

Rewriting Strategies in Java

Emilie Balland, Pierre-Etienne Moreau and Antoine Reilles

*UHP & LORIA, INRIA & LORIA and INPL & LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex France*

Abstract

In any language designed to express transformations, the notion of rewrite rule is a key feature. Its conciseness as well as its strong theoretical foundations are essential. The notion of strategy is complementary: this describes how rules are applied. In this paper, we show how a high-level strategy language can be implemented in a JAVA setting. We present the integration of the visitor combinator design pattern into TOM. This corresponds to an interpreter for strategy expressions. To be more efficient, we present a compilation method based on bytecode specialization. This low-level transformation is expressed in TOM itself, using rules and strategies.

1 Introduction

Rule based transformations are used in a wide range of applications including compiler construction, optimization, refactoring, software renovation as it is shown in [12,7]. Complex program transformations are achieved through a number of elementary modifications, defined by *rewrite rules*, whereas the way they are applied corresponds to the notion of *strategy*.

The main interest of using strategic rewriting systems is the conciseness, as well as the strong theoretical foundations of this paradigm. In practice, elementary program transformations appear explicitly in the program as rules. The strategies defined separately allow the user to precisely control the applications of the transformations. The separation between rules and strategies helps to ensure properties such as correction or termination. In the context of optimizer construction, [13] demonstrates that standard strategies (i.e. innermost or top-down traversals) are not sufficient. Even if separating transformations and control provides safety, languages based on rewriting must offer modular and extensible strategy expressions to be really usable.

There already exist several environments, or meta-environments, well tailored to ease the development of such transformation systems. Among them, let us mention OBJ, MAUDE, ASF+SDF, STRATEGO, ELAN, HATS or TXL for example. Since 2001, we have been developing the TOM system [8,1], whose main originality is to be built on top of an existing language: JAVA. The main advantage of this approach

is that any JAVA program is a TOM program. Therefore, the system automatically provides a large library and support for builtins such as `int`, `float`, and `String`, I/O, system calls, *etc.* As a counter part, the design of the language is more constrained since the introduced constructs should be compatible with the JAVA model and syntax. In this paper, we give in Section 2 an outline of the TOM language. In Section 3, we show how strategic programming, pioneered by OBJ, ELAN, MAUDE, and STRATEGO can be lifted into the JAVA programming environment. In Section 4, we show how it can be applied to implement a compiler for the strategy language of TOM using a rule based library designed to perform JAVA bytecode transformations.

2 Rule based programming in Java

2.1 First order term

The notion of *term* is essential when considering rewrite rules: this is the main data-structure. GOM [9] is a language and a tool for describing typed tree structures. Given an algebraic signature, it generates an implementation efficient both in time and space. The implementation offers maximal sharing which is similar to the ATerm library [10]. As in ApiGen [11], the generated code ensures that only well typed terms can be built. As illustrated below, GOM offers modular importations and supports primitive types of JAVA:

```

module Pico
imports String int
abstract syntax
Expr = Var(name:String)
      | Cst(val:int)
      | Plus(e1:Expr, e2:Expr)
      | Mult(e1:Expr, e2:Expr)

```

A main contribution of GOM is to provide a *hook mechanism* to define canonical forms that have to be preserved. In other words, to each algebraic operator, a construction function can be associated. This function ensures by construction that terms are in normal form, and thus that invariants are preserved. In the following example, we define a hook attached to `Plus`, which ensures that the addition with a neutral element is automatically simplified:

```

Plus:make(e1,e2) {
  %match(e1,e2) {
    Cst(0),x -> { return 'x; }
    x,Cst(0) -> { return 'x; }
  }
}

```

2.2 Rule based programming

TOM is a language extension which adds pattern matching primitives to existing imperative languages. Pattern matching is directly related to the structure of objects

and therefore is a very natural programming language feature, commonly found in functional languages. This is particularly well-suited when describing various transformations of structured entities like for example trees/terms, hierarchical objects and XML documents. The main originality of the TOM system is its language and data-structure independence [8]. From an implementation point of view, it is a compiler which accepts different *host languages* and whose compilation process consists in translating the matching constructs into the underlying native language. Since GOM generates JAVA implementations of terms, in the following, we consider JAVA as the host language.

On the practical side, TOM provides three main features: a “%match” construct to match objects, a “” construct to build objects, and a mapping definition formalism used to connect the algebraic level to the implementation level, called *formal anchor*. In this paper we only consider implementations of data-structures generated by GOM, but note that the notion of formal anchor allows to match against any kind of data-structure.

The %match construct adds pattern matching facilities similar to functional languages. The action part is written in JAVA, providing flexibility. The “” construct builds a term and can be used anywhere a JAVA expression is allowed. For example, the expression ‘Cst(5) builds the corresponding term.

Note that JAVA expressions can be mixed with TOM expressions and that invariants are systematically ensured. Therefore, ‘Plus(Cst(2+3),Cst(2-2)) will be evaluated into Cst(5), after reducing 2+3 and 2-2, using JAVA semantics, and removing the neutral element Cst(0), thanks to the hook mechanism.

In addition, TOM provides associative matching with neutral element (also known as list-matching) that is particularly useful to model the exploration of a search space and to perform list or XML based transformations.

To illustrate the expressiveness of list-matching we consider a variadic operator ExprList whose signature is defined as follows:

```
Expr = ... | ExprList( Expr* )
```

This allows to build list of expressions. Therefore, the term ExprList(Cst(3), Var("x"),Cst(7)) denotes a list composed of three elements. To define the search for a variable Var(name) the following construct can be used:

```
boolean hasVar(Expr l) {
  %match(l) {
    ExprList(a*,Var(name),b*) -> { return true; }
    ExprList(a*,Plus(x,y),b*) -> { return hasVar('x') || hasVar('y'); }
  }
  return false;
}
```

In this example, *list variables* annotated by a * are instantiated by a (possibly empty) list. Therefore, for the considered sequence, the Var(name) will be found in second position, and the JAVA variable name will be instantiated by the string "x". The variables a* and b* are respectively instantiated by ExprList(Cst(3)) and ExprList(Cst(7)). This corresponds to the list which contains Cst(3) and the

list composed of `Cst(7)`.

3 Strategic programming in Java

It is now well established that first order functions or rules are not sufficient to describe transformations in an elegant way: higher-order features are needed to describe how transformation rules should be applied. This is the purpose of *strategies*.

3.1 TOM strategy language

An elementary strategy is a minimal construct that corresponds to a transformation. We distinguish three basic ingredients: `Identity` (does nothing), `Fail` (always fails), and the notion of transformation rule (performs an elementary transformation). In our system, they must be type-preserving and are defined by extending a default strategy:

```
%strategy EvalConst() extends Fail() {
  visit Expr {
    Plus(Cst(c1),Cst(c2)) -> { return 'Cst(c1 + c2); }
    Mult(Cst(c1),Cst(c2)) -> { return 'Cst(c1 * c2); }
  }
}
```

When applied to a node of sort `Expr`, a transformation is performed if a pattern matches the current node. Otherwise, the default strategy is applied (`Fail` in this case). The `EvalConst` strategy describes how to simplify the addition or the product of two constants. Note the use of `c1 + c2`, which is a JAVA expression. For example, it returns `Cst(12)` when applied to `Mult(Cst(3),Cst(4))`.

On top of elementary strategies, more complex strategies can be built, involving basic combinators as presented in [13,14]. By denoting $s[t]$ the application of the strategy s to the term t , the *basic combinators* are defined as follows:

$\text{Sequence}(s_1, s_2)[t]$	$\rightarrow s_2[t']$ if $s_1[t] \rightarrow t'$ failure if $s_1[t]$ fails
$\text{Choice}(s_1, s_2)[t]$	$\rightarrow t'$ if $s_1[t] \rightarrow t'$ $s_2[t']$ if $s_1[t]$ fails
$\text{All}(s)[f(t_1, \dots, t_n)]$	$\rightarrow f(t_1', \dots, t_n')$ if for all i , $s[t_i] \rightarrow t_i'$ failure if there exists i such that $s[t_i]$ fails
$\text{One}(s)[f(t_1, \dots, t_n)]$	$\rightarrow f(t_1, \dots, t_i', \dots, t_n)$ if $s[t_i] \rightarrow t_i'$ failure if for all i , $s[t_i]$ fails
$\text{Omega}(s, j)[f(t_1, \dots, t_n)]$	$\rightarrow f(t_1, \dots, t_j', \dots, t_n)$ if $s[t_j] \rightarrow t_j'$ failure if $j > n$ or if $s[t_j]$ fails

Using this formalism, a strategy trying to apply the user strategy `EvalConst()` and performing identity if it fails can be built with `Choice(EvalConst(),Identity())`. In the presented implementation, it can be applied to a term `t` using the `.visit(t)` method.

By combining elementary strategies and basic combinators, it becomes possible to define higher-level constructs. The fix-point operator can be defined as `Repeat(s) = Choice(Sequence(s,Repeat(s)),Identity())`.

This strategy will apply the strategy `s` repeatedly until it fails, and then return the last result obtained before failure, thus `Repeat` will never fail.

To write recursive strategy definitions such as `Repeat(s)`, we use a recursion operator μ , which is similar to the `rec` of functional languages. Then, `Repeat(s)` is defined by `Repeat(s) $\hat{=}$ μx .Choice(Sequence(s,x),Identity())`, where `x` denotes a variable, and `s` a parameter of the strategy. Defining such strategies in JAVA is not easy because a graph has to be built to encode the recursion. The `x` should be seen as a pointer to the strategy itself. In `JJTraveler` for example, these recursive strategies can be defined, but the graph corresponding to the recursion should be built *by hand*. In `TOM`, we raised the recursion operator to the object level, using `mu` and variables, allowing the definition of recursive strategies in a term based framework:

```
Strategy Repeat(Strategy s) {
    return 'mu(MuVar("x"),Choice(Sequence(s,MuVar("x")),Identity()));
}
```

As a consequence, the expansion phase which transforms a term into a graph (called μ -expansion) is performed at runtime. Strategy expressions can have any kind of parameters. It is common to have a `JAVA Collection` as parameter, for instance to collect all variable definitions in an AST. Using a `Stack` is also common to find free variables, or to bind variables to their declaration (as in the μ -expansion, defined by a strategy of course!). As illustrated above, parameters can also be strategies (`s` for instance). This allows user defined strategies to behave differently depending on how they are applied, or to build continuations in strategies.

Finally, when a signature is defined in the GOM language, congruence and construction strategy operators are generated. Those are used to discriminate constructors and thus allow to describe higher-order strategies such as *map* for instance. Given a signature `List = Cons(head:Element,tail:List) | Empty()`, the *map* strategy can be defined by `'mu(MuVar("x"),Choice(_Cons(s,MuVar("x")),_Empty()))`, where `_Cons` and `_Empty` denote congruence operators associated to `Cons` and `Empty`. `_Cons(s1,s2)` applies `s1` and `s2` to the arguments of a term rooted by `Cons`, otherwise it fails. Using construction strategies, this naturally allows the definition of dynamic rules. Also, the use of those congruence strategies permits the definition of local strategies, where the order of evaluation of the subterms is defined by the user. By defining two parameterized strategies `Get` and `Set` (which manage a `JAVA HashMap`), we can easily implement the rewrite rule $f(x) \rightarrow g(x)$ by `'Sequence(_f(Set("x")),Make_g(Get("x")))`.

The combination of basic strategy combinator and JAVA has been pioneered by JJTraveler. In the presented work, the main contributions are the possibility to define elementary transformation rules using `%strategy`, the introduction and the implementation of the explicit recursion operator `mu`, as well as the automatic generation of congruence operators. This is important since it makes the framework easier to use and contributes to the promotion of formal methods in the JAVA community.

Some elements of reflexion

A shortcoming of rewrite rules is their context free nature: rules have only access to the subterm they are applied to. Using parameterized transformation rules it is of course possible to propagate context information, but this may not be convenient, for example to know to which redex a rule is applied, when traversal operators are involved. We have added the ability for each strategy, including transformation rules, to know where they are applied wrt. the original term. These positions are represented by first order objects corresponding to a sequence of integers and can be used to build strategies such as “*replace at position ω* ” or “*get subterm at position ω* ” by nesting `Omega` strategies, where `Omega(s,i)` applies `s` to the i^{th} subterm. When dealing with transformation systems which can produce several results, the combination of traversal operators and positions allows to compute all possible reachable terms in an elegant way. This extension is very useful to describe tools and analyzers that deal with reachability or control flow analysis problems.

A main originality of TOM is the ability to perform matching against any kind of term implementation, and in particular against the objects that implement a strategy. Therefore, a strategy expression, including the recursion operator, are first class objects. TOM can be used to match, transform, or dynamically create any strategy expression. A strategy is also “traversable”, in such a way that rules and strategies can be applied on a strategy itself. While quite new for us, this offers reflective capabilities that can be used to perform optimization or on the fly compilation of dynamically created strategies, as illustrated in Section 4. Along with congruence strategies and construction strategies, this allows to encode pattern matching and rewrite rules as dynamic strategies, thus providing dynamic scoped rules. Note that the presented language does not allow to construct higher-order strategies as in the HATS strategy language [15]. In particular, it is not possible to construct dynamic instantiations of rules.

3.2 Implementation in JAVA

The implementation of such strategies relies on the use of several design patterns, the most important being the *visitor combinators* [14] one (do not confuse with the “visitor design pattern”). The intention of this pattern is similar to the *visitor* one, but allows combination of visitors, and the definition of various tree traversal strategies by combination of basic visitors. To use those strategies, the term structure has first to be `Visitable`, that is to exhibit the term structure via the `Visitable` interface:

```
public interface Visitable {
```

```

public abstract int getChildCount();
public abstract Visitable getChildAt(int i);
public abstract Visitable setChildAt(int i, Visitable child);
}

```

This interface allows to decompose a term and to explore its subterms, and is used by the elementary strategies `All` and `One` to describe the tree traversal. Strategies themselves have to implement this interface, to be manipulated as terms and transformed by strategies as described in the previous subsection.

Strategies are part of the visitor combinators design patterns, and so are the combined visitors. Thus strategies implement the `Visitor` interface, providing a `Visitable visit(Visitable v)` method, used to describe the behavior of a strategy when applied to a given term `v`. Those strategies may have strategies as children, for instance `Sequence(s1,s2)` will be represented by a `Visitor` object of class `Sequence` with two fields `first` and `then` which are strategies. In that case, the `visit` method is implemented as follows:

```

Visitable visit(Visitable v) throws VisitFailure {
    return then.visit(first.visit(v));
}

```

It applies the `first` strategy to the subject term `v`, then apply the strategy `then` to the result. If one of those application fails, the whole sequence fails. In our framework, the notion of failure is encoded by a JAVA exception. This is why the `visit` method may throw a `VisitFailure` exception. The `All` and `One` combinators use the `Visitable` interface to apply their child strategy to the subterms of the subject, and build the result using the `setChildAt` method. To ease the strategy compilation, in our implementation the visitor children are not stored as object fields like in the previous example, but rather in a visitor array named `visitors`, to ease accessing them with generic algorithms.

In order to create type safe user strategies, we introduce a combinator `Fwd` which depends on the typed interface of the tree structure. The goal of this combinator is to dispatch the call to the generic `visit` to type specific transformation methods. In order to define new transformation strategies, the user has to extend this `Fwd` class and redefine the typed variants of the `visit` method, thus ensuring type preserving transformations.

The `Fwd` combinator possesses one child, which is the default strategy, and is designed to be extended by inheritance. Its `visit` method follows the `visitor` design pattern [5], by calling an `accept` method on the visited tree node, which is used to select a type specific `visit` method, by the classical double dispatch method. For instance, each node of type `Expr` in the tree representation generated for the `Pico` module presented in Section 2 possesses an `accept` method:

```

public PicoAbstract accept(PicoVisitor v) throws VisitFailure {
    return v.visit_Expr(this);
}

```

where `PicoVisitor` is the interface that provides typed visit methods.

The `visit` method of the `Fwd` combinator will call the `accept` method when applied to a term belonging to the `Pico` module (*i.e.* instance of the `PicoAbstract`), and falls back to the default strategy otherwise. Each specific type-preserving `visit` method is implemented as a call to this default strategy.

```
public class Fwd implements PicoVisitor {
    protected Visitor defaultStrategy;
    public Visitable visit(Visitable v) throws VisitFailure {
        if(v instanceof PicoAbstract) {
            return ((PicoAbstract) v).accept(this);
        } else {
            return defaultStrategy.visit(v);
        }
    }
    public Expr visit_Expr(Expr arg) throws VisitFailure {
        return (Expr) defaultStrategy.visit(arg);
    }
}
```

Then, each user defined strategy will only have to extend this `Fwd` class and override one or many type specific `visit` method to implement a type safe transformation, that can be combined with the elementary strategies to form a complex transformation. The `EvalConst` strategy defined in section 3.1 can then be defined as the following class:

```
public class EvalConst extends Fwd {
    public EvalConst() { super('Fail()); }
    public Expr visit_Expr(Expr arg) throws VisitFailure {
        %match(arg) {
            Plus(Cst(c1),Cst(c2)) -> { return 'Cst(c1 + c2); }
            Mult(Cst(c1),Cst(c2)) -> { return 'Cst(c1 * c2); }
        }
    }
}
```

This framework may seem difficult to use from a user point of view, but as the typed tree structure is generated by the GOM compiler, the framework is also instantiated for this typed structure, and the only requirement for the user is to extend the `Fwd` class to create its transformation strategies. In the context of TOM, the implementation of user defined strategies is straightforward when using the `%strategy` construct since the JAVA code is automatically generated.

4 Optimizing strategies by Bytecode transformation

As mentioned in section 3, TOM strategies are represented by a tree composed of user defined strategies basic combinators. As a consequence, high level strategies, such as leftmost-innermost, are expressed by a combination of low level primitives. This approach has many advantages and makes the strategy language extensible. A

counter-part of this modularity is the cost of complex traversals due to the visitor design pattern. The time spent in evaluating each elementary strategy is not negligible in comparison with the time spent in effectively performing the transformations. This has led us to study whether a strategy expression could be efficiently compiled or not, in order to improve the overall efficiency of a strategy based system.

To achieve this goal, two different approaches can be considered:

- (i) a high-level transformation which consists in simplifying a strategy expression into a more efficient, but semantically equivalent one,
- (ii) a low-level transformation directly applied on the bytecode, consisting in optimizing the implementation via program transformation.

The first approach consists in transforming a strategy into a more efficient one. For that, we can consider equivalence classes of strategies and choose as normalized form the most efficient representation. As TOM strategies are first-class citizens, they can be transformed using rewrite rules. Therefore, this first approach can be described in TOM. Below, we give two examples of simplification rules:

```
Sequence(Identity(), x) -> { return 'x; }
Choice(Fail(), x)      -> { return 'x; }
Not(One(Not(x)))       -> { return 'All(x); }
Not(All(Not(x)))       -> { return 'One(x); }
```

One advantage of this method is the simplicity of its implementation, but to obtain a real gain, the detection of more complicated patterns is needed.

On another side, by instrumenting the strategy library we realized that most of the time is spent in executing function calls. This shortcoming is due to the visitor combinator design pattern and only appears in the implementation code (i.e. JAVA). As a result, this deficiency can only be reduced at a low-level. As TOM is a language built on top of JAVA, this optimization can be achieved by statically analyzing JAVA sources. However, this solution is not acceptable because it would impose a second transformation step, just after the compilation from TOM to JAVA.

For these reasons, our approach consists in performing the optimization at runtime, directly on bytecode programs. Furthermore, the TOM formalism is particularly well suited to describe such kind of transformation. In the following, we show how a compiled JAVA class can be analyzed and transformed using TOM, and we give some experimental results to express optimizations on strategy code.

4.1 Bytecode transformation by rewriting

A particularity of JAVA is to target a standard, machine independent, low level bytecode as compiled form of programs, which is interpreted by the JAVA virtual machine. Then, it becomes natural to perform program transformations at the bytecode level. This technique is used in particular to provide language extensions such as aspect oriented programming, to perform sophisticated static analysis, generate middleware code or improve runtime performances.

To manipulate JAVA classes, TOM provides a library [2] which supplies a term usable by TOM out of a JAVA class. The library enables to define transformations of this term by strategic rewriting as well as functionalities to generate a new JAVA

class from the modified term.

The library generates a GOM term using the ASM library [4]. This term is a memory-efficient representation of the JAVA class, which can then be traversed and transformed using TOM. Figure 1 shows the steps of bytecode transformation in this framework. After translating the JAVA class into a GOM term, we use TOM features to define transformations and traversals and to obtain a new GOM term which can be transformed into a new JAVA class.

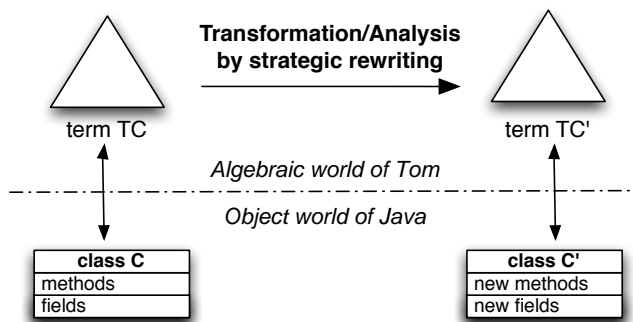


Fig. 1. Bytecode transformations in TOM

As we have seen in the previous section, one interesting feature of TOM is its ability to perform list matching. Every JAVA method is represented in GOM by a list of instructions. So finding if a `Iload(i)` is followed by a `Istore(i)` can be expressed by the pattern: `InstructionList(*, Iload(i), Istore(i), *)`.

With one pattern, we can find all the occurrences of the sequence `Iload(i); Istore(i)` in the body of a method. Moreover, the strategy language can be used to express complex transformations in a very concise way. Due to this library, we are able to express optimizations on strategy code directly using TOM.

4.2 Optimizing strategies

We have seen previously that the main problem of strategy efficiency was due to the implementation by the visitor combinator design pattern. Every strategy combinator is parameterized by strategies and the visit method of a combinator calls the visit methods of the parameters. When considering a large strategy term, most of the time is dedicated to visit calls.

Given a strategy expression, the objective of the strategy compiler is to generate a specialized version of the `visit` method, by successively inlining strategy combinators contained in it. At the end, the resulting code no longer depends on calls to other `visit` methods of strategy combinators. Bytecode manipulation, along with a Just-in-time tool, is well suited to achieve this objective as strategies can be visited during the execution of a TOM program. Indeed, it is possible to determine at runtime the type of each node of the tree, allowing us to proceed to the compilation. Figure 2 shows the strategy tree of a Top-Down. Every combinator has its own `visit` method. Each of them, except for the user defined strategy, contains at least one call to another `visit` method.

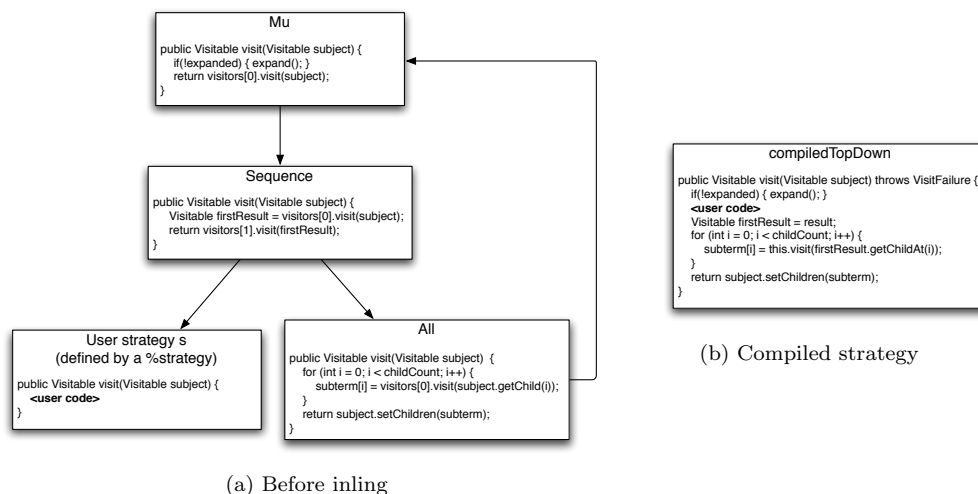


Fig. 2. Inlining of the Top-Down strategy: the resulting code is clearly more efficient. Note the recursion encoded by `this.visit`

In our implementation, every combinator can access its parameters: they are stored in an array `visitors[]`. For example, in the `Sequence` class, `visitors[0]` corresponds to the first strategy to apply, and `visitors[1]` to the second one. When considering the statement `return visitors[1].visit(firstResult)`, the strategy compiler has to detect the call to `visit` in order to inline its body. This is performed at runtime using reflective capabilities of JAVA: the dynamic type of `visitors[1]` is determined and the implementation of the corresponding `visit` is inlined. In our example, the considered dynamic type is `All` and the call to `visit` can be replaced by the loop over children. This process is repeated until getting a fix-point which corresponds to the compiled strategy.

In our framework, there is a mapping from the real bytecode (extracted using ASM) to an algebraic view (implemented by GOM). Therefore, a JAVA class is represented by a term. For each event generated by ASM corresponds an algebraic constructor. Due to lack of space, we cannot explain in detail how a JAVA class is described (ASM generates more than 250 different kinds of events). In the following we consider that a JAVA class is characterized by a list of methods. Each of them has a name, a profile, as well as an associated list of instructions. This abstract view can be described using an algebraic signature:

```

TMethodDef = Method(info:TMethodInfo, code:TMethodCode)
TMethodInfo = Info(name:String, desc:TMethodProfile)
TMethodCode = Code( TInstruction* )
  
```

In order to inline a call to `visit`, the first step consists of finding the calls to this method. Given a strategy (i.e. a graph of objects), we consider the algebraic term that represents the list of classes that occur in the graph of objects. Starting from the root, we have to find the implementation of the `visit` method. This is done by inspecting the list of methods associated to a given class.

```

%match(mList) {
  MethodList(*,Method(MethodInfo[name="visit"],code),*) -> {
  
```

```

    /* processing where we can use the code variable */
  }
}

```

In this example, the associativity of `MethodList` is used to search for a method whose name is `visit`. When found, the body of this method is retrieved and stored in the `code` variable. Note also that syntactic matching is a concise way to retrieve information in deep subtrees (the slot `name` for example).

Now that the term corresponding to the method's body has been retrieved, the inlining operation can really start. This second step consists in determining, for each call to `visit`, the type of the callee and to collect its implementation code. The type of each sub-strategy is determined at runtime using JAVA reflexivity. The corresponding code is mapped to a GOM term and inlined: this consists in replacing the call to `visit` by the considered term.

```

%strategy InlineVisitCode() extends Identity() {
  visit TMethodCode {
    /* Match the bytecode of visitors[index].visit(_)'*/
    Code( Aload(0),Getfield[name="visitors"],indexInst,Aaload[],
        Invokeinterface[name="visit"],tail*) -> {
      /* find corresponding instructions */
      newCode = ...
      return 'Code(newCode, tail*);
    }
  }
}

```

In this example the `visitors[index].visit(_)` pattern is detected by searching for five consecutive bytecode instructions: `Aload(0)` puts 0 on the stack in order to retrieve the class variable `visitors` (using `Getfield`). The variable `indexInst` can be instantiated by any bytecode instruction. This is useful since there exists different ways to put the `index` on the stack. `Aaload` corresponds to the access to `visitors[index]`. Finally, `Invokeinterface` invokes the method `visit`. The statement `newCode = ...` performs the runtime type detection, loads the class, and store the term representation into `newCode`. The inlining is performed by returning `'Code(...)`. This replaces the five matched instructions by `newCode`.

To be correct, note that we have to rename each label and index before inlining `visit` calls. Otherwise, name capture problems may occur. Similar to the inlining process, the renaming process is implemented by a user defined strategy named `Rename`.

The combination of renaming and inlining is defined by a strategy, using a `TopDown` operator to perform a traversal of the tree:

```

MuStrategy inliner =
  'Sequence(
    TopDown(Rename()),
    TopDown(InlineVisitCode())
  );

```

```
newCode = (TMethodCode)inliner.visit('code');
```

Here, `code` is the variable used in the pattern defined above to find the definition of the `visit` method. This strategy of optimization can be called on the initial strategy term before visiting.

To obtain this new code, there is a `compile` method that generates, from an original strategy term, a class that corresponds to the compiled strategy term and returns an instance of it. Instead of generating a file for this new class, it is directly loaded into the JVM.

4.3 Benchmarks

We present in Figure 3 a few benchmarks obtained for different applications, when using the just-in-time strategy compiler. Concerning space usage, code inlining leads to duplication, but in a linear way. In practice, strategies are not so huge and contains less than 100 elementary combinators. Therefore the space increase is not an issue.

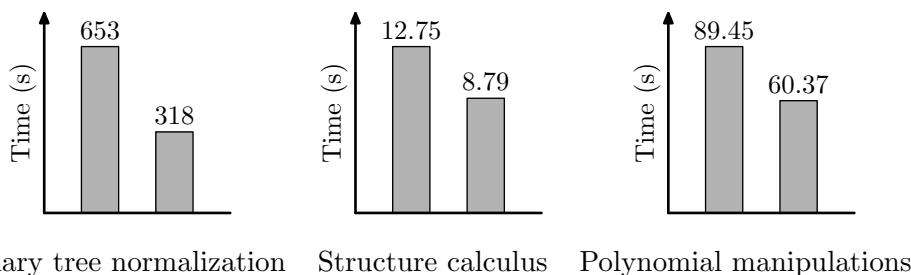


Fig. 3. Experimental results: programs are run without and with strategy optimisation. The execution time is expressed in seconds (shorter is better).

The first example corresponds to the innermost normalization of a binary tree whose size ranges from 2^2 to 2^{32} nodes. The strategies involved are extending failure. Therefore, each time a rule cannot be applied to a subterm, a new exception is thrown and caught by a `catch` statement. We obtain a gain by a factor 2 when using the compiled strategy. This can be explained by the fact that the strategy code is inlined, and thus the exception mechanism is kept local to a method instead of being distributed across methods and objects. This makes the implementation much more efficient.

The two following examples are real life applications which use strategies. The structure example is a theorem prover in the calculus of structures, as presented in [6]. This system features *deep inference*, which means it is necessary to traverse a proof term in depth to find new redexes where the deduction rules can be applied. The last application is a tool used to find quasi-interpretations of functional programs [3]. This involves manipulating of polynomial expressions, traversing them to apply transformations and substitutions, replacing variables by values for evaluation, usually in a bottom-up way.

5 Conclusion

TOM brings pattern matching and rewriting to JAVA, allowing to express transformations in a clean and efficient way. In order to describe complex transformations as found in compilers, optimizers or program analysis, we introduced a strategy language integrated into JAVA allowing to control precisely the rules applications and to describe in a formal way tree traversals and transformations.

Those strategies are built by composing objects representing elementary strategies, which can then be viewed as an interpreter of strategies, using the visitor combinator design pattern.

Among the transformations we can describe with TOM, we focussed here on low-level program transformations, for example JAVA bytecode. In particular, we illustrate those bytecode transformations by the just-in-time compilation of strategy expressions, using partial evaluation techniques. To express this transformation, we presented a TOM library built on top of ASM and allowing to transform bytecode expressions in an algebraic manner. Due to the reflexive properties of our strategy library, we use strategic programming to traverse the bytecode representation of the strategy expressions, perform analysis and inlining.

In this paper, we have presented rules for method inlining on strategy codes. A next step should be to add new rules for implementing classical loop-invariant code motion. Due to the recursive operator, the strategy code can contain imbricated loops and it would be interesting to remove the computations that can be performed outside.

Acknowledgments: We sincerely thank Jeremie Delaitre for the initial development of bytecode rewriting in TOM and its work on strategy compilation.

References

- [1] Balland, E., P. Brauner, R. Kopetz, P.-E. Moreau and A. Reilles, *Tom: Piggybacking rewriting on java*, in: *Proceedings of the 18th Conference on Rewriting Techniques and Applications (RTA'07)*, 2007.
- [2] Balland, E., P.-E. Moreau and A. Reilles, *Bytecode rewriting in tom*, in: *Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation - BYTECODE 07*, 2007.
- [3] Bonfante, G., J.-Y. Marion and R. Péchoux, *A characterization of alternating log time by first order functional programs.*, in: *LPAR*, 2006, pp. 90–104.
- [4] Bruneton, É., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.
- [5] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, Lecture Notes in Computer Science **707** (1993), pp. 406–431.
- [6] Kahramanoğulları, O., P.-E. Moreau and A. Reilles, *Implementing deep inference in TOM*, in: P. Bruscoli, F. Lamarche and C. Stewart, editors, *Structures and Deduction* (2005), pp. 158–172, ISSN 1430-211X.
- [7] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *CC '01: Proceedings of the 10th International Conference on Compiler Construction* (2001), pp. 52–68.
- [8] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, LNCS **2622** (2003), pp. 61–76.
- [9] Reilles, A., *Canonical abstract syntax trees*, in: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications* (2006), to appear.

- [10] van den Brand, M., H. de Jong, P. Klint and P. Olivier, *Efficient annotated terms*, Software, Practice and Experience **30** (2000), pp. 259–291.
- [11] van den Brand, M., P.-E. Moreau and J. Vinju, *A generator of efficient strongly typed abstract syntax trees in Java*, IEE Proceedings - Software Engineering **152** (2005), pp. 70–78.
- [12] Visser, E., *A survey of rewriting strategies in program transformation systems*, in: B. Gramlich and S. L. Alba, editors, *Proceedings of the Workshop on Reduction Strategies in Rewriting and Programming (WRS '01)* (2001).
- [13] Visser, E., Z.-e.-A. Benaïssa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *Proceedings of the third ACM SIGPLAN international conference on Functional programming (ICFP '98)* (1998), pp. 13–26.
- [14] Visser, J., *Visitor combination and traversal control*, in: *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA '01)* (2001), pp. 270–282.
- [15] Winter, V. L., *Strategy Construction in the Higher-Order Framework of TL*, Electronic Notes in Theoretical Computer Science (ENTCS) **124** (2005), pp. 149–170.