Tom: Piggybacking rewriting on java

Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, Antoine Reilles

Tom: Piggybacking Rewriting on Java

Emilie Balland[1], Paul Brauner[1], Radu Kopetz[2], Pierre-Etienne Moreau[2], and Antoine Reilles[3]

[1] UHP & Loria
[2] INRIA & Loria
[3] INPL & Loria

Abstract. We present the TOM language that extends JAVA with the purpose of providing high level constructs inspired by the rewriting community. TOM furnishes a bridge between a general purpose language and higher level specifications that use rewriting. This approach was motivated by the promotion of rewriting techniques and their integration in large scale applications. Powerful matching capabilities along with a rich strategy language are among TOM's strong points, making it easy to use and competitive with other rule based languages.

1 Introduction

Term Rewriting provides a theoretical framework that is very useful to model, study, and analyze various parts of a complex system, from algorithms to running software. During the last 20 years, there were many successful attempts in understanding, certifying, and proving properties of software, such as termination or confluence.

Term Rewriting is also a great tool for building software. Following the development of Lisp, the first equational interpreters, OBJ and EQI were introduced in 1975 by J. A. GOGUEN and M. J. O'DONNELL. Many tools have integrated the notion of term rewriting in their implementation, among them, let us mention Reve 1984, ML 1985, Clean 1986, RRL 1988, ASF+SDF 1989, Spike 1992, ELAN 1993, Larch Prover 1993, Caml 1993, Otter 1994, MAUDE 1995, CafeOBJ 1995, C*i*ME 1996, DMS 1997, Stratego 1998, Hats 1999, TOM 2001, *etc.* Some of them are not only tools which use the notion of term rewriting, but general purpose programming languages whose semantics and execution mechanism are fully based on the notions invented, defined, and studied by the rewriting community: *term*, *pattern matching*, *equational theory*, *rewrite rule*, *strategy*, *etc.* to mention just a few of them.

Throughout the course of 10 years we developed the ELAN system [3]. We integrated some of the best algorithms to implement pattern-matching, rule based normalization, and non-deterministic computations. As a result, ELAN is certainly, along with MAUDE [8], one of the most established term rewriting based implementations which efficiently compiles associative-commutative rewriting combined with non-deterministic strategies. This was a very fruitful experience and it taught us many lessons. One of them is that implementing

a good term data-structure is difficult, data conversion (also known as *marshalling*) is often a bottleneck, integrating built-in data-types such as integers or doubles is not so easy, mutable data-structures, such as arrays, are essential for the implementation of efficient data-structures like hash-tables. But one of the most important things we learned is that even if *efficiency* is important to make our technology credible, *integration capabilities* are even more important to make our research widely used both in academic and industrial projects.

In 2001 we started the design of a new language called Tom [12], available at `http://tom.loria.fr/`, whose goal was to pursue the promotion of term rewriting by making our concepts and techniques more easily available. One solution could have been to increase our implementation efforts by providing more data-structures, more libraries, more input/output facilities, more threads, more native interfaces, more graphical user interfaces, more, more, more. But we have to admit that these are difficult tasks and that many other languages already do that very well. In fact, this is not our main business. Therefore, we chose another approach: make our technology available on top of an existing language. This concept is called *Formal Island* [1].

In a first possible scenario, we start from an already existing application and our goal is to implement new functionalities, or re-implement some old ones, using rewriting. The expected result is a more concise and abstract description, and the possibility to reason about this new piece of code. In this case, the data-structure used by the application are already defined. We cannot translate them forth and back before and after each rewrite step, this would introduce unacceptable marshalling costs. Behind the notion of *formal island* there is the notion of *formal anchor*, also called *mapping*, which describes how a concrete data-structure can be seen as an algebraic term. This idea, related to P. Wadler's views, allows Tom to rewrite any kind of data structure, as long as a *formal anchor* is provided. In a second scenario, the application is both specified and implemented at the same time, using rewrite rules. In that case, the system should be easy to use: the definition of an algebraic signature, the definition of rules, and that's all! In this paper we focus on this second approach, which is exactly what Tom provides when the underlying host language is Java.

The paper is organized as follows. In Section 2, we present how term rewriting is implemented and integrated into Java. Section 3 focuses on the strategy language and its control mechanism. Section 4 exposes meta-programming features added to the strategy language for managing non-deterministic computations and for modifying strategies at runtime. The two following sections present respectively some key details of the Tom implementation and some significant applications. Section 7 and 8 present related work and conclude.

2 Implementing term rewriting

Term rewriting systems are mostly concerned with computing reduced forms of a ground term *wrt.* a set of rules. To this end, the Tom language allows the definition of many-sorted signatures that are used to generate correctly typed

algebraic terms. Further on, a rewrite system based on syntactic or equational pattern matching can be defined and applied on these terms.

First order terms. Similarly to other rule based or functional languages, TOM provides a construct to define many-sorted algebraic signatures:

```
module Peano
  Nat = zero() | suc( Nat )
```

In this example, `Peano` is the name of the module, `Nat` is a sort, `zero` and `suc` are constructors. More generally, a module may import another module, such as `Peano`, or any other predefined one like `String` or `int`. Given a signature, a well formed and well typed term can be built using the backquote (`) construct: `zero(), `suc(zero()) are correct, as opposed to `suc(zero(),zero()) or `suc(3) which are respectively not well-formed and not well-typed.

Due to the fact that TOM is built on top of JAVA, a term can be used in any JAVA expression such as `System.out.print("t = " + `suc(suc(zero())))`. To ensure that the type of a term can be statically checked by the underlying JAVA compiler, the implementation of `suc(suc(zero())) should reflect the type defined by the signature. To solve this problem, we followed a different approach from other classical implementations such as ASF+SDF, OCaml, ELAN, MAUDE, ML, or Stratego.

Usually, the compiler checks that all term manipulations result in correctly typed terms, while at runtime level a generic untyped term implementation is used. In our case, we use a generator [13] that compiles the algebraic signature into a typed term structure that can be directly used by a JAVA programmer. The need for a typed term structure comes also from the fact that pure JAVA code, not handled by the TOM compiler, could create wrongly shaped terms. The generated structures are efficient, and provide types at the implementation level. As a consequence, for each sort a JAVA class with the same name is generated: `Nat t = `suc(suc(zero()))` defines a variable `t` of sort `Nat`. By generating a statically typed implementation we provide a smooth and natural integration of the notions of signature and term into JAVA.

Pattern matching and rewriting. Implementing term rewriting may be considered a simple task. To know if a rewrite rule $l \to r$ can be applied for a ground term t, it is sufficient to have a pattern matching algorithm that computes, when it is possible, a substitution σ such that $\sigma l = t$, and fails otherwise. The application of the rule consists in replacing t by σr. However, real cases are more complicated. The rules may be applied not only on top, but also to subterms; they may have conditions; the patterns may contain symbols that belong to an equational theory (such as associativity and commutativity for example) and the application order of several rules may be prioritized. Besides, one may be interested in not only getting a single result, but also getting the set of all possible reductions of a given term t. The combinations of all these variants are difficult to tackle with. In practice, each implementation considers only a subset of them.

A main objective of Tom is to be as generic as possible. Therefore, we provide a key primitive on top of Java that can be used to handle most of the situations described above: the `%match` construct. Its semantics is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` is parameterized by a list of subjects (*i.e.* expressions evaluated to ground terms) and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and fresh variables, without any restriction of linearity. The right-hand side is *not* a term, but a Java statement that is executed when the pattern matches the subject. But thanks to the backquote construct (`‘`) a term can be easily built and returned. Similar to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (*i.e.* right-hand side) may be fired for a given subject as long as no `return` or `break` is executed. To implement a simple reduction step, for each rule, we just have to encode the left-hand side by a pattern and consider a Java statement that returns the right-hand side. To encode a rewrite system, the notion of function that already exists in Java is fundamental. For example, the addition and the comparison of Peano integers may be encoded as follows:

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

```
boolean greaterThan(Nat t1, Nat t2) {
 %match(t1, t2) {
   x,x           -> { return false; }
   suc(_),zero() -> { return true;  }
   zero(),suc(_) -> { return false; }
   suc(x),suc(y) -> {
          return ‘greaterThan(x,y); }
}}
```

The reader should note that anonymous variables (`_`) are allowed and that variables such as `x` or `y` do not need to be declared: they are local to each left-hand side and their type is automatically inferred.

List matching. In addition to free constructors, list operators can be also declared. They are a variant of associative operators with neutral element:

```
module Peano
  Nat     = zero() | suc( Nat )
  NatList = conc( Nat* )
```

The notation `Nat*` means that `conc` is a variadic operator where each subterm is of sort `Nat`. It can be seen as a concatenation operator over `Nat` lists: `‘conc()` denotes the empty list while `‘conc(zero(),suc(zero()))` corresponds to the list that contains `zero()` and `suc(zero())`. List operators can be used in the left-hand side of a rule in order to perform list matching:

```
Collection bag = new HashSet();
%match(list) {
  conc(_*,suc(x),_*) -> { bag.add(‘x); }
}
System.out.println("numbers: " + bag.toString());
```

In this example, one can remark the use of *list variables*, annotated by a '`*`', which intuitively corresponds to the Kleene star: such a variable is instantiated by a (possibly empty) list. Note that an action is fired for each pattern and substitution that matches the subject. Since list matching is not unitary, the action `bag.add(‘x)` is evaluated for each element of `list` that matches against `suc(x)`. When applied to a list of Peano integers, this code stores each natural whose successor is in the list into the set represented by `bag`. This non-functional approach is very useful to encode non-deterministic computations such as the exploration of a search space. Combining list operators and conditions allows for the definition of complex algorithms in a concise manner, as illustrated by the following sorting algorithm:

```
public NatList sort(NatList list) {
  %match(list) {
    conc(X1*,x,X2*,y,X3*) -> {
      if(greaterThan(x,y)) { return ‘sort(conc(X1*,y,X2*,x,X3*)); } }
    _ -> { return list; }
  }
}
```

Given a partially sorted list, the `sort` function looks for 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. When the list is sorted this condition cannot be satisfied and the next pattern is tried: the sorted list is returned. This example also shows how a conditional rule can be naturally encoded using the `if` construct provided by Java.

Normal forms. When manipulating non-free algebras, it is convenient to work with terms in normal form. These normal forms are defined by a confluent and terminating rewrite system that is *systematically* applied to each term. This was the purpose of unnamed rules in ELAN for example. Instead of relying on normalization functions that have to be explicitly called by the programmer, Tom proposes to integrate the computation of normal forms into the definition of the data structure. This approach is very close to the recently introduced OCaml *private types* [11]. Normal forms are specified using the notion of *hook* which defines construction functions in the term signature. For instance, suppose that we want to work on $\mathbb{Z}/3\mathbb{Z}$. Then, we have to systematically apply the rule $suc(suc(suc(x))) \rightarrow x$ when creating new terms. This is specified by a *hook* attached to the *suc* operator.

```
module Peano
  Nat = zero() | suc( Nat )
  suc:make(t) {
    %match(t) { suc(suc(x)) -> { return ‘x; } }
  }
```

Each time a *suc* is built, `make(t)` is called with `t` instantiated by the *subterm* of the considered *suc*. This is why the rewrite rule above only rewrites two *suc*. A default allocator is called when no rule can be applied

3 Controlling Rewriting

When using rewriting as a programming or modeling paradigm, it is common to consider rewrite systems that are non-confluent or non-terminating. To be able to use them, it is necessary to exercise some control over the application of the rules. In Tom, a solution would be to use Java to express the control needed. While this solution provides a huge flexibility, its lack of abstraction renders difficult the reasoning about such transformations.

Rewriting based languages provide more abstract ways to express control of rule applications, by using reflexivity and the meta-level for Maude, or the notion of rewriting strategies for ELAN, Stratego [16], or ASF+SDF [4]. Strategies such as *bottom-up*, *top-down* or *leftmost-innermost* are higher-order features that describe how rewrite rules should be applied. We have developed a flexible and expressive strategy language inspired by ELAN, Stratego, and JJTraveler [17] where high-level strategies are defined by combining low-level primitives. For example, the *top-down* strategy is recursively defined by `TopDown(s)` $\triangleq$ `Sequence(s,All(TopDown(s)))`.

Elementary strategies. An elementary strategy corresponds to a minimal transformation. It could be *Identity* (does nothing), *Fail* (always fails), or a set of *rewrite rules* (performs an elementary rewrite step only at the root position). In our system, strategies are type-preserving and have a default behavior (introduced by the keyword `extends`) that can be either `Identity` or `Fail`:

```
%strategy R() extends Fail() {
  visit Nat {
    zero()      -> { return `suc(zero()); }
    suc(suc(x)) -> { return `x; }
  }
}
```

When a strategy is applied to a term t, as in a `%match`, a rule is fired if a pattern matches. Otherwise, the default strategy is applied. For example, applying the strategy `R()` to the term `suc(suc(zero()))` will produce the term `zero()` thanks to the second rule. The application to `suc(suc(suc(zero())))` fails since no pattern matches at root position.

Recursive and parameterized strategies. More control is obtained by combining elementary strategies with *basic combinators* such as `Sequence`, `Choice`, `All`, `One` as presented in [2, 16]. By denoting `s[t]` the application of the strategy `s` to the term `t`, the *basic combinators* are defined as follows:

`Sequence(s1,s2)[t]` $\rightarrow$ `s2[t']` if `s1[t]` $\rightarrow$ `t'`
failure if `s1[t]` fails

`Choice(s1,s2)[t]` $\rightarrow$ `t'` if `s1[t]` $\rightarrow$ `t'`
`s2[t']` if `s1[t]` fails

`All(s)[f(t1,...,tn)]` $\rightarrow$ `f(t1',...,tn')` if `s[t1]` $\rightarrow$ `t1'`,..., `s[tn]` $\rightarrow$ `tn'`
failure if there exists `i` such that `s[ti]` fails

`One(s)[f(t1,...,tn)]` $\rightarrow$ `f(t1,...,ti',...,tn)` if `s[ti]` $\rightarrow$ `ti'`
failure if for all `i`, `s[ti]` fails

An example of composed strategy is `Try(s)` $\triangleq$ `‘Choice(s,Identity())`, which applies `s` if it can, and performs the *Identity* otherwise. To define strategies such as *repeat*, *bottom-up*, *top-down*, *etc.* recursive definitions are needed. For example, to repeat the application of a strategy `s` until it fails, we consider the strategy `Repeat(s)` $\triangleq$ `Choice(Sequence(s,Repeat(s)), Identity())`. In Tom, we use the recursion operator μ (comparable to `rec` in OCaml) to have stand-alone definitions: μ`x.Choice(Sequence(s,x),Identity())`.

The `All` and `One` combinators are used to define tree traversals. For example, we have `TopDown(s)` $\triangleq$ μ`x.Sequence(s,All(x))`: the strategy `s` is first applied on top of the considered term, then the strategy `TopDown(s)` is recursively called on all immediate subterms of the term.

Exploration strategies. Strategy expressions can have any kind of parameters. It is common to have a Java `Collection` as parameter to collect some information in a tree. For example, let us consider the following strategy which collects the direct subterms of an f. This program creates a hash-set, and a strategy applied to `f(f(a()))` collects all the subterms which are under an `f`: *i.e.* {`a()`, `f(a())`}.

```
%strategy Collect(c:Collection) extends Identity() {
  visit T {
    f(x) -> { c.add(‘x); }
  }
}
Collection bag = new HashSet();
‘TopDown(Collect(bag)).apply( ‘f(f(a())) );
```

4 Meta-programming

The strategy language presented in Section 3 is very expressive and powerful to control how a set of rules should be applied. This is very convenient to collect information or traverse a tree for example. But, there is no real support to perform non-deterministic computations as in the exploration of a search space, which is essential when implementing a model checker or a prover for instance. For this purpose, we have added two new extensions to the strategy language.

Reifying $t_{|\omega}$. Given a term t, suppose that we want to compute the set of all its possible successors *wrt.* a rewrite rule $l \rightarrow r$. We have to find all possible redexes, and for each of them to compute all the substitutions that solve the matching problem. In other words, we want to compute $\{t[\sigma_1 r]_{\omega_1}, t[\sigma_2 r]_{\omega_1}, \ldots, t[\sigma_p r]_{\omega_n}, \ldots, t[\sigma_q r]_{\omega_n}\}$, where ω_i denotes a redex position, and σ_j is a substitution such that $\sigma_j l = t_{|\omega_i}$.

Solving this problem involves manipulating the notion of *position* in a term, and some associated operations: getting the subterm at position ω, and replacing this subterm. To the best of our knowledge, there is no rewriting based language where positions, which are internal to the implementation, can be explicitly

manipulated. We introduce a new operation, `getPosition()`, which raises the notion of position at the object level, providing this global information to the level of strategies. To compute the set of all successors of t for example[4], we consider the *top-down* application of a strategy parameterized by the term t itself and a collection `bag`. For each redex position ω (*i.e.* when a pattern matches), we store $t[\sigma r]_\omega$ in `bag`, using `getPosition()` to obtain ω and `replace` to perform the replacement:

```
%strategy Collect(t:T, bag:Collection) extends Identity() {
  visit T {
    a()  -> { bag.add(replace(t, getPosition(), `b())); }
    f(x) -> { bag.add(replace(t, getPosition(), `x));   }
    g(x) -> { bag.add(replace(t, getPosition(), `c())); }
  }
}
T t = `f(g(a()));
`TopDown(Collect(t,bag)).apply(t);
```

The resulting `bag` contains `f(g(b()))`, `g(a())`, and `f(c())`. This reification of the *position* notion to the object level is new in the domain of rewriting based languages, and it adds expressiveness while keeping programs close to their specification.

Rewriting a strategy. Strategy expressions are terms, and thus can themselves be subject to pattern matching and reduction by strategies. This permits dynamical adaptation of a strategy depending on the environment.

For example, suppose that we have two strategies, `s1` and `s2` which can commute. If computing `s2;s1` is more efficient than computing `s1;s2`, we can define a rule to reorder the sequences of `s1` and `s2`:

```
%strategy Reorder() extends Identity() {
  visit T {
    Sequence(s1(),s2()) -> { return `Sequence(s2(),s1()); }
  }
}
```

This strategy can be further applied *top-down* to any strategy `s` with:
`Strategy optimized_s = `TopDown(Reorder()).apply(s)`.

5 Implementation

Since its first version in 2001, Tom itself has been written using Tom. The system is composed of a compiler and a library which defines the strategy language and offers support for predefined data-types such as integers, floats, strings, collections, and many other Java data-structures. The compiler is organized, in a pure functional style, as a pipeline of program transformations (type inference, simplification, compilation, optimization, generation). Each phase transforms a Java+Tom abstract syntax tree (AST) using rewrite rules and strategies. At

[4] many other operations and strategies may be defined

the end a pure Java AST is obtained. The system is composed of 1000 `%match` constructs, and 200 user strategy definitions, totalizing more than 40000 lines of code. The complete environment has been integrated into Eclipse[5] providing a simple and efficient user interface to develop, compile, and debug rule based applications. Each component of the Tom environment is highly modular and has been designed with flexibility and reusability in mind, without introducing any performance overhead. Due to the lack of space, we mention here only a few key ingredients.

Data representation. The generator of data-structures is integrated in Tom but can be also used independently. Given a signature, it generates a set of Java classes that provide static typing. A subtle hash-consing technique is used to offer maximal sharing [14]: there cannot be two identical terms in memory. Therefore, the equality tests are performed in constant time, which is very efficient in many cases including when non left-linear rewrite rules are considered. In addition to the generation of lightweight data-structures, specialized hash-functions are generated for each constructor of the signature, making the generated implementation often more efficient than a hand-coded term data-structure.

Pattern matching. Thanks to the *formal anchor* approach, Tom is not restricted to a fixed term data-structure. We have designed a compilation algorithm where the data-structure is a parameter of the pattern matching algorithm. In particular, Tom can be used to match and rewrite XML documents. Moreover, the underlying host language is also a parameter, making possible the compilation (including list-matching) into different target languages such as C, Java, OCaml, and Python. This approach has been formally studied in [10]. Besides, for each compilation of a set of patterns, Tom provides a Coq proof that the generated code is correct *wrt.* the semantics of pattern matching.

Strategies. Most of the strategy library is written in Tom, making its extension easy. Only a few of low-level elementary strategies (Sequence, Choice, All, One, *etc.*) are implemented in Java. However, the considered object design-patterns for these elementary strategies facilitate their extension also. To add a new probabilistic choice operator for example, less than 30 lines of code have to be written, without any re-compilation of the system. This makes Tom an ideal platform to experiment new paradigms. Once again, the methodology used to implement the strategy library is not restricted to a given term representation, being possible to switch to another one by properly applying the *interface* concept offered by Java.

6 Applications

The Tom system is no longer a prototype. It has been used to implement many large and complex applications, among them the compiler itself. It has also been used in an industrial context to implement a query optimizer for Xquery, a platform for transforming and analysing timed automata using XML manipulation,

[5] `http://www.eclipse.org/`

etc. In this section we focus on some applications where the key characteristics of TOM were particularly useful. *Proof assistant.* lemuridae is a proof assistant for superdeduction [6] (a dynamic extension of sequent calculus). Proof trees benefit from maximal memory sharing which allows for the handling of big proofs while tactics are naturally translated into strategies. Besides, the expressiveness of TOM patterns makes the micro-proofchecker only one hundred lines long and therefore enables a high degree of confidence in the prover.

NSPK. TOM has been used to implement the verification of the Needham-Schroeder public key protocol by model checking. Several strategies have been experimented. The resulting implementation can be compared favorably with state of the art rule based implementation such as MAUDE or Murφ.

Rho-calculus. An interpreter for the rho-calculus [7] with explicit substitutions was developed using TOM. It is surprisingly short and close to the operational semantics of the calculus taking advantage of all the capabilities of TOM. The calculus itself is expressed using rewrite rules and parameterized strategies, while the interactive features and user interface operations take advantage of the underlying JAVA language.

Calculus of structures. TOM is used to implement an automatic prover for the system BV in the calculus of structures [13]. Normal forms are used to implement the several associative-commutative operators with neutral elements while first-order positions allow to manage the high level of non determinism introduced by deep inference.

On several classical benchmarks TOM is competitive with state of the art implementations like ASF+SDF, ELAN, or MAUDE[6]. In the following, `Fibonacci` computes 500 times the 18^{th} Fibonacci number using a Peano representation. `Sieve` computes prime numbers up to 2000 using list matching to eliminate non-prime numbers: $(c_1*, x, c_2*, y, c_3*) \rightarrow (c_1*, x, c_2*, c_3*)$ if x divides y[7]. `Evalsym`, `Evalexp`, and `Evaltree` are three benchmarks based on the normalization of the expression 2^{22} mod 17 using different reduction strategies. These three benchmarks were first presented in [5]. All these examples are available on the TOM source repository[8]. The measurements were done on a MacBook Pro 2.33 GHz, using Java 1.5.0 and gcc 4.0.

	`Fibonacci`	`Sieve`	`Evalsym`	`Evalexp`	`Evaltree`
`ASF + SDF`	0.4 s	24.1 s	1.7 s	2.0 s	1.6 s
ELAN	1.1 s	–	5.3 s	11.8 s	10.1 s
MAUDE	2.3 s	17.7 s	8.8 s	15.4 s	21.3 s
TOM `C`	0.6 s	0.2 s	1.9 s	2.0 s	2.2 s
TOM `Java`	1.9 s	2.2 s	7.8 s	8.4 s	8.2 s

[6] note that MAUDE is an *interpreter*. The experimental results are extraordinarily good compared to the compiled and highly optimized low-level C implementations

[7] on this example, the performance of ASF+SDF may be explained by the lack of support for builtin integers

[8] `http://gforge.inria.fr/projects/tom/`

7 Related work

The creation of TOM was motivated by the difficulty in integrating or reusing a term rewriting based language in an industrial context. The initial idea of piggybacking rewriting on a generalist host language was inspired by Lex and Yacc and generalized in the framework of formal islands. TOM is the result of a long term effort, on one hand integrating innovative ideas and concepts, and on the other hand combining and incorporating key notions and techniques developed in well reputed research teams. Regarding the data-structures, the implementation of the generator has been done in strong cooperation with the authors of ApiGen [15] and followed our experience with the ATerm [14] library used by ASF+SDF. The originality of our solution is to provide typed constructs resulting in a faster and safer implementation. Moreover, we introduce the notion of *hook* which is strongly related to OCaml *private types*. Concerning matching theories, TOM is similar to ASF+SDF by providing list-matching, that corresponds to associative matching with neutral element. Contrary to MAUDE and ELAN, the restriction to list-matching instead of more complex theories like associative-commutative makes the implementation simpler and powerful enough in many cases. The design of the strategy language has been inspired by ELAN, Stratego, and JJTraveler. Compared to ELAN, TOM does not support implicit non-deterministic strategies, implemented using back-tracking. But due to reification of $t_{|\omega}$, explicit non-deterministic computations are practical. ASF+SDF does not have a strategy language but provides traversal functions that can be used to control how a set of rule should be applied. By raising the notion of strategy to the object level, TOM offers meta-programming capabilities that may remind the meta-level of MAUDE. With regard to strategy languages, Stratego is certainly the language to which TOM is the most close, the main differences being strategies as terms and explicit non-deterministic computations.

8 Conclusion

In this paper we introduced the system TOM, which brings rewriting techniques to the world of mainstream programming languages. In addition to this original result, the contributions of TOM include: the notion of *formal island*; the certification of pattern matching; a support for *private types* in JAVA; an efficient implementation of typed and maximally shared terms; user definable recursive strategies (in JAVA) using the μ operator; strategies considered as terms; reification of $t_{|\omega}$ that makes non-deterministic computations explicit.

We are currently working on two extensions. One is the formalization and the integration of the notion of anti-patterns [9], which enables the expression of negative constraints inside patterns. The second one concerns the integration of termgraph rewriting capabilities into the language and the extension of the strategy library.

References

1. E. Balland, C. Kirchner, and P.-E. Moreau. Formal islands. In M. Johnson and V. Vene, editors, *Proceedings of AMAST 2006*, volume 4019 of *LNCS*, pages 51–65. Springer-Verlag, 2006.
2. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of WRLA 1996*, volume 4 of *ENTCS*. Elsevier Science Publishers, Sept. 1996.
3. 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 WRLA 1998*, volume 15 of *ENTCS*. Elsevier Science Publishers, Sept. 1998.
4. M. Brand, A. Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
5. M. Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In *Compiler Construction*, volume 1575, pages 198–213, 1999.
6. P. Brauner, C. Houtmann, and C. Kirchner. Principles of superdeduction. In *LICS*, Jan. 2007. To appear.
7. H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Proceedings of RTA 2003*, volume 2706 of *LNCS*, pages 76–87. Springer-Verlag, June 2003.
9. C. Kirchner, R. Kopetz, and P. Moreau. Anti-pattern matching. In *Proceedings of the 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 110–124. Springer-Verlag, 2007.
10. C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In P. Barahone and A. Felty, editors, *Proceedings of PPDP 2005*, pages 187–197. ACM Press, July 2005.
11. X. Leroy. The objective caml system release 3.09. (`http://caml.inria.fr/`).
12. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
13. A. Reilles. Canonical abstract syntax trees. In *Proceedings of WRLA 2006*. ENTCS, 2006. To appear.
14. M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.
15. M. van den Brand, P.-E. Moreau, and J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–78, Dec. 2005.
16. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26, NY, USA, 1998. ACM Press.
17. J. Visser. Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN conference on OOPSLA*, pages 270–282, NY, USA, 2001. ACM Press.