imperative language and the concrete target language is implemented by pattern matching. To each abstract instruction corresponds a pattern, and an associated action which generates the correct sequence of target language instructions. In Java for example, the pattern-action associated to the `IfThenElse` abstract instruction is:

```
IfThenElse(cond,succes,failure) -> {
  prettyPrint( "if(" + generate(cond) + ") {" );
  generate(succes);
  prettyPrint( "} else {" );
  generate(failure);
  prettyPrint("}");
}
```

Due to lack of space, we cannot give much more detail about the compilation of TOM. But, our experience clearly shows that the main interests of TOM can be characterized by the expressiveness and the efficiency introduced by the powerful matching constructs. In practice, the use of pattern matching and list-matching helps the programmer to clearly express the algorithms and, as illustrated in Table 1, it reduces the size of the programs by a factor 2 or 3 in average. In Table 1, we presents statistics for three typical TOM applications, corresponding to the three main components of the system: the type-checker, the compiler, and the generator. For each component, we report the self-compilation time in the last column (measured on a Pentium III, 1200 MHz). The first two columns give some size information. For instance, the type-checker consists of 555 lines including 40 pattern matching constructs. After being compiled, the generated Java code consists of 1484 lines.

| Specification | TOM (patterns/lines) | Generated Java code (lines) | TOM to Java compilation time (s) |
|---|---|---|---|
| TOM checker | 40/555 | 1484 | 0.930 |
| TOM compiler | 81/1490 | 2833 | 1.322 |
| TOM generator | 87/1124 | 3804 | 3.004 |

**Table 1.** Size and compilation time for three TOM specifications

### 4.3 TOM as a back-end for rule-based programming systems

In the field of compiler construction, but more generally in the domain of symbolic computation, the concept of rewrite rule based language is widely used [6, 1, 23, 12]. The emergence of efficient compilers make the use of rewriting an interesting alternative to develop real applications. Among the different rule-based programming languages, let us mention ASF+SDF [11, 23, 21], Cafe-OBJ [7], ELAN [2], Stratego [24, 9] and Maude [4]. For most of these systems, compilers generating C code are developed. Of course, each compiler integrates an algorithm to efficiently compile the pattern matching needed to apply the rewrite rules.

The design of TOM follows our work on the efficient compilation of rewriting as implemented in the ELAN compiler [25, 10]. As a consequence, TOM is particularly well-suited for implementing concepts and features coming from rule-based programming languages. The main advantages of

using and developing the TOM system is to reduce the implementation effort: this could be done by integrating efficient matching algorithms [20, 18] in this common framework, in such a way that each algorithm is implemented only once and used by several systems.

In this direction, a prototype of ELAN compiler using TOM as back-end has already been successfully implemented for a subset of the language. Furthermore, the ASF+SDF group[5] and the ELAN group[6] are currently designing a common extensible compiler based on TOM. This task seems to be possible since the compilation of a rewrite system can be divided into two disjoint compilation phases. The first phase consists in compiling the matching process, while the second one aims at building the result terms obtained after one step of rewriting. This approach allows to define two different languages based on matching: only the second phase has to be targeted for the considered language.

We only sketch the compilation principle of a set of rules $R$. The idea is to generate a function `f` in the target language for each *defined* symbol $f$ occurring at the top-position of a left-hand side of some rule in $R$. This function is defined by pattern matching, with a case

$$l_1, \ldots, l_n \texttt{ -> action\_r}$$

for each rule $f(l_1, \ldots, l_n) \to r$ in $R$. The related action is derived from the right-hand side $r$ of the rule by compiling $r$ into an expression in the target language. The same compilation principle holds for right-hand sides introduced by conditional rules (used in ASF+SDF), and rules with local assignments and strategies (used in ELAN).

## 5 Related Work

Several systems have been developed in order to integrate pattern matching and transformation facilities into imperative languages. Let us mention for instance R++ [5, 19], App [16], Prop [13], Pizza [17] and JaCo [27].

Each of these systems has its own specificity. 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. Prop is a multi-paradigm extension of C++, including pattern matching constructs. Pizza is a Java extension that supports parametric polymorphism, first-class functions, class cases and pattern matching. Finally, JaCo is an extensible Java compiler written in an extension of itself: Java + extensible algebraic types.

All these approaches are interesting and propose some very powerful constructs, but from our point of view, they are too powerful and less generic than TOM. In the spirit, Prop, Pizza and JaCo are very close to TOM: they add pattern matching facilities to a classical imperative language, but the approach is not similar. Indeed, Prop, Pizza and JaCo are more intrusive than TOM: they really extend C++ and Java with several new pattern matching constructions. On the one hand, the integration is better and more transparent. But on the other hand, the term data structure cannot be user-defined: 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. This may be a drawback because it is not easy to convince a user to program in a declarative way if the first thing to do is to translate the existing main data structures.

[5] `http://www.cwi.nl/projects/MetaEnv`
[6] `http://elan.loria.fr`

## 6 Conclusion and Further Work

In this paper we have presented a tool for non-intrusive pattern-matching which seems to be particularly well-suited to perform transformations of trees/terms. In our opinion, TOM 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). TOM does not depend on a particular implementation of terms, this gives the user a large freedom in the choice of a term representation to consider. TOM even does not impose a homogeneous term representation, different data structures can be mixed inside terms.

For sake of expressiveness, it is crucial to continue the integration of equational matching into TOM. For now, we have successfully considered the case of list-matching, which was already supported by Asf+Sdf. In the future, we still have to go beyond this first case-study by considering other more complicated and useful equational theories like Associativity-Commutativity and its extensions.

The current implementation of TOM is designed in a multi-layered component approach: each component takes an AST as input, and returns a new AST. With such an approach, the dependence between the different components is very low and exists only at the data representation level. We are currently working on a more flexible version of two components: the compiler and back-end generator. The current matching compiler generates a program written in a fixed abstract intermediate language. This abstract language is then translated by the generator into the chosen target language. Unfortunately, some abstract instructions may have no straightforward translation into a given target language. This is the case of `break` or `goto` in Eiffel for example. The matching compiler is parameterized by a set of available intermediate instructions (without `gotos` for example). Similarly, we plan to make the back-end generator more generic with respect to supported target languages.

### Acknowledgments

We would like to thank Mark van den Brand and Eelco Visser for fruitful discussions and comments on the design of TOM.

## References

1. U. Aßmann. *OPTIMIX, A Tool for Rewriting and Optimizing Programs.* Chapman-Hall, 1998.
2. P. Borovanský, 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*, ENTCS. Elsevier Sciences, 1998.
3. T. H. Brus, 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, pages 364–384, Portland, Oregon, USA, 1987. Springer-Verlag.
4. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), 1996. ENTCS.
5. J. M. Crawford, 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*, pages 490–497, Menlo Park, 1996. AAAI Press / MIT Press.

6. C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.
7. K. Futatsugi 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.
8. P. Hudak, S. L. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *SIGPLAN Notices*, Mar, 1992.
9. P. Johann and E. Visser. Warm fusion in stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 2000.
10. H. Kirchner 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*, 11(2):207–251, 2001.
11. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
12. D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Proceedings of Compiler Construction, 10th International Conference*, volume 2027 of *LNCS*, pages 52–68. Springer, 2001.
13. L. Leung. Prop homepage: `http://cs1.cs.nyu.edu/phd_students/leunga/prop.html`.
14. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
15. P.-E. Moreau and O. Zendra. $GC^2$: A Generational Conservative Garbage Collector for the ATerm Library. Technical report A02-R-126, LORIA - INRIA, France, 2002.
16. G. Nelan. App homepage: `http://www.primenet.com/~georgen/app.html`.
17. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, USA, 1997.
18. M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, editors, *Proceedings of Compiler Construction, 4th International Conference*, volume 641 of *LNCS*, pages 258–270. Springer, 1992.
19. R++. homepage: `http://www.research.att.com/sw/tools/r++`.
20. R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern maching. In W. Kuich, editor, *Proceedings of ICALP 92*, volume 623 of *Lecture Notes in Computer Science*, pages 247–260. Springer-Verlag, 1992.
21. M. van den Brand and al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of Compiler Construction, 10th International Conference*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
22. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.
23. M. G. J. van den Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor, *Proceedings of Compiler Construction, 8th International Conference*, volume 1575 of *LNCS*, pages 198–213. Springer, 1999.
24. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
25. M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), 1996. Springer-Verlag.
26. P. Weis, M. Aponte, A. Laville, M. Mauny, and A. Suárez. *The CAML Reference Manual*. INRIA-ENS, 1987.
27. M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 6th ACM SIGPLAN International Conference on functional Programming (ICFP'2001), Florence, Italy*, pages 241–252. ACM Press, 2001.