

A Pattern Matching Compiler for Multiple Target Languages

Pierre-Etienne Moreau¹, Christophe Ringeissen¹, and Marian Vittek²

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

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. 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 (ASTs 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 [2, 6, 12], 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. There are two common solutions to this problem.

One possibility would be to enrich an existing imperative programming language with pattern matching facilities [5, 7, 8, 13]. This hard-wired approach ties users to a specific programming language. The situation is thus little better than that in declarative languages. Furthermore, because terms are built-in, user-defined data structures must be converted to the term structure. Such marshalling complicates the user's program, and it incurs a significant performance penalty.

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 [10] where a C library called ATERMS [9] has been developed. In this library, pattern matching is implemented via a function called `ATmatch`, which consists in 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.

To solve the deficiencies of the above two solutions, and in particular for the 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 a 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 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 [3, 11]. Our tool can be viewed as a YACC-like compiler translating patterns into executable pattern matching automata. Similarly 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 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 of 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.

Generality TOM is able to consider multiple target languages (C, Java, and Eiffel). TOM is implemented in TOM itself as a series of AST transformations. 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 a single variable may have multiple 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. Further 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 how TOM is used in designing a compiler, via some transformations of ASTs performed by pattern matching: in fact, this example is the current implementation of TOM itself (Section 4.2). 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 new 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.

For expository reasons, we assume that TOM only adds one new construct: `%match`. This construct is similar to the `match` primitive found in ML and related 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 may thus be 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 understanding of TOM's features, let us consider a simple symbolic computation (addition) defined on Peano integers represented by *zero* and *successor*. When using Java as the 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 have 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, where only the TOM constructs are parsed in detail. The first phase of the transformation process consists of *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 uses 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. In particular, 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: $\{zero : \mapsto Nat, \text{ suc} : Nat \mapsto Nat, \text{ plus} : Nat \times Nat \mapsto Nat\}$

For simplicity, we consider in this example that the `ATERM` library [9] is used for term representation. This library is a concrete implementation of the Annotated Terms data type (`ATERMS`). In particular it defines an interface to create and manipulate term data structures. Furthermore, 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 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)` and `AFun f_suc = makeAFun("suc",1)`. The representation of the constant `zero`, for example, is given by `makeApp1(f_zero)`. Similarly, given a Peano integer `t`, its successor can be built by `makeApp1(f_suc,t)`. So far we have shown how to represent data using the `ATERM` library, and how defining matching with TOM, but, we have yet to reveal how these two notions are related. Given a Peano 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).isEqual(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 the next section.

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 in-depth 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 two main constructs: `%typeterm` and `%op` are used to define respectively algebraic sorts and many-sorted signature of the algebraic constructors.

3.1 Sort definition

In TOM, terms, variables, and patterns are many-sorted. Their algebraic sorts have to be introduced by the `%typeterm` primitive. In addition to this primitive, the mapping from algebraic sorts to concrete sorts (the target language type, such as `Term`) has to be defined. Several sub-functions are used for this purpose. 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.isEqual(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. This is analogous to the treatment of semantic actions in YACC. 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, 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 s_1 and s_2). This predicate should return `true` if the symbols s_1 and s_2 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).
- `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 t_1 and t_2). Similarly to `cmp_fun_sym(s1,s2)`, this predicate should return `true` if terms t_1 and t_2 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(t2) }”

To clarify the presentation we only used the ATERMS data structure, but it should be noticed that any other data structure could be used as well. 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)`”. We stress once again that because TOM has no knowledge, 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.
- `make(t1, ..., tn)` denotes a function parameterized by n variables, where n is the arity of the considered symbol. This function should define how a term rooted by the considered symbol can be built. The definition of this function is optional since it is not used by TOM during the matching phase. However, when defined, it can be used by the programmer to simplify the construction of terms (see Section 3.4).

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

<pre>%op Nat zero { fsym { f_zero } make { makeAppl(f_zero) } }</pre>	<pre>%op Nat suc(Nat) { fsym { f_suc } make(t) { makeAppl(f_suc,t) } }</pre>
---	--

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 identifiers not defined as constants are seen as variables. Thus, the pattern `suc(y)` corresponds to the term $suc(y)$ where y is a variable, and the pattern `suc(zero)` corresponds to the Peano integer *one*.

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. Note that this is a hybrid language construct, mixing 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 the `%match` construct is as follows:

Matching: given a 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, 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.

3.4 Making terms

In addition to sort definition, construction definition and matching constructs, TOM provides a mechanism to easily build ground terms over the defined signature. This mechanism, called *back-quote* (and written ‘```’), can be used in any target language block as a kind of escape mechanism. The syntax is simple: the *back-quote* is followed by a well-formed term written in prefix notation. The last closing parenthesis denotes the end of the escape mechanism.

Considering the previously defined addition function on Peano integers, the right-hand side could have been written ‘`suc(plus(x,y))`’ and the construction of the `suc` node would have been done by TOM, using the `make` attribute introduced in Section 3.2.

3.5 Equational matching

An important feature of TOM is to support equational matching. In particular, list matching, also known as associative matching with neutral element.

Since a list can be efficiently and naturally represented by a (head,tail) tuple, TOM provides an extra construct for defining associative data structures: the `%typelist` primitive. When defining such a structure, three extra access functions have to be 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 . Using `ATERMS`, it can be defined by “`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 obvious: “`is_empty(l) { l.isEmpty() }`”.

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

- `make_empty()` should return an empty list. This object corresponds to the neutral element of the considered data structure.
- `make_insert(e,l)` should return a new list l' where the element e is inserted at the head of the list l (i.e. expressions `equals(get_head(l'),e)` and `equals(get_tail(l'),l)` should be `true`).

One characteristic of list-matching is the possibility to return several matches. Thus, the semantic of the `%match` construct has to be extended as follows:

Selected pattern: given a pattern which matches the subject, for each computed match, the list of free variables is instantiated and the action part is executed. If the execution control is transferred outside the `%match` construct the matching process ends. Otherwise, *another match is computed*. When no more match is available, the execution control is transferred to the next *pattern-action* whose pattern matches the subject.

This principle can be used to implement a sorting algorithm using a conditional pattern matching definition. In the following, we consider an associative data structure `List` and an associative operator `conc`:

<pre>%typeterm List { implement { TermList } get_fun_sym(t) { f_conc } cmp_fun_sym(t1,t2) { t1 == t2 } equals(l1,l2) { l1 == l2 } get_head(l) { l.getFirst() } get_tail(l) { l.getNext() } is_empty(l) { l.isEmpty() } }</pre>	<pre>%oplist List conc(Term*) { fsym { f_conc } make_empty() { makeList() } make_insert(e,l) { l.insert(e) } }</pre>
--	--

Considering that two `Term` elements can be compared by a function `greaterThan`, a sorting algorithm can be implemented as follows:

```
public TermList sort(TermList l) {
  %match(List l) {
    conc(X1*,x,X2*,y,X3*) -> {
      if(greaterThan(x,y)) { return 'sort(conc(X1*,y,X2*,x,X3*))'; }
    }
    _ -> { return l; }
  }
}
```

In this example, one can remark the use of *list variables*, annotated by a `*`: such a variable should be instantiated by a (possibly empty) list. Given a partially sorted list, the `sort` function tries to find two elements x and y such that x is greater than y . If two such elements exist, they are swapped and the `sort` function is recursively applied. Otherwise, all other possible matches are tried (unsuccessfully). As a consequence, the first pattern-action is not exited by a `return` statement. Thus, as mentioned previously, the execution control is transferred to the next pattern-action whose pattern matches the subject (the second one in this example), and the sorted list `l` is returned.

4 Applications

4.1 Implementing matching operations using Tom

As an example of using list matching, we consider the problem of retrieving information in a queue of messages containing two fields: destination and data. In our example, we define a function which looks for a particular kind of message: a message addressed to `b` and whose data has a given `subject`. To illustrate the flexibility of TOM, we no longer use the `ATERM` library, and all data structures are internally defined. Thus we use the language `C`, and we respectively consider `term` and `list` data structures to represent messages and queues.

<pre>struct term { int symbol; int arity; struct term **subterm; }; %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 } }</pre>	<pre>struct list { struct term *head; struct list *tail; }; %typelist List { implement { struct list* } get_fun_sym(t) { CONC } 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 conc(Term*) { fsym { CONC } make_empty() { NULL } make_insert(e,l) { cons(e,l) } }</pre>
--	---

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) {
    conc(X1*,msg(b,subject(x)),X2*) -> {
      if(term_equal(x,search_data)) {
        print_term("read_msg: ",x);
        return 'conc(X1*,X2*);
      }
    }
  }
  _ -> { /* msg not found */ return queue; }
}
```

In this function, 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`³. 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 transformation of 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. First, it is a universal representation for every manipulated information. Second, compared to a collection of spreaded objects in memory, a term can be more easily printed and exchanged with other tools at any stage of the transformation. Last, 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 compiler is shown in Figure 1.

³ available at <http://elan.loria.fr/tom>

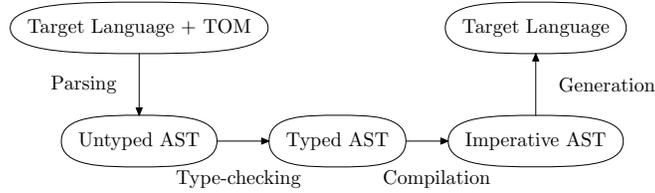


Fig. 1. General layout of the TOM compiler

As illustrated, four main compilation phases can be distinguished. Each phase corresponds to an abstract syntax whose signature is defined in TOM, using the signature definition formalism presented in Sections 3.1 and 3.2.

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 AST is a list of nodes of two kinds: (1) A *Target Language Node* is a string that contains a piece of code written in the target language. This node does not contain any TOM construct; (2) A *TOM Construct Node* is an AST that represents a TOM construct. The role of the TOM compiler consists in replacing all *TOM Construct Nodes* by new *Target Language Nodes*, without modifying, and even parsing, the remaining *Target Language Nodes*. When considering the Naturals example, after parsing, the pattern $suc(y)$ is represented by the following AST:

```
Term(App1(Name("suc"), [App1(Name("y"), [])]))
```

Informally, this means that an operator called `suc` is applied to a list of subterms. This list is made of a unique term corresponding 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 determines 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 does not depend on neither the target language nor the term data structure. During this phase, a `match` construct is analyzed, and depending on 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 genTermMatchAuto(Term term, TermList path, TermList actionList) {
  %match(Term term) {
    Variable(_, termType) -> {
      assign = 'Assign(term,Variable(Position(path),termType));
      action = 'Action(actionList);
      return assign.append(action);
    }
    UnnamedVariable(termType) -> {
      action = 'Action(actionList);
      return action;
    }
    Appl(Symbol(...),termArgs) -> {
      // generate Declarations, Assignments and an IfThenElse
      failureList = makeList();
      ...
      succesList = declList.append(assignList.append(automataList));
      cond = 'EqualFunctionSymbol(subjectVariableAST,term);
      return result.append('IfThenElse(cond,succesList,failureList));
    }
  }
}

```

Generation. This phase corresponds to the back-end generator: it produces a program written in the target language. 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 and C 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 the following table, it reduces the size of the programs by a factor 2 or 3 in average. 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. As illustrated by the compilation speed, the efficiency of the generated code is sufficient in practice for this kind of application.

Specification	TOM (patterns/lines)	Generated Java code (lines)	TOM to Java compilation time (s)
TOM checker	40/555	1484	0.331
TOM compiler	81/1490	2833	0.600
TOM generator	87/1124	3804	0.812

5 Related Work

Several systems have been developed in order to integrate pattern matching and transformation facilities into imperative languages. For instance, `R++` [1] and `App` [7] are preprocessors for `C++`: the first one adds production rule constructs to `C++`, whereas the second one extends `C++` with a `match` construct. `Prop` [4] is a multi-paradigm extension of `C++`, including pattern matching constructs. `Pizza` [8] is a `Java` extension that supports parametric polymorphism, first-class functions, class cases and pattern matching. Finally, `JaCo` [13] is an extensible `Java` compiler written in an extension of itself: `Java + extensible algebraic types`.

All these approaches propose some very powerful constructs, but from our point of view, they are too powerful and less generic than TOM. In spirit, `Prop`, `Pizza` and `JaCo` are very close to TOM: they add pattern matching facilities to a classical imperative language, but the method of achieving this is completely different. 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. This may be a drawback when one wants to extend an existing project, since it is hard to convince a user to program in a declarative way if the first thing to do is to translate the existing main data structures.

6 Conclusion and Further Work

In this paper we have presented a non-intrusive tool for extending existing programming languages with pattern matching. 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). In this context, a prototype of `ELAN` compiler using TOM as back-end has already been successfully implemented for a subset of the language, and the `ASF+SDF` group⁴ and

⁴ <http://www.cwi.nl/projects/MetaEnv>

the ELAN group⁵ are currently designing a common extensible compiler based on TOM.

For the sake of expressiveness, it is important 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.

Acknowledgments: We would like to thank Mark van den Brand, Jurgen Vinju and Eelco Visser for fruitful discussions and comments on the design of TOM. We also thank the anonymous referees for valuable comments and suggestions that led to a substantial improvement of the paper.

References

1. 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 Eighth Innovative Applications of Artificial Intelligence Conference*, pages 490–497, Menlo Park, 1996. AAAI Press / MIT Press.
2. 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.
3. 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.
4. L. Leung. Prop homepage: cs1.cs.nyu.edu/phd_students/leunga/prop.html.
5. J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In V. Dahl and P. Wadler, editors, *Proceedings of PADL'03*, volume 2562 of *LNCS*, pages 110–127. Springer-Verlag, 2003.
6. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
7. G. Nelan. App homepage: www.primenet.com/~georgen/app.html.
8. 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.
9. 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.
10. M. G. J. van den Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In *Proceedings of Compiler Construction, 8th International Conference*, volume 1575 of *LNCS*, pages 198–213. Springer, 1999.
11. M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *LNCS*, pages 154–168, New Brunswick (New Jersey), 1996. Springer-Verlag.
12. P. Weis, M. Aponte, A. Laville, M. Mauny, and A. Suárez. *The CAML Reference Manual*. INRIA-ENS, 1987.
13. 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.

⁵ <http://elan.loria.fr>