

# Aspect-Oriented Logic Meta Programming

Kris De Volder (kdvolder@vub.ac.be)

May 13, 1998

## Abstract

It is our opinion that declaring aspects by means of a full-fledged logic language has a fundamental advantage over using a restricted special purpose aspect language. As an illustration we present a simplified implementation of the Cool aspect weaver. Cool declarations are represented as logic facts in a Prolog like logic meta-language for Java. A fundamental advantage of this approach is that it enables *aspect-oriented logic meta programming*.

## 1 Introduction

We will briefly introduce the TyRuBa system. TyRuBa is an acronym for “Type Rule Base”. It was designed as an experimental system to explore the notion of *Type-Oriented Logic Meta Programming* [DV98]. The idea was inspired by the recent development of very expressive type systems, the type languages of which more and more resemble programming languages. The type system of the functional language Gofer [Jon95] was the main source of inspiration. This language embeds in its type system a kind of restricted logic programming language. The idea of type-oriented (logic) meta programming [DV98] basically comes down to the following statement: “The static type-language of a programming language should itself be a full-fledged (logic) programming language.”. Types can then be manipulated, constructed and implemented by means of a kind of static-type manipulating meta programs which are run as part of the type-checking process, at compile time. This greatly increases the expressive power of the type system.

A type language and an aspect language have a lot in common. They are both special-purpose declarative formalisms for expressing program annotations. Because of this similarity our arguments also apply to aspect-oriented programming. Our position statement therefore is: “*An aspect language should be embedded in a general purpose (logic) meta-programming language. This facilitates aspect-oriented (logic) meta programming.*”

As an illustration of the potential of the approach, we implemented a subset of the aspect language Cool as proposed by Lopes [LK97]. We did not copy all of her work because this would have taken too much time and effort. Instead we restricted ourselves to a simplified subset. What we want to draw attention to is the advantage of using a full-fledged logic language. The Cool aspect declarations are represented as logic facts and can therefore be accessed and declared by logic rules. The fundamental advantage this offers is that it enables *aspect-oriented logic meta programming*, thus increasing the expressiveness of the “aspect language” tremendously.

## 2 TyRuBa

The TyRuBa system is basically a simplified Prolog variant with a few special features to facilitate Java code generation. We assume familiarity with Prolog and only briefly discuss the most important differences.

### 2.1 Variables, Constants and Compound Terms

TyRuBa’s lexical conventions differ from Prolog’s. Variables are identified by a leading “?” instead of starting with a capital. This avoids confusion between Java identifiers and Prolog variables.

Some examples of TyRuBa variables are: `?x`, `?Abc12`, etc. Some examples constants are: `x`, `1`, `Abc123`, etc.

Because TyRuBa offers a quoting mechanism which allows intermixing Java code and logic terms, the syntax of terms is slightly different from Prolog's to avoid confusion with function or procedure calls in Java: TyRuBa compound terms are written with `<` and `>` instead of `(` and `)`.

## 2.2 Quoted Java Code

TyRuBa provides a special kind of compound term that represents a piece of “quoted” Java code. Basically this is simply a special kind of string delimited by `{` and `}`. Instead of characters however, the elements of such quoted code blocks may be arbitrary Java tokens, intermixed with logic variables or compound terms. The Java tokens are interpreted as special constants whose name corresponds exactly to the printed representation of the token. Tokens in quoted Java code may include `{` and `}` as long as these are properly balanced. The following is an example of a quoted Java term.

```
{void foo() {  
    Array<?E1> contents = new Array<?E1>[5];  
    ?E1 anElement=contents.elementAt(1); } }
```

In the above example we see a compound term `Array<?E1>`. We find several name constants `contents`, `new`, `anElement`, ... There are two integer literals 5 and 1. The remainder of the tokens such as `=`, `.` `(` are Java tokens treated as name constants with “strange names”.

## 3 Logic Meta Programming

The idea of logic meta-programming is very simple. A base program is represented indirectly by means of a set of logic propositions. These logic propositions are stored in a logic data repository and the link between the actually represented program and its logic representation is concretized under the form of a code generator. Basically the code generator performs queries to find out what it needs to know in order to construct the base program and outputs code in the base language.

Logic meta-programming becomes possible in such a setting because a logic program can be thought of as representing the set of logic propositions that can be proven from their facts and rules. The full power of the logic paradigm may thus be used to describe base-language programs indirectly. This opens up a tremendous potential as we will try to illustrate in the rest of this position paper.

The mapping scheme between logic representation and base program may vary and determines the kind of information that is reified and accessible to meta programs. The representational mapping and code generator used in this example assume that classes are represented by means of a number of facts which state that the class has certain methods, instance variables or constructors. This is very similar to Lamping's “methods as assertions” approach [Lam94]. Figure 1 gives an example Java class declaration and its corresponding representation as a set of TyRuBa propositions. Note that the propositions are all qualified by the symbol `JCore`<sup>1</sup>. For the time being this is unimportant and can be ignored.

## 4 The Synchronization Problem

The problem in writing multi-threaded Java applications is that synchronization code ensuring data integrity tends to dominate the source code completely. As a result it becomes entangled and unmanageable. As an illustration of the problem, consider the (generated) code which includes synchronization aspect code listed in figure 2. The “synchronized” version of the class has some

---

<sup>1</sup>In Lopes's system `JCore` is a simplified version of Java which is used to express the basic functionality without aspect code.

<pre> class BoundedStack {     static final int MAX = 10 ;      int pos = 0 ;     Object[] contents=new Object[MAX];      public void print ( ) {...body...}      public Object peek ( ) {         return contents[pos]; }     public Object pop ( ) {         return contents[--pos]; }     public void push (Object e) {         contents [pos++]=e ; }     public boolean empty ( ) {         return pos == 0 ; }     public boolean full ( ) {         return pos == MAX ; } } </pre>	<pre> class_(JCore,BoundedStack). var_(JCore,BoundedStack,int,MAX,{     static final int MAX = 10;}). var_(JCore,BoundedStack,int,pos,{int pos = 0;}). var_(JCore,BoundedStack,{Object []},contents,     {Object [] contents = new Object [MAX];}). method_(JCore,BoundedStack,void,print, [],{     public void print()},{...body of print...}). method_(JCore,BoundedStack,Object,peek, [],{     public Object peek()},{return contents[pos]; }). method_(JCore,BoundedStack,Object,pop, [],{     public Object pop()},{return contents [--pos]; }). method_(JCore,BoundedStack,void,push, [Object],{     public void push(Object e)},{contents [pos++]=e;}). method_(JCore,BoundedStack,boolean,empty, [],{     public boolean empty()},{return pos==0;}). method_(JCore,BoundedStack,boolean,full, [],{     public boolean full()},{return pos==MAX;}). </pre>
---	---

Figure 1: A Java program (left) and its representation in TyRuBa (right).

instance variables which are used to keep count of how many times a method has been started. These counters are updated and consulted upon entry and exit of a methods.

```

class BoundedStack {
    public Object peek ( ) {
        synchronized ( this ) {
            while ( ! ( (COOLBUSY_Lpush_R==0) && (COOLBUSY_Lpop_R==0) ) ) {
                try { wait ( ) ; }
                catch ( InterruptedException COOLe ) { }
            }
            ++ COOLBUSY_Lpeek_R ;
        }
        try { return contents [ pos ] ; }
        finally { synchronized ( this ) {
            --COOLBUSY_Lpeek_R; notifyAll() ;
        }}
    }
    ...other method declarations...
    private int COOLBUSY_Lpeek_R = 0 ;
    ...other variables...
}

```

Figure 2: Synchronization code generated for BoundedStack

Aspect-oriented programming solves this problem by providing a special-purpose aspect language with which the synchronization aspect can be expressed separately from the base functionality. We implemented a simple subset of the Cool aspect language proposed by Lopes [LK97]. We did not copy the syntax exactly however, but simply express the synchronization aspect by means of logic facts. As an example consider the BoundedStack “aspect program”:

```

selfExclusive(BoundedStack,push).
selfExclusive(BoundedStack,pop).
selfExclusive(BoundedStack,print).

```

```

mutuallyExclusive(BoundedStack, [push, pop, peek]).
mutuallyExclusive(BoundedStack, [push, pop, empty]).
mutuallyExclusive(BoundedStack, [push, pop, full]).
mutuallyExclusive(BoundedStack, [push, pop, print]).
requires(BoundedStack, push, {!full()}).
requires(BoundedStack, pop, {!empty()}).

```

The `xxxExclusive` facts declare which methods should not be called concurrently. Whenever there is a fact `selfExclusive(?c, ?m)` this means that the method `?m` of class `?c` should not be started concurrently with itself. If there is a fact `mutuallyExclusive(?c, ?methods)` then this means that no method in the list `?methods` may be started concurrently with any other method in the list. A method from a mutually exclusive list may however be started concurrently with itself unless it is declared `selfExclusive` as well. The `peek` method for example is allowed to be executed concurrently with itself, but not with `push` or `pop`. Note that by identifying methods by their names only, we implicitly assume that the base program does not use method overloading. The same simplifying assumption is made by Lopes also. It is not difficult to support overloading, but the example would then become more verbose because the types of the arguments would also have to be listed to identify a method. Additional guards, other than those derived from `xxxExclusive` declarations, may be added to a method by declaring a fact: `requires(?c, ?m, ?condition)` This means that the method `?m` in class `?c` may not be started unless the `?condition` expression evaluates to `true`. The given example `requires` declarations ensure that no elements are ever popped from an empty stack nor pushed onto a stack which is full.

Some additional aspect declarations given by facts of the form

```

onEntry(?class, ?method, ?statements).
onExit(?class, ?method, ?statements).

```

specify synchronization related actions that have to be performed upon entry and exit of a method. These declarations were not used in the example but will be used indirectly to specify the actions that maintain counter variables upon entry and exit of a method.

## 5 The Weaver: a special purpose TyRuBa code generator

Because TyRuBa is an experimental system, the code-generator and representational mapping are not hard-coded into it. Instead there is “hook” which allows plugging in custom code generators implemented in TyRuBa itself. The framework of logic meta-programming supported by the TyRuBa system is an excellent medium to implement the Cool aspect weaver. Because of limited space we cannot go into the details of the implementation of the code generator and only give a rough outline. The interested reader is referred to our Ph.D. dissertation [DV98]. As an indication that TyRuBa is very well suited to the task at hand we may mention that it took us less than a day to implement a code generator that supports the above aspect declarations. The most interesting thing to mention about it is that the aspect-oriented meta programming technique is already used in the very implementation of the weaver itself!

### 5.1 Low-level aspect declarations

The reason that we can use aspect-oriented meta programming in part of the implementation of the aspect weaver is due to the fact that part of the aspect language can be defined in terms of its own more low-level features. The synchronization code for maintaining the method counters could for example be added by means of `onEntry` and `onExit` declarations. Likewise, the conditions that consult the counters to verify whether a method may be started can be declared by means of `requires`. We therefore first implemented support for the more low-level aspect declarations `onExit`, `onEntry`, and `requires`. Because of lack of space we won't go into the implementation of these more low-level features.

## 5.2 Mutually exclusive and self exclusive declarations

We can easily provide support for `selfExclusive` and `mutuallyExclusive` declarations in terms of the more low-level declarations by means of logic rules. Since this is a nice example of aspect-oriented logic meta programming we will take a look at one of these rules to get an idea of what they look like.

```
requires(?class,?name,{COOLBUSY<?other> == 0}) :-  
  mutuallyExclusive(?class,?names),  
  element(?name,?names),element(?other,?names),  
  NOT(equal(?name,?other)).
```

What this rule states is that a guard condition “`COOLBUSY<?other>==0`”<sup>2</sup> for must be added to a method `?name` whenever `?name` and `?other` are two distinct methods occurring together in a single `mutuallyExclusive` declaration.

Some other rules, not presented here, take care of `selfExclusive`, add the `COOLBUSY<?method>` counter variables, and insert administrative code to maintain the counters upon entry and exit of a method. An outline of the resulting output code for `BoundedStack` was already listed in figure 2.

## 6 Aspect-Oriented Meta Programming

There is a major fundamental advantage to using TyRuBa instead of a special purpose aspect language. The facts declaring the aspects can be accessed and declared by logic rules, thus enabling *aspect-oriented meta programming*. We already made use of this potential in the implementation of the code generation for `selfExclusive` and `mutuallyExclusive`. We present one more example showing that this is indeed a considerable advantage and increases the power of the aspect language tremendously.

As we were experimenting with the `BoundedStack` example we were not entirely pleased with the way method locking strategies are expressed by means of `mutuallyExclusive` and `selfExclusive` declarations. As we reasoned about these declarations, we came to the conclusion that the rationale behind the declarations was a reasoning about which methods modify or inspect what state. It would therefore be better if this kind of information can be declared directly and explicitly. Instead of the previously given set of `xxxExclusive` declarations, we would like to write:

```
modifies(BoundedStack,push,this).  
modifies(BoundedStack,pop,this).  
inspects(BoundedStack,peek,this).  
inspects(BoundedStack,empty,this).  
inspects(BoundedStack,full,this).  
modifies(BoundedStack,print,SystemOut).  
inspects(BoundedStack,print,this).
```

The above should provide sufficient information to derive `selfExclusive` and `mutuallyExclusive` properties automatically. We will see that indeed it does and that we can define some simple rules that express how to do so. The first rule simply states that a method is self exclusive if it modifies some state.

```
selfExclusive(?class,?method) :- modifies(?class,?method,?thing).
```

A method which inspects a state is mutually exclusive with all methods which modify the same state:

---

<sup>2</sup>The variable names appearing in figure 2 are “mangled” identifiers representing compound TyRuBa terms of the form `COOLBUSY<...>`

```
mutuallyExclusive(?class,[?inspector|?modifiers]) :-
    inspects(?class,?inspector,?thing),
    FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
```

We need one more rule to say that all methods which modify the same state should be mutually exclusive with each other.

```
mutuallyExclusive(?class,?modifiers) :-
    NODUP(?thing,modifies(?class,?xxx,?thing)),
    FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
```

## 7 Conclusion

We have illustrated how an aspect language can be embedded in the logic paradigm by representing aspect declarations as logic facts. We did this for a simplified subset of the Cool aspect language proposed by Lopes to express synchronization aspects of Java programs. For this particular example it was fairly easy to implement the aspect weaver as a special purpose TyRuBa code generator.

An equally clear syntactic separation between aspect program and basic functionality can be achieved when using a general purpose declarative programming language such as TyRuBa instead of a special purpose aspect language. In the presented example, one set of facts specifies the basic functionality of the `BoundedStack` class. Another completely disjunctive set represents the synchronization-aspect program.

That aspects are expressed by means of a full-fledged logic programming language has a major fundamental advantage over using a restricted special-purpose aspect language. It allows for what we call *aspect-oriented meta programming*. Aspect declarations are merely facts in the logic program and they can therefore be consulted or defined indirectly by means of logic rules. Bluntly put, this means that if you do not like the aspect language the way it is, you can simply define your own language or language extension “on the fly” by means of a few simple rules. As an example we presented and implemented an alternative to the Cool declarations that allows us to capture the reasoning behind the `mutuallyExclusive` and `selfExclusive` declarations in terms of which method inspects or modifies what state. Implementing this extension of the aspect language was very easy, only requiring three simple and intuitive logic rules.

## References

- [DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, (in progress) 1998.
- [Jon95] M. P. Jones. *Gofer*. CS, Yale, August 1995.
- [Lam94] John Lamping. Methods as assertions. In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming 8th European Conference, ECOOP '94 Bologna, Italy, Proceedings*, volume 821 of *Lecture Notes in Computer Science*, pages 60–80. Springer-Verlag, New York, N.Y., July 1994.
- [LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-007 P9710047, Xerox Palo Alto Research Center, <http://www.parc.xerox/aop>, 1997.