Design and use of aspects in rule-based programming

Karl Trygve Kalleberg [1,2]

Department of Informatics
University of Bergen, P.O. Box 7800
N-5020 BERGEN, Norway

Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands

Eelco Visser [3]

Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands

Abstract

Recently, aspect-oriented programming has enjoyed recognition as a practical solution for an important class of separate and cross-cutting concerns. This paper shows aspects to be useful in the context of rule-based programming languages by demonstrating an adaptable term type checker and after-the-fact algorithm extension. It briefly discusses some of the challenges faced when designing and implementing an aspect extension for and in a rule-based term rewriting system.

Key words: aspect-oriented programming, language extension, rule-based programming

1 Motivation

The goal of aspect-oriented software development is the modularization of crosscutting concerns. The goal of modularization is to structure our software close to how we want to think about it, and was already identified in [Par72].

[1] Research carried out with funds from the Research Council of Norway (NFR)
[2] Email: karltk@ii.uib.no
[3] Email: visser@cs.uu.nl

Aspect-oriented programming (AOP) presupposes that a decomposition of the properties of a problem domain falls into *components* and *aspects*.

Components are units of basic functionality which map cleanly into language abstractions such as functions, whereas aspects are non-functional properties which cross-cut these.

Make aspects first-class citizens in the language provide us with greater flexibility in modularizing our software, as the cross-cutting properties need no longer be scattered across the components, but may be declared entirely in separate units, one for each property.

In most language settings, the poster-child examples for aspects include logging (e.g., having the compiler automatically insert logging code before and/or after calls to certain functions) and error-handling policies (e.g. automatically adding parameter validation for certain functions, and being able specify whether the validation should happen in the caller or the callee). [xx: reference].

While many of these crosscutting concerns easily transport over to rule-based programming languages, we will focus on *policy enforcement*, exemplified by format checking of terms and *after-the-fact extensibility*, exemplified by algorithm adaptation.

There are also other cross-cutting properties which are peculiar to rule-based programming and their domain of operation. Of particular interest is being able to trace origins through multiple transformation stages [vDKT93] and doing rewriting with layout [BV00].

As we shall discuss later, other metaprogramming techniques, such as macros, meta-object protocols and general program transformation also offer solutions to the same problems.

Compared to macros, aspects provide the benefit that the variation points need not be known or designed for by the original implementor. The aspect language is declarative, small and precise, making its use by programmers a lot more practical than general program transformation. Adding an aspect extension to an existing language is a relatively minor undertaking compared to a full meta-object protocol, while still offering non-trivial enhancements.

2 The fundamentals of Stratego

In order to discuss aspects for rule-based programming, we shall briefly introduce the Stratego language for term rewriting. Stratego is based on the concept of rewriting strategies. A rewriting strategy is an algorithm for transforming a term with respect to a set of rewrite rules.

In most rewrite engines these strategies are fixed, and often require the set of rules to be confluent and terminating. In Stratego, the strategies are user-defined, thus providing the user a fine-grained control over the selection of rules and the order of application.

While a term (discussed next) may represent anything, the Stratego library and tool chain is geared towards computer languages and program transformation.

2.1 Signatures Terms

Signatures describe the structure of terms. A *signature* consists of a set of *constructors*, each taking zero or more arguments.

Example 2.1 `signature`

```
constructors
  Num  : Int -> Expr
  Plus : Expr * Expr -> Expr
  Mul  : Expr * Expr -> Expr
```

In Ex 2.1, `Num` and `Plus` are constructors taking one and two arguments, respectively.

Stratego provides the constructor `Int` in its library. Following the signature for simple arithmetic in Ex 2.1, we can make the term

```
{\tt Plus(Mul(Int(1), Mul(2)), Int(3))}
```

By definition, a term may not contain wildcards or variables.

2.2 Match and build

At the heart of Stratego we find the two primitive operators `match` and `build`. `build` is used to construct a term, written `!Int(0)` or `!Plus(x, y)` where `x` and `y` are variables containing terms.

A "term" containing wildcards (written `_`) or variables is called a *pattern*. The `match` operator will attempt match a pattern against a term, binding any variables in the pattern if the match succeeds. Evaluating `?Plus(_, z)` against the term `Plus(Int(0), Int(1))` will bind `z` to `Int(1)` but ignore whatever is the first argument to `Plus`.

2.3 Rewrite rules

Rewrite rules express the basic transformations on terms. They have the form `R : l -> r`, where `R` is the rule name, `l` the left-hand side pattern and `r` the right-hand side pattern. It may optionally have a `where`-clause, called a *rule body*.

Example 2.2 `rules`

```
  Eval: Plus(Int(x), Int(y)) -> Int(z) where <add> (x, y) => z
```

The body contains a strategy expression, described next. If applied to the term `Plus(Int(10), Int(32))`, the rule in Ex 3.1 will result in the term `Int(42)`, as expected.

2.4 Rewriting strategies

A *rewriting strategy* is an algorithm for transforming a term. If it succeeds, the result is the transformed term. If it fails, there is no result. Strategies control the order of application of rules or other strategies. For example `bottomup(Eval)` will attempt to apply the `Eval`[4] rule to all terms, in a bottom up fashion.

Strategies can be combined into a *strategy expression* using a set of built-in operators, such as *sequential composition* (`;`), (*non-deterministic* `+`) or *deterministic choice* (`<+`, `+>`).

The language also provides ready-made primitive strategies: `id` will always succeed and return the identity term and `fail` will always fail.

Given Ex 3.1's definition of `Eval`, the strategy `bottomup(Eval)` cannot work: We cannot expect to see a `Plus` at the bottom of a term; it always has two subterms. `bottomup(Eval <+ id)` would do the job, by first attempting to apply `Eval`, then `id` if `Eval` failed. `bottomup(try(eval))` is an equivalent but more convenient way to express the same.

A strategy is defined on the form `S = ...` where `S` is the strategy name and `...` a strategy expression, e.g.

```
partial-eval = bottomup(try(Eval))
```

2.5 Rules and strategy parameters

The strategy invocation `bottomup(try(Eval))` is an example of higher-order parameters. The definition for `bottomup` is

```
bottomup(s) = all(bottomup(s)) ; s}.
```

We can do the same with rules:

```
Eval(op): Plus(Int(x), Int(y)) -> Int(z) where <op> (x, y) => z
```

Additionally, rules and strategies may receive term arguments. This is conceptually the same as regular argument passing in imperative languages. Consider

```
ReplaceMagic(|x) : Int(0xDEADBEEF) -> Int(x)
```

which can be used to swap one magic number for another: `ReplaceMagic(|0xC0FFEE)`.

2.6 Modularization in Stratego

All signature, rule and strategy definitions must occur inside a module. A module is simply a file starting with the declaration `module` *name*. A module is futher decomposed into *sections*, each with a *section header*. Rule definitions

[4] For reasons of egyptology, rule names are written in `CamelCase`, and strategies have `hyphenated-names`

go under the `rules` section header, strategies under the `strategies` header, and so on.

When decomposing properties from our problem domain, these are the abstractions we can map to.. This modularization mechanism is susceptible to the same fundamental problem popularized by the AOP community: What happens when our properties cannot be cleanly expressed as components, for example if we want to enforce a program-wide policy or simply add logging or profiling?

In the sequel, we shall describe an extension to the Stratego language which deals with this.

3 AspectStratego

AspectStratego in an extension to the Stratego language which tackles a certain class of cross-cutting but separate concerns, namely those that can captured by the declarations of the extension language.

We provide the programmer with *joinpoints*; predicates on the program structure. These can be combined into boolean expressions called *pointcuts*. The programmer can use a pointcut in an *advice* to pinpoint places to insert code *before*, `after` or `around`. The inserted code is declared as part of the advice. Advice are in turn gathered in named entities called *aspects*. The act of composing a program with its aspects is called *weaving*.

In the interest of comparison with other aspect languages, the popular logging aspect is displayed in Ex **??**.

Example 3.1 `module logging-example`

```
strategies
  invoked(|s) = ![ "Rule '", s, "' invoked" ]

aspects
  pointcut log-rules(n) = rules(* => n)
  aspect logger =
  before(r)        : log-rules(r) = log(|Debug, invoked(|r))
  after-fail(r)    : log-rules(r) = log(|Debug, failed(|r))
  after-succeed(r): log-rules(r) = log(|Debug, succeeded(|r))
  after(n)         : log-rules(n) = log(|Debug, finished(|r))
```

The aspect reads not totally unlike the AspectJ language [KHH+01]. We have tried to keep much of the terminology, as well as some properties of its joinpoint model, though the latter has been adapted to fit better within the paradigm of rule-based rewriting systems.

As with all Stratego code, aspects must reside in modules. An aspect can only be declared within an `aspects` section [5]. The same holds for a pointcut.

[5] While `aspects` sections can be interleaved with the other Stratego sections `strategies`,

The fact that pointcuts are declared outside the aspects allow them to be shared between aspects, a feature we will rely on in 4.2.

The code in **??** contains all of the basic ingredients of our aspect language: joinpoints, pointcuts, aspects and advice.

A *joinpoint* is a well-defined point in the program execution through which the control flow passes twice: once on the way into the sub-computation identified by the joinpoint, and once on the way out.

The code snippet `rules(n)` is an example of a joinpoint.

Joinpoints can be combined into pointcuts. A *pointcut* is an boolean expression on joinpoints. In the code, `log-rules(n)` is the most trivial of pointcuts: just a joinpoint.

An *advice* is a body of code which is always associated with a pointcut. The code will, at the user's specification, be inserted *before*, *after* or *around* the points where its associated pointcut matches.

The *aspect* `logger` is composed of four advice. `before(r) : log-rules(r) = log(|...)` is an advice, more specifically a *before advice*. The body, `log(| invoked(|r))`, is a perfectly ordinary strategy expression. `log` is part of the Stratego library, and `invoked` is a strategy constructing an appropriate string.

Observe that the call to an external, named strategy opens up the possibility for expression "meta"-aspects which augment, override or replace the internals of the logging aspect itself.

Attention must be drawn to the slightly confusing direction of information flow in the joinpoint, pointcut and advice arguments. The pointcut `log-rules(n) = rules(n)` is to be read as a pattern match on rules, with the free variable `n`. When a match succeeds, `n` will contain the name of the matching rule, and this will be "exported" through the pointcut, as the *pointcut parameter* `n`.

Subsequent advice, e.g. `before(r) : log-rules(r) = ...` can "import" the pointcut parameters, and reference them in the advice body as ordinary variables.

Example 3.2 `module real-code`
```
imports liblib logging-example simple-arithmetic
strategies
Eval : Plus(Int(x), Int(y)) -> Int(<add> (x, y))
Eval : Mul(Int(x), Int(y)) -> Int(<mul> (x, y))

main = load-term ; bottomup(try(Eval))
```

Ex 3.2 shows importing the module in Ex **??**, all its aspects are considered. In this case, the effect is that all rules (even ones in the standard library) will have the logging code defined in the `logger` aspect's advice weaved into them.

`rules`, `constructors`, by their very definition, aspects should rather be declared in separate modules.

Name	*Matches*
`calls(`*name-expr* `=>` *n*`)`	strategy or rule invocations
`strategies(`*name-expr* `=>` *n*`)`	strategy executions
`rules(`*name-expr* `=>` *n*`)`	rule executions
`matches(`*pattern* `=>` *t*`)`	pattern matches
`builds(`*pattern* `=>` *t*`)`	term constructions
`fails`	explicit invocations of fail

Table 1
Synopsis of the joinpoints

Given that `simple-arithmetic` contains the signature from Ex 2.1 and provided that `load-term` constructs a valid term for this signature, the program will print a trace of the `Eval` rules to the console.

3.1 Details of the aspect language

3.1.1 Joinpoints

Table 1 shows the available joinpoints in AspectStratego.

The *name-expr* can either be a complete identifier name, such as `Eval` or a prefix, such as `bottom*`. If need arises, this could easily be extended to full regular expressions. The result of a *name-expr* is a string.

The *pattern* is an ordinary Stratego pattern, which may contain both variables and wildcards. The variables may be exported by the pointcut, e.g

```
pointcut pc0(n, t) = matches(Int(n) => t)
```

will export the subterm of `Int` as `n`, and the whole term (including `Int`) as `t`, both as terms.

3.1.2 Pointcuts

A *pointcut* is an boolean expression on joinpoints and joinpoint context information involving the logical operators `;` (and), `+` (or) and `not`.

Table 2 shows the available *Joinpoint context information*. These are predicates on the program flow or structure that can be used to further refine the location identified by a joinpoint.

An example of a pointcut is `calls(foo) ; args(x, y) ; withincode(bar)`, which will match all calls to `foo` with two higher-order strategy arguments, within in all strategies and rules named `bar`.

When `matches` and `builds` occur as joinpoints, they mean invocations of the `match` operator inside a rule left-hand side or `build` operator invocations inside a rule right-hand side.

The pointcut

Name	*Matches*
`withincode(`*name-expr* `=>` *n*`)`	joinpoints within a strategy or rule
`matchflow(`*flow-regex* `=>` *t*`)`	joinpoints with given execution history
`args(`n_0`,`n_1`,...,`n_n`)`	joinpoints with given arity
`matches(`*pattern* `=>` *t*`)`	pattern matches
`builds(`*pattern* `=>` *t*`)`	term constructions

Table 2
Synopsis of joinpoint context information

Name	*The body of code is ...*
`before`	run before the pointcut
`after`	run after the pointcut
`after fail`	run after the pointcut, iff the code in the pointcut failed
`after succeed`	run after the pointcut, iff the code in the pointcut succeeded
`around`	wrapped around the pointcut

Table 3
Synopsis of the aspects

```
pointcut pc1 = rules(*) ; matches(Int(_)) ; builds(Int(_))
```

will match all rules `R : Int(...) -> Int(...)` where `...` is any pattern.

3.2 Advice

Table 3 lists the available advice types for AspectStratego.

We have already seen the before and after advice, but some subtleties deserve further clarification.

In Stratego, a strategy or rule is always applied to an implicit term. The code

```
!Int(0) ; ?Int(x) ; !x
```

is to be read as follows: replace the implicit term with `Int(0)`, match the pattern `Int(x)` against the implicit term and bind the variable `x` on success, replace the implicit term with the content of `x` (i.e. 0).

Example 3.3 `aspects`

```
  pointcut pc0(n) = builds(Int(n))
  aspects subverter =
    before(t) : pc0(t) = <add> (t, 1) => t' ; !Int(t')
```

Weaving the `subverter` in Ex 3.3 into the above code fragment, would (after didactical sanitation) result in the code:

```
!Int(0) ; <add> (t, 1) => t' ; !Int(t) ; ?Int(x) ; !x
```

Which would change the final result to 1, not 0.

How the strategy or rule invocations inside the advice body changes the implicit can be controlled in two ways: a strategy expression, such as the entire advice body, may be wrapped (entirely or partially) in a `where`-clause, in which case the implicit term remains untouched after advice body execution. If the strategies invoked in the advice body take term arguments and leave the implicit term untouched, such as `log` in ??, the `where` is not needed.

This manipulation of the implicit term turns out to be extremely useful in `around` advice. In around advice, the implementor of the advice decided where (and if) the pointcut itself should be executed, by invoking the placeholder strategy `proceed`. As an advice may `proceed` multiple times, it is trivially possible to add "retry failed match" handling.

Furthermore, the flexibility of around obviously allows the aspect programmer to completely override and replace the implementation of existing strategies and rules, such as the ones found in the Stratego standard library.

3.3 Implementation of the weaver

Apart from the syntactical extensions, our aspect compiler is but one additional step in the compiler pipeline. It is itself divided into four traversals, two of which are worth further mention.

After normalization, the advice are collected and pointcuts are evaluated. They are decomposed into conjunctive normal forms called *fragments*. A fragment contains one jointpoint and an arbitrary set of joinpoint context predicates all separated by logical *and*.

For example, the pointcut `( rules(n) + strategies(n) ; args(y)` is split into the two fragments `rules(n) ; args(y)` and `strategies(n) ; args(y)`.

From each fragment we build a rule, where the left-hand side matches a term in the Stratego AST. For example, the joinpoint `rules` will match against `RDefT`. The joinpoint context predicates are evaluated in the rule body, and if they all succeed, the rule succeeds. Each built rule

As the rules are generated for each pointcuts, they are tagged with all "subscribing" advice.

The actual weaving then simply amounts to a bottomup traversal of the AST, exhaustively applying all generated rules. On any node where one or more rules match, their associated advice are collected and applied in place.

If the rule matches a strategy expression (`calls`, `fails`, `matches`, `builds`), the template in Fig 3.3 is inserted where the cursive identifiers (i.e. *before*) are filled in with the approprate advice type, if any.

If the rule matches a definition (`strategies`, `rules`), the original definition is renamed and a *shadowing* strategy is created. The shadowing strategy

```
/before/ ;
if /pointcut-code/ then
if /after-succeed/ then try(/after/) else /after/ ; fail end
else
if /after-fail/ then try(/after/) else /after/ ; fail end
end
```

will have the template in Fig. 3.3 as its body, with *pointcut-code* being an invocation of the original definition.

The weaver can be realized entirely in the front-end, on the normalized abstract syntax tree. It does not need access to the lower-level intermediate representations. Which is not to say that weaved code doesn't open up for new optimizations in the backend – in fact, this is an obvious area for improvement.

4 Case studies

We have used the aspect extension described previously for two case studies relevant to rule-based programming. The first is adding structural well-formedness checking to a set of rewrite rules and the second is a way to extend algorithms by invasive strategy composition.

4.1 Type checking

In Stratego, terms are built with constructors from a signature, but the language does not enforce a typing discipline on the terms. With the signature in Ex 2.1, you are perfectly allowed to `!Num(Plus(0, 0))`.

As the normal mode of operation for Stratego is local and piecewise rewriting of terms, possibly from one signature to another, invalid intermediates cannot be forbidden.

The StrategoXT environment comes with term-checking tools which can be applied to the resulting term of a Stratego program. While all signature violations will be caught by this tools, it cannot help tell where in your program the transgression took place.

One is left with the usual suspects for dealing with this issue: sprinkling the program code liberally with either calls to `debug` or manually insert type checking in strategic places, which will traverse entire terms any report any errors.

The former is cumbersome, as changes to the code will require manual re-inspection should another failure unexpectedly be discovered. The latter is slow, as it can do no better than complete term traversal to perform a type check. When using embedded concrete syntax, which often results in big, static terms, this is particularly annoying, as the terms constructed from the concrete syntax are known to be well-formed.

This smacks of an aspect, as it is clearly a cross-cutting and separate concern. The code in Ex 4.1 shows just such an aspect.

Example 4.1 `module typecheck-example`
```
aspects
  pointcut typecheck-rules(n, t) = rules(n) ; builds(t)

  aspect typechecker =
    around(n, t) : typecheck-rules(n, t) =
      proceed ; ( typecheck(|t) <+ ( <log|incorrect-term(n) ; fail ))
```

Ex 4.2 shows an example usage of the aspect.

Example 4.2 `module real-code`
```
imports liblib typecheck-example

strategies
typecheck = typecheck(grammar-checker)
main = ....
```

The `typechecker` aspect makes use of the format checker functionality in StrategoXt to pervasively weave format checking into all rules in your program.

Caveat: In the interest of brevity, this example does not deal with programs whose rule sets construct terms for multiple grammars, but this could be dealt with either by partitioning the rule sets into different modules, one module for each grammar, or by having them follow discernible naming conventions.

4.2 Extending algorithms

5 Discussion

When implementing the declarative aspect language, we were faced with the traditional dilemma of realizing it (a) as a full interpreter (optionally with a partial evaluator), (b) as a fully compiled language where the aspect "programs" were dynamically linked into the compiler at runtime, or (c) by exploiting the runtime properties of dynamic rules heavily.

Both (a) and (b) have been amply documented in existing literature. In this article, we show how dynamic rules can be used as a partial substitute for interpretation.

It turns out that the only part of our aspects which need runtime variability provided by interpretation are the pointcut expressions. In particular, the entire weaver can be easily be realized using (static) rules and strategies.

However, even after normalizing the pointcuts into pointcut fragments, they cannot be expressed entirely as dynamic rules, as dynamic rules may not have overlapping left-hand sides. This restriction prohibits having one `calls(foo*)` pointcut and one `calls(foobar)`, thus name matching must be done interpretively in the dynamic rule body. The same holds for argument matching.

5.1 Related work

In [KL03], it is argued that origin tracking and rewriting with layout are instances of the more general problem of propagating term annotation, and that propagation is a separate concern which should be adaptable by the programmer.

Aspect-weaving by graph rewriting, [?], [Läm99], [MRB+], [KLM+97], [LK97]. [KL03], [vDKT93], [Ass03].

5.2 Future work

- Better error handling
- Better reflection
- Use-case: Totem propagation
- Use-case: Transformation invariants?
- Use-case: Views on terms?

6 Conclusion

- Expressed in a familiar fashion (AspectJ aspect model)

References

[Ass03] U. Assmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., 2003.

[BV00] M.G.J. van den Brand and J.J. Vinju. Rewriting with layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of RULE2000*, 2000. [ps], [slides].

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[KL03] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society Press, September 2003. 10 pages.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Läm99] Ralf Lämmel. Declarative aspect-oriented programming. In Olivier Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop* *on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

[LK97] C.V Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, 1997.

[MRB+] MohammadReza Mousavi, Michel Reniers, Twan Basten, Michel Chaudron, Giovanni Russello, Angelo Cursaro, Sandeep Shukla, Rajesh Gupta, and Douglas C. Schmidt. Using Aspect-GAMMA in the design of embedded systems. In *Proceedings of Seventh Annual IEEE International Workshop on High Level Design Validation and Test*, pages 69–74, Cannes, France. IEEE Computer Society Press, Los Alamitos, CA, USA, 2002.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[vDKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15(5-6):523–545, 1993.