Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 204 (2008) 71–82
www.elsevier.com/locate/entcs

Rules and Strategies in Java

Pierre-Etienne Moreau and Antoine Reilles

INRIA & INPL & Loria
(moreau|reilles)@loria.fr

Abstract

ELAN was one of the first rule based language to introduce a clear separation between the notion of *rule* and the notion of control, also called *strategy*. Starting from this experience, our goal was to make available these constructs in a widely used language such as JAVA. In this paper we present the essential features we have considered when designing the TOM language, which is an extension of JAVA that integrates the notions of rules and strategies. Relying on the implementation, we explain how these ingredients can be implemented and smoothly integrated in a JAVA environment.

Keywords: rule, strategy, control, pattern-matching, programming

1 Introduction

The notion of rewrite rule is an abstraction that can be used to model various processes. It has been used intensively to model, study, and analyze different parts of a complex system, from algorithms to running software. On one side it can be used to describe the behavior of a transition system for instance. On the other side, it provides a theoretical framework useful to certify and prove properties such as termination or confluence.

Besides its straightforward interpretation, the notion of rewrite rule can be used to produce efficient implementations. It has been successfully used in theorem provers and proof assistants such as RRL, Otter, C*i*ME, Coq, as well as in the kernel execution mechanism of rule based and functional languages such as ASF+SDF [7], Clean, Caml, ELAN [4], MAUDE [3], and Stratego [10] for instance.

Programming with rewrite rules is apparently easy: a complex transformation can be decomposed into elementary transformations, encoded using a rewrite rule, then, we rely on the rule engine to fire a rule whenever it is possible. Most of the time, we are interested in getting a result whose computation is deterministic. In other words, the result should be reproducible, the set of rules should be terminating and confluent. However, things are rarely confluent by nature. One solution could be to use the Knuth-Bendix completion, but this is not realistic on large programs.

doi:10.1016/j.entcs.2008.03.054

In practice, starting from an initial signature and a simple set of rewrite rules, the programmer often modifies the signature and the rules in order to encode some control. From a software engineering point of view, this annihilates the elegance of term rewriting and makes the system much more complex and difficult to maintain.

A first solution to this problem is to assign a priority to each rule and to consider an execution mechanism that encodes a fixed order of reduction, such as *innermost* or *outermost*, also called *call by value* or *call by name* in functional programming languages. Another solution is to separate the control from the rules. Instead of encoding the control into the rules themselves, it is described in a distinct expression or language. The expressions that specifies how the rules should be applied are called *strategies*. The design of such a strategy language is not easy and several attempts and proposals have been made.

OBJ is one of the first languages that introduced an explicit form of strategy, called *evaluation strategy*. To each operator a list of integers can be attached to specify in which order the arguments should be evaluated.

MAUDE followed this approach and added the notion of *meta-level*. In this setting, a rewrite rule has a name, considered as a constant, and can be explicitly applied via the `meta-apply` operator. The application of a set of rules can be controlled by another program, expressed by rewriting and using `meta-apply`. This new program can also be controlled by another program from the meta-meta-level. This tower of reflexivity is very elegant and expressive, but a bit difficult to use.

ELAN has its origins in OBJ, but followed another approach. Instead of having a meta-level, ELAN was the first language to introduce an explicit *strategy language* to control the application of rules. Each rule has a name which corresponds to an elementary strategy. A strategy can then be combined with another one using operators such as `;` (sequence), `repeat`, `dont-care`, and `dont-know` for instance. This strategy language was both very expressive and easy to use.

Stratego has been inspired by ELAN, MAUDE, and the functional programming style. It introduces a quite elegant and simple strategy language. Similarly to ELAN, a rule is an elementary strategy that can be combined with strategy operators such as `;` (sequence), `<+` (left-choice), *etc.* The main contribution comes from the introduction of a recursion operator and two generic congruence operators `All` and `One`, that can be used to describe higher-level strategies such as *top-down*, *innermost* or *outermost*. In this setting, we have $TopDown(s) = \mu x.s\ ;\ All(x)$, which applies s to a term t, and then recursively applies the $TopDown(s)$ strategy to the immediate subterms of t.

ASF+SDF has also a strategy language, in the same spirit as the OBJ's one. To each operator an annotation can be attached, that specifies its behavior. The combination of `traversal` and `bottom-up` indicates that a given set of rules should be applied in a bottom-up way for example. This approach is of course less general than the previous ones, but it is an interesting trade-of between expressiveness and simplicity to use.

During the last decade we have accumulated an important experience in both implementing and using rule based languages. In this paper we try to isolate the essential constructs and features that have to be considered when designing a new rule and strategy based language. In a second part, we explain how those features can be smoothly integrated and efficiently implemented into an object oriented programming language such as JAVA.

2 Our wish list

Starting from the ELAN experience, we have tried to design a new language based on the same concepts. This process leads us to analyse what were the good points, and what were the points that could be improved. The ELAN language clearly has many interesting constructs, in particular the notion of rules, strategies, and equational matching. Unfortunately, its cohabitation with largely used programming languages such as C or JAVA needs some improvement. In particular, it could be very interesting to use the constructs provided by ELAN in those programming languages themselves. Starting from that situation, our goal was to design a new language, called TOM, with a comparable expressiveness, but in a more accessible programming environment. In this section we present what are the requirements and the essential features that have been considered when designing the TOM programming language.

Terms

A first requirement was to be able to describe and manipulate tree shaped structures, or terms. The ELAN experience showed that it is more convenient to manipulate many sorted terms, especially since this simple typing of terms does help catching many programming errors. Languages such as ASF+SDF, ELAN or MAUDE do mix the term algebra used to express the rules and the concrete input syntax of the specifications. This feature is very convenient to prototype transformations, but makes the implementation of the language much more complex. In order to keep the language and the implementation simple, we wanted to keep the syntax of rules in a prefix notation, like in many functional programming languages.

Describing elementary transformations.

When designing complex applications, it is important to be able to decompose the various transformations into different rewrite systems or elementary transformations, and then decide which rule apply to which term. A first step towards this goal is to let the user define labeled rules:

$$[\ell]\ l \to r$$

A same label can be given to several rules to define a rewrite system. The application of such system has to be explicitly specified by the user. Given a term t, the application of a labeled rule performs a single step of reduction at the root position, if t is matched by a left-hand side. Otherwise, the application fails.

Computing canonical forms.

Another important feature inherited from ELAN is the ability to automatically maintain terms in normal form with respect to a rewrite system. This is done via the definition of unlabeled rules which are applied until getting an irreducible term. The considered rewrite system has of course to be confluent and terminating.

Using unlabeled rules, it is possible to specify a canonical form for the term data structures of the application, such as representing logical formulas in disjunctive normal form, or enforcing constant expression evaluation in a programming language. For instance, the following rule set can be used to express how to build boolean formulas in disjunctive normal form:

```
Not(And(l,r))  -> Or(Not(l),Not(r))
Not(Or(l,r))   -> And(Not(l),Not(r))
Not(True())    -> False()
Not(False())   -> True()
Not(Not(x))    -> x
And(x,Or(y,z)) -> Or(And(x,y),And(x,z))
And(Or(y,z),x) -> Or(And(y,x),And(z,x))
```

Maintaining a data structure in canonical form is an essential feature that improves the quality of software. The programmer does no longer have to take care of this maintenance by calling normalisation functions. In addition, this makes the writing of elementary transformations simpler since only canonical forms have to be considered.

Controlling the rewriting.

In our case, the situation is a bit particular since in addition to the notion of rule, there is an underlying Turing complete language, namely JAVA. To control the application of rules, an attracting possibility would be to use JAVA directly. This language natively offers the composition (`;`), the repetition (`while`), and many other control statements. However, the more we use JAVA, the more it becomes complex to reason about programs and to perform proofs. How to show that a rewrite system is terminating under a given strategy when this strategy is expressed in JAVA? Therefore, the problem is to find a good strategy language which is both expressive and simple to use.

As mentioned in the introduction, we have studied and experimented the expressiveness of several existing strategy languages. The one proposed by Stratego [10] has several interesting properties. It is atomic, being composed of less than 10 elementary combinators. It is expressive, allowing the definition of various traversal strategies. Initially not tailored to support non-deterministic searches, we will see that this limitation can be removed in a simple and elegant way, using context information.

We thus decided to base our strategy language on elementary combinators, that are combined with labeled rules to build more complex strategies. Given a set of

rules, there are two fundamental operators for combining rules: the choice operator $Choice(s_1, s_2)$ which applies s_2 only if the application of s_1 fails. The sequence operator $Sequence(s_1, s_2)$ which applies s_1, and then, if that succeeds, applies s_2. The different combinators are described in Figure 1.

Combinator	Semantics
`Identity()`	Does nothing, and returns the original term
`Fail()`	Always fails
`All(`v`)`	Applies v to all direct subterms in sequence
`Choice(`v_1`,`v_2`)`	Applies v_1, then v_2 if v_1 failed
`Sequence(`v_1`,`v_2`)`	Applies v_1, then v_2. Fails if one of them fails
`Not(`v`)`	Fails if v is applicable, returns identity otherwise
`Omega(`i`,`v`)`	Applies v to the i-th sub-term if it exists, fails otherwise
`One(`v`)`	Applies v to all sub-terms in order, until a success
`IfThenElse(`c`,`v_1`,`v_2`)`	Applies c, then if c success, applies v_1, otherwise v_2

Fig. 1. Elementary strategy combinators

Recursive strategy definitions are essential to describe the common rewriting strategies such as *top-down*, *bottom-up* and *leftmost-innermost*. For instance, *bottom-up* is defined as $BottomUp(v) = Sequence(All(BottomUp(v)), v)$

The basic strategy combinators and the strategies that are created by composing them are generic, and will perform equally on any data structure, when the labeled rules are specific to the data structure (since they perform pattern matching and term construction). In order to build elegantly some transformations, we need data structure dependent strategy combinators, such as congruence strategies. They are used to decompose terms and to apply strategies to subterms. For example, the congruence strategy for the `And` constructor, noted `_And` has two arguments, which are strategies to be applied respectively to the left and right subterms of an `And`. `_And(s1,s2)` applied to the term `And(x,y)` will apply `s1` to `x` and `s2` to `y`, and will fail if applied to a term that is not rooted by `And`.

Knowing the context.

Usually, a rule application, even under strategies, is context free. When a rule or a strategy is applied, the result only depends on the term to which the strategy is applied. However, it is common to require some knowledge about the context when applying a strategy. For example, we may want to know if a particular subterm is in positive or negative position in a formula.

For that, context information such as the list of terms that were traversed by the strategy before accessing the current subterm, or the position of the current

subterm in the traversed term should be accessible when evaluating the right hand side of a labeled rule. The explicit representation of the notion of position also corresponds to classical operations found in the literature, such as the access to a given subterm ($t_{|\omega}$) or the replacement ($t[u]_\omega$) for instance.

Exploring a search space.

Many applications of rule based systems do require the manipulation of non deterministic rewrite systems. For instance, cryptographic protocol verification can be treated as a reachability problem in a particular non deterministic rewrite system that models the protocol. We thus require the strategy language to be able to support non determinism, by letting the user compute the set of successors of the application of a set of rules, and also the successors of the application of rules to different redexes. In ELAN, only the first case can be easily implemented.

Such a control should be flexible enough to support the exploration of a search space in a *depth-first* or *breadth-first* manner.

Integration into JAVA.

Considered separately, each feature mentioned previously is not a revolution and already appears in a form or another in an existing rule based language, except the explicit representation of position information which is a real contribution. The main difficulty of our approach is to *integrate everything into* JAVA. This is also the main contribution of this work.

3 Our approach

As presented in [5,1], the main component of our system is the TOM compiler. It mainly introduces two constructs: a `‘<term>` to build an algebraic term, and a `%match` construct, similar to `switch/case` which executes an action (a JAVA statement) when a pattern matches a given subject. This system does not impose a specific term data structure: the implementation of the term data structure becomes a parameter of the TOM compiler, using a *mapping* mechanism. An advantage of this approach is to make the `%match` construct usable on any kind of data structure. The counterpart is that the user has to provide an implementation.

Terms

To help the programmer, we have designed a tool called GOM [6] that takes an algebraic signature as input and generates a JAVA implementation of this signature. This work is an extension of [8]. The interaction between TOM and GOM is illustrated in Figure 2. In this formalism, a specification for boolean expressions can be the following:

```
%gom {
  module Bool
```

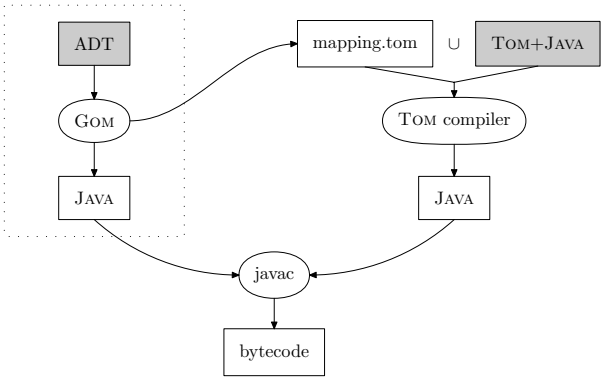


Fig. 2. Given a signature (ADT), a Java implementation is generated. In addition, a *mapping* that makes the implementation a parameter of Tom is generated. The algebraic data structure can be directly manipulated in a Tom program. The compilation process translates the data structure and the rule based program into Java classes, that can be combined with other libraries.

```
  imports String
  abstract syntax
  Bool = True()
       | False()
       | Not(b:Bool)
       | And(l:Bool,r:Bool)
       | Or(l:Bool,r:Bool)
       | Var(name:String)
}
```

The language provides modularity with the use of the `imports` keyword. Java classes implementing this structure are then generated, providing a typed interface as well as an efficient implementation based on maximal sharing. Following the *factory* design pattern, these classes provide static construction functions.

Computing canonical forms.

The implementation of canonization functions can be done by encoding a rewrite system into functions: to each defined symbol a function is associated. These functions are called when a term has to be built, and the corresponding normal form is returned. The main drawback of this approach is that the user (a Java programmer) can *forget* to call these normalisation functions and directly use the factory generated by Gom instead. This would result in terms which are no longer in normal form. To avoid this problem, we consider that the notion of unlabeled rule is strongly tied to the term data structure that is used in the application. We thus propose to define the set of unlabeled rules in conjunction to the definition of the data structure, in the Gom formalism. The code that implements the normalisation function will then be generated in the construction functions provided by the factory. An important consequence of this design is the impossibility to build a term which is not in normal form. Therefore, the boolean formulae that should only be

manipulated in disjunctive normal form can be defined as follows:

```
%gom {
  module Bool
  imports String
  abstract syntax
  Bool = True()
       | False()
       | Not(b:Bool)
       | And(l:Bool,r:Bool)
       | Or(l:Bool,r:Bool)
       | Var(name:String)
  rules() {
    Not(And(l,r))  -> Or(Not(l),Not(r))
    Not(Or(l,r))   -> And(Not(l),Not(r))
    Not(True())    -> False()
    Not(False())   -> True()
    Not(Not(x))    -> x
    And(x,Or(y,z)) -> Or(And(x,y),And(x,z))
    And(Or(y,z),x) -> Or(And(y,x),And(z,x))
  }
}
```

This construct ensures that there is no way to obtain a term that is not normal with respect to the rewrite system. It is also possible to use conditional rewrite rules in the ruleset, which lets the definition of ordered or balanced trees more convenient.

Describing elementary transformations.

The notion of labeled rule cannot be implemented with the same approach. A labeled rule corresponds to an elementary strategy that has to be manipulated as an object. Therefore, its implementation cannot be a *function*: this is not a first order object in JAVA. In our setting, an elementary strategy is implemented by a *class* which has an `apply` function. The implementation of such a class is generated automatically when using the `%strategy` keyword:

```
%strategy swap() {
  visit Bool {
    Or(Var(x),Var(y)) -> {
      if(x.compareTo(y)) {
        return `Or(Var(y),Var(x));
      }
    }
  }
}
```

The construction `%strategy swap()` defines a labeled rule whose name is `swap`. Similarly to the `%match` construct, the right-hand side of a rule can be any JAVA statement. In this example, `return ‘Or(Var(y),Var(x))` is used to return an object of sort `Bool`. Such a rule can then be applied to a term using the `apply` function. The following instructions applies the `swap` strategy to a simple term, storing the result in the `res` variable:

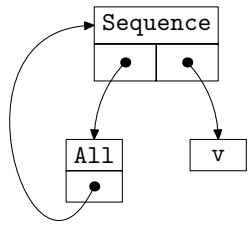
```
Bool b = ‘Or(Var("b"),Var("a"));
Bool res = ‘swap().apply(b);
```

Controlling the rewriting.

In [11], J. Visser introduced a new pattern to implement elementary combinators. This work, implemented in JJTraveler, is very important since it makes available in JAVA the combinator that exists in the Stratego language. A minor drawback of the approach is the implementation of the elementary strategies (the labeled rules), which requires some extra effort from the user. In particular, a quite complex interface has to be instantiated for each labeled rule. This code is now automatically generated by the `%strategy` keyword. A simple solution to implement our strategy language would have been to reuse JJTraveler in combination with the `%strategy` construct. This was our first attempt.

When programming with rules and strategies, as in Stratego [9,10] and ELAN [2], the definition of recursive composed strategies is very frequent. The definition of common rewriting strategies such as *top-down*, *bottom-up*, and *leftmost-innermost* is done via the use of an explicit recursion operator μ. This operator, like the `let rec` construct in functional languages, is essential. However, in JJTraveler this μ operator does not exist. Thus, the graph that represents the recursive strategy has to be encoded directly in JAVA. This is not easy and error prone. For example, the construction of the *BottomUp* strategy $\mu x\,.\,All(x)\ ;\ s$ is encoded as follows:

```
package jjtraveler;
public class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```



This graph is obtained by first allocating a `Sequence` with a dummy pointer set to `null`. Then the correct graph is built. This is clearly not the abstraction level we want to provide. In our framework, which is an extension of JJTraveler, we allow the definition of strategies that explicitly use the μ recursion operator. In JAVA, the *bottom-up* strategy can be implemented as follows:

```
Strategy BottomUp(Strategy s) {
  return ‘mu(x, Sequence(All(x),s));
}
```

This strategy can then be used to apply a strategy `s` to each node of a term `u` with `t = ‘BottomUp(s).visit(u)`.

Context and position

When analysing source code, context free information is not enough. For instance, when verifying a particular part of a program, sometimes we require to know what are the variables that are defined in the context, as well as their type. Such information can be obtained by letting user strategies to use a mutable state, that will be altered during the tree traversal. When finding a variable declaration for instance, the variable is stored, and further on, when finding a variable, it can be compared to what has been previously stored:

```
%strategy VarCheck(Stack bag) {
  visit Instruction {
    Assign(var,expression,body) -> {
      bag.push(‘var);
      this.visit(‘body);
      bag.pop();
    }
  }
  visit Expression {
    Variable(var) -> {
      if(!bag.contains(‘var)) {
        throw new Exception("Undefined variable " + ‘var);
      }
    }
  }
}
```

Another way to take the strategy application context into account is to use the notion of *position* in a term. The idea is to come back to the textbook notion of rewriting, resorting to the notion of position to identify a particular subterm or redex ($t_{|\omega}$ or $t[u]_\omega$ for example). The novelty here is to give access to the current position in a term while traversing it. This position can be manipulated as a first class object in the programming language.

To implement such a feature, each strategy combinator has to maintain the path from the root to the currently traversed term. This information is automatically maintained by the elementary combinators provided by the library: `Sequence`, `All`, `One`, *etc.* From a user perspective, the current position is returned by the function `getPosition()`. The object representing the position is not mutable and can be stored to be reused later. It will not be further modified when the strategy continues its traversal.

The expressive power provided by the explicit representation of positions is very high. For example, this feature is essential to implemented non deterministic exploration. Given a rewrite system implemented by a strategy, to compute the set of all

possible successors, a two step algorithm can be used. First, a top-down traversal is performed to identify all possible redexes. This set of redexes can be represented by a list of pairs (t, ω_i) where t is the term, and each ω_i correspond to the position of a redex. In a second step, the list is iterated, and the successors of $t_{|\omega_i}$ are computed and collected.

4 Conclusion

The design of a language that integrates the notions of rules and strategies in a mainstream programming language such as JAVA is not an easy task. We described the constructs and features that are required, and we showed how they can be integrated into the JAVA language.

This results in a language in which formally defining algorithms using terms, rules, and strategies is easy and elegant, without losing the flexibility and versatility of the underlying host language. It makes possible the mixture between the formal definition of a logic system, using TOM, with user code that provides a fully fledged interface implemented using low level code. The integration of formal aspects into a classical programming language eases a gradual integration of formal methods into existing projects. In practice, this integration becomes natural since the use of algebraic constructs, as those provided by TOM, makes the code smaller, more readable and robust.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[2] Peter Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, October 1998.

[3] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[4] Pierre-Etienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.

[5] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian 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.

[6] Antoine Reilles. Canonical abstract syntax trees. In Grit Denker and Carolyn Talcott, editors, *Proceedings of WRLA 2006*, volume 176, pages 165–179, Vienna, Austria, 2006. Electronic Notes in Theoretical Computer Science.

[7] Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[8] Mark van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. A generator of efficient strongly typed abstract syntax trees in java. *IEE Proceedings - Software Engineering*, 152(2):70–78, December 2005.

[9] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[10] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[11] Joost Visser. Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, pages 270–282, New York, NY, USA, 2001. ACM Press.