– I must say, cracking is much like acupuncture. It's about finding the right spots to insert some NOPs.

– Håvard Sørbø.

5 Modularising Cross-Cutting Transformation Concerns

Properties such as logging, persistence, debugging, tracing, distribution, performance monitoring and exception handling occur in most programming paradigms and are normally very difficult or even impossible to modularise with traditional modularisation mechanisms because they are cross-cutting. Recently, aspect-oriented programming has enjoyed recognition as a practical solution for separating these concerns.

This chapter describes an extension to the Stratego term rewriting language for capturing such properties. It demonstrates how this aspect extension offers a concise, practical and adaptable solution for dealing with unanticipated algorithm extension for forward data-flow propagation and dynamic type checking of terms. The chapter describes and discusses some of the challenges faced when designing and implementing an aspect extension for and in a rule-based term rewriting system.

The aspect language described in this chapter was first presented in the paper *"Combining Aspect-Oriented and Strategic Programming"* written together with Eelco Visser [KV05].

5.1 Introduction

Good modularisation is a key issue in design, development and maintenance of software. Software should be structured close to how one wants to think about it [Par72] by cleanly decomposing the properties of the problem domain into basic function units, or *components*. These can be mapped directly to language constructs such as data types and functions. Not all properties of a problem decompose easily into components. Some turn out to be non-functional and these frequently cross-cut the module structure. Such properties are called *aspects*. The goal of aspect-oriented software development [KLM+97] is the modularisation of cross-cutting concerns. By making aspects part of the programming language, one is left with greater flexibility in modularising software. The cross-cutting properties need no longer be scattered across the components. Using aspects, they may now be declared entirely in separate units, one

for each property. Examples of general aspects include security, logging, persistence, debugging, tracing, distribution, performance monitoring, exception handling, origin tracking and traceability. All these occur in the context of rule-based programming in addition to some which are domain-specific such as rewriting with layout. Existing literature predominantly discusses aspect-based solutions to these problems for object-oriented languages and the documentation of paradigm-specific issues and deployed solutions for the rule-based languages is scarce.

This chapter describes the design and use of aspects in the context of rule-based programming. It introduces the AspectStratego language which enables modular declaration of many separate cross-cutting concerns encountered in rule-based transformation languages. A discussion of the joinpoint model underlying AspectStratego is provided. In addition, the practical usefulness of the extension is demonstrated by three small case studies motivated by the problem of constant propagation. The contributions of this chapter include:

1. The description of a novel aspect language extension implemented for and in a rule-based programming language.

2. An example of its suitability for adding flexible dynamic type checking of terms in a precise and concise way.

3. A demonstration of its application to unanticipated algorithm extension by showing how aspects can help in generalising a constant propagation strategy to a generic and adaptable forward propagation scheme using principles of invasive software composition [Ass03].

This chapter is organised as follows. The next section describes the running example of this chapter: a simple constant propagator. Section 5.3 introduces an extension to Stratego which allows separate declaration of cross-cutting concerns and shows how this extension facilitates declarative code composition. Section 5.4 discusses three cases where the aspect extension is found to be highly useful: logging, type checking of terms and algorithm adaptation. Section 5.6 discusses previous, related and future work. Section 5.7 summarises.

5.2 Constant Propagation

The code in Figure 5.1 shows an excerpt of a strategy for propagating constants applicable to an imperative language with assignment, `While` and `If` constructs. The example in Figure 5.2 illustrates the application of the constant propagator to a short program.

```
1  module prop-const
2  signature
3   constructors
4    Var : Id ⟶ Var
5          : Var ⟶ Exp
6    Int : String ⟶ Exp
7    Plus : Exp × Exp ⟶ Exp
8    If : Exp × Exp × Exp ⟶ Exp
9    While : Exp × Exp ⟶ Exp
10   Assign : Var × Exp ⟶ Exp
11 rules
12   EvalBinOp : Plus(Int(i), Int(j)) ⟶ Int(k)
13               where <addS>(i,j) ⇒ k
14   EvalIf : If(Int("0"), e1, e2) ⟶ e2
15   EvalIf : If(Int(v), e1, e2) ⟶ e1 where <gtS> (v, "0")
16 strategies
17   prop-const =
18     PropConst <+ prop-const-assign <+ prop-const-if
19     <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp))
20   prop-const-assign =
21     Assign(?Var(x), prop-const ⇒ e)
22     ; if <is-value> e then rules( PropConst : Var(x) ⟶ e )
23                       else rules( PropConst :- Var(x) ) end
24   prop-const-if =
25     If(prop-const, id, id)
26     ; (EvalIf ; prop-const <+
27        (If(id, prop-const, id) /PropConst\ If(id, id, prop-const)))
28   prop-const-while =
29     ?While(e1, e2)
30     ; (While(prop-const, id)
31        ; EvalWhile
32          <+ (/PropConst\* While(prop-const, prop-const)))
```

Figure 5.1: An excerpt of a Stratego program defining an intraprocedural conditional constant propagation transformation strategy for a small, imperative language.

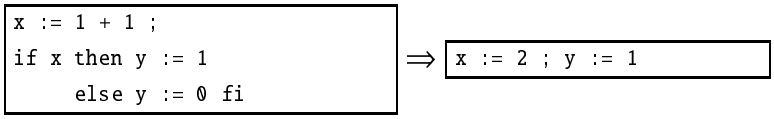

Figure 5.2: Example showing the constant propagation strategy.

The principle of the constant propagation algorithm is straightforward: A traversal through the abstract syntax tree (AST) of a program is done. Whenever an assignment of a constant value to a variable is encountered, this is recorded using a dynamic rewrite rule set named `PropConst`. If the variable is subsequently assigned a non-constant value, the corresponding rule is deleted. This is done in `prop-const-assign` defined on lines 19–22. The process results in rule set mapping variable names to their constant values.

The dynamic rule set is subsequently used to replace every variable with its value where this is known. This replacement opens up for the elementary evaluation rules `EvalBinOp` and `EvalIf` defined on lines 12–14. These rules simplify some expressions involving `Plus` and `If`, respectively.

The `prop-const` strategy on lines 16–18 is the top level driving strategy which takes care of recursively applying the constant propagation throughout a term. It works by calling the rule `PropConst` to replaces any variable term for which the value is known to be constant. If the `PropConst` rule fails, the current term is not a variable with a known constant. In that case, the strategy `prop-const-assign` is attempted. If the `prop-const-assign` is applied to an `Assign` term and the right hand side of the assignment evaluates to a constant, a new `PropConst` rule is generated. If the right hand side of the assignment is not a constant, any previously defined `PropConst` rule for this variable is removed since its value is no longer known. Should the strategy `prop-const-assign` fail, two other strategies are tried in order, namely `prop-const-if` and `prop-const-while`. These are described below. If all strategies fail, `prop-const` falls back to applying itself recursively to all subterms of the current term and finally try to apply the `EvalBinOp` rule (lines 12–13) on the result. This ensures that all language constructs, such as `Plus`, are traversed.

The `prop-const-if` strategy on lines 23–26 matches an `If` construct using the congruence operator while at the same time applying `prop-const` to the condition expression. If the congruence succeeds, the `prop-const-if` strategy proceeds by either (1) simplifying the `If` using the `EvalIf` rule and then recursively continuing the `prop-const` algorithm on the result, or (2) applying `prop-const` recursively to the then-branch and else-branch in turn, keeping only `PropConst` rules which are valid after both branches, i.e. those rules that are defined and equal in both branches.

Recall that the dynamic rule intersection operator `s1 /PropConst\ s2` used on line 26 applies both strategies `s1` then `s2` to the current term in sequence while distribut-

ing (clones of) the same rule set for the dynamic rule `PropConst` to both strategies. Afterwards, only those rules which are equal in both branches are kept. A similar explanation applies to `prop-const-while`, defined on lines 27–31, where the fixpoint operator `/PropConst\* s` is used. This operator applies `s` repeatedly until a stable rule set is obtained. Each iteration will apply `s` to the *original* term and the result of the final iteration is kept as the new term.

Generalisation and Adaptation

As written, the algorithm already has some extension points the user of the constant propagator may plug into without modifying the algorithm itself. For example, adding another evaluation rule for `EvalIf` that deals with non-zero constants is trivially possible. There are also other extensions and adaptations users may want to apply to this algorithm which are impossible to do without reimplementing the algorithm from scratch. Section 5.4.1 demonstrates that pervasive logging is one such example. Using aspects, it is possible to extend the code with logging capabilities to record all rule invocations. Section 5.4.2 shows another problematic case where pervasive (dynamic) type checking of terms to ensure the result is a correct term is desired. This is also handled easily with aspects. Finally, Section 5.4.3 shows how the algorithm can be refactored into a more generalised schema for forward propagating data-flow transformations. All extensions and adaptations are performed with the help of the aspect extension to the Stratego language described next.

5.3 AspectStratego

AspectStratego is an extension to the Stratego language which addresses the problem of declaring cross-cutting concerns in a modular way. The language extension bears some resemblance to the AspectJ language [KHH+01].

The reader is not assumed be familiar with AspectJ, but for readers familiar with AspectJ, some differences and similarities are remarked: Much of the terminology in this chapter is inherited from AspectJ. The joinpoint model of Aspect Stratego is somewhat similar to that of AspectJ, but has been adapted to fit better within the paradigm of rule-based rewriting systems. Both AspectJ and AspectStratego provide the programmer with expressions called pointcuts. In AspectStratego these are boolean predicates on the program structure unlike the set theoretic approach taken in AspectJ. Pointcuts are used to pick out well-defined points in the program execution, called joinpoints, and are available in 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.

```
1 module prop-const-logger
2 imports logging prop-const
3 aspects
4  pointcut call = strategies(prop-const)
5  aspect prop-const-logger =
6   before : call = log(|Debug, "Invoking constant propagator")
```

Figure 5.3: Aspect extending the constant propagation module with logging. The `log` is from `logging` module, part of the Stratego library.

Figure 5.3 shows how one may use an aspect to extend the constant propagator with trivial logging. It declares a pointcut, `call`, on line 4 that identifies all strategies named `prop-const`. The aspect on line 5–6 declares that before every joinpoint identified by `call`, the code fragment `log(...)` shall be inserted.

Section 5.4 will define the give a more advanced example of logging. Next, the new terminology and language features introduced in this example will be defined.

5.3.1 Joinpoints

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 purpose of the aspect language is allowing the programmer to precisely and succinctly identify and manipulate joinpoints.

5.3.2 Pointcuts

A *pointcut* is a boolean expression over a fixed set of predicates, defined in Table 5.1, and the operators ; (*and*), + (*or*) and `not`. Pointcuts are used to specify a set of joinpoints. There are two kinds of predicates in a pointcut: joinpoint predicates and joinpoint context predicates [1]. A *joinpoint predicate* is a pattern on the Stratego program structure used to pick out a set of joinpoints. A *joinpoint context predicate* is a predicate on the runtime environment which can be used in a pointcut to restrict the set of joinpoints matched by a joinpoint predicate.

Table 5.1 lists the supported joinpoint and joinpoint context predicates. A *pointcut declaration* is a named and optionally parametrised pointcut intended to allow easy sharing of identical pointcuts between advice. The parameters are used to expose details about the pointcut to the advice. The declaration `pointcut call = strategies(prop-const)` from Figure 5.3 shows a parameterless pointcut named `call`

[1] This terminology and implementation differs from the AspectJ language which provides *primitive pointcut designators* instead, see [KHH+01].

Joinpoint	*Matches*
`calls(`*name-expr* $\Rightarrow$ *n*`)`	strategy or rule invocations
`strategies(`*name-expr* $\Rightarrow$ *n*`)`	strategy executions
`rules(`*name-expr* $\Rightarrow$ *n*`)`	rule executions
`matches(`*pattern* $\Rightarrow$ *t*`)`	pattern matches
`builds(`*pattern* $\Rightarrow$ *t*`)`	term constructions
`fails`	explicit invocations of fail
Joinpoint context	*Matches*
`withincode(`*name-expr* $\Rightarrow$ *n*`)`	joinpoints within a strategy or rule
`args(`n_0`,`n_1`,...,`n_n`)`	joinpoints with given arity
`lhs(`*pattern* $\Rightarrow$ *t*`)`	rule left-hand sides
`rhs(`*pattern* $\Rightarrow$ *t*`)`	rule right-hand sides
Advice	*Action on joinpoint*
`before`	run advice before
`after`	run advice after
`after fail`	run after, iff code in pointcut failed
`after succeed`	run after, iff code in pointcut succeeded
`around`	run before and after
Cloning	*Action on declaration*
`clone` *kind name-expr* $\Rightarrow$ *name-expr*	clone and rename a named declaration

Table 5.1: Synopsis of the AspectStratego joinpoints, joinpoint context predicates and advice variants. The *name-expr* can either be a complete identifier name, such as `EvalBinOp`, a prefix, such as `prop-*`, a suffix, such as `*-assign` or an infix, such as `*-const-*`. The result of a *name-expr* is a string, and may optionally be assigned to a variable using the `=>` *x* syntax. The *kind* is either of the keywords `strategies`, `rules` or `overlays`. The *pattern* is an ordinary Stratego pattern, which may contain both variables and wildcards. When a *name-expr* is used for cloning, the literal parts may be replaced. I.e., `clone *-prop-* as *-myprop-*` will rewrite the middle part of the identifier.

with the joinpoint predicate `strategies` and no joinpoint context predicates. It picks out all definitions of strategies named `prop-const`.

5.3.3 Advice

An *advice* is a body of code associated with a pointcut. There are three main kinds of advice: *before*, *after* and *around*. The different forms of advice specify where the body of advice code should be placed relative to the joinpoint matched by its pointcut. Table 5.1 lists the available advice types for AspectStratego. The declaration on line 6 in Figure 5.3 is an example of a before advice. The strategy `log` is provided by the library, and will be discussed later. This code will be inserted – weaved – into the `prop-const` strategy from Figure 5.1 as follows:

```
prop-const = log(|Debug, "Invoking const....")
  ; (PropConst <+ prop-const-assign <+ prop-const-if
  <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp)))
```

Composing code by inserting advice like this opens up the possibility for manipulating the current term. Recall that all strategies and strategy expressions in Stratego are applied to the current term unless they are specifically applied to a variable or pattern with the application operator (`<s> x`). Exactly how the strategy or rule invocations inside the advice body changes the current term can be controlled in two ways: the advice body is a strategy expression and may be wrapped (entirely or partially) in a `where` to control how and if the current term is modified. In the above case, `log` only takes term arguments and is designed to leave the current term untouched, making `where` superfluous.

This manipulation of the current term turns out to be very useful in `around` advice where the implementer of the advice has full control over how the pointcut should be executed. The placeholder strategy `proceed` is available for this purpose. By placing the `proceed` within a `try` or as part of a choice (`+`), it is trivially possible to add failure handling policies. The flexibility of `around` allows the aspect programmer to completely override and replace the implementation of existing strategies and rules by not invoking `proceed` at all. This can even be applied to strategies found in the Stratego standard library.

The usefulness of current term manipulation stems from terms normally being passed via the current term from one strategy to another, not as term arguments. E.g., in the following example, `strat2` will be applied to the current term left behind by `strat1`:

```
strat1 ; strat2
```

An alternative, more imperative formulation of the same would be:

```
strat1 ⇒ r ; strat2(|r)
```

This is not within the style of Stratego as it becomes cumbersome to use when one replaces sequential composition (;) with the other strategy combinators, such as left choice (<+). Current term manipulation is thus mostly analogous to manipulating input parameters and return values in AspectJ.

A note about rule and strategy priority is warranted. Aspects may be used to modify an existing rule (or strategy), but there is no mechanism by which aspects can directly change the priority of a rule or an aspect.

5.3.4 Cloning

A very useful feature provided by AspectStratego is the ability to clone existing named definitions, such as rules, strategies or overlays. For example, the declaration below will clone all the strategies starting with the name `prop-*` and, for each, create an identical copy with the `my-prop-*` name prefix (the matching value of `*` will be expanded, of course).

```
clone strategies prop-* ⇒ my-prop-*
```

The flat structure of Stratego, with only one global name space for all definitions, makes cloning straightforward. It is allowed, but generally discouraged, to give the clone a name which conflicts with existing definitions. In Stratego, multiple strategies or rules may have the same name, so cloning with a conflicting name must remain allowed – it is sometimes what the developer intends.

Using the cloning feature, it becomes possible to rewrite the pointcuts to apply to clones strategies, i.e. to `my-prop` instead of `prop`. Cloning enables aspects to instantiate multiple *concurrent* variants of existing library functionality in the same program. This allows existing language-specific transformations to be adapted for new subject language signatures. New signatures may introduce additional language constructs. Support for these constructs may be added to an existing (potentially cloned) transformation using the techniques for unanticipated algorithm adaptation discussed later.

The cloning feature was, to the knowledge of the author, first proposed (for Java) in [BBK+05].

5.3.5 Weaving

The pointcuts are designed to be evaluated entirely at compile-time. All cloning declarations are evaluated and resolved before any pointcuts are matched. Given an advice declaration, the compiler will interpret its pointcut declaration on the Stratego abstract syntax tree (AST) to find the location where the advice body must be weaved. The code in the advice body is then inserted into the AST before, after or around the joinpoint.

The body of the advice has a rudimentary reflective capability, which is also resolved at compile-time. Table 5.1 indicates that the advice body has access to rule and strategy names. The Stratego runtime has no reflective nor code-generating capabilities so these names are mostly useful for logging purposes. Advice body code also has access to patterns from match expressions, and may evaluate these patterns at runtime. This is demonstrated in Section 5.4.2.

5.3.6 Modularisation

All AspectStratego code, including aspects, must reside in modules. This seems sensible, since the goal of aspects is to modularise cross-cutting concerns. An `aspect` or `pointcut` can only be declared within an `aspects` section of a module. This is similar to how for example overlays must reside in the `overlays` section. While `aspects` sections may be interleaved with the other Stratego sections (e.g., `strategies`, `rules`, `signature`), it is encouraged that each aspect is declared in a separate module. First, this helps keep aspects – separate, cross-cutting concerns – truly separate, both in design and implementation. Second, this also allows them to be selectively enabled or disabled using compiler flags without any code modification at all. Modules may be substituted in the build system without source code modification. The mechanisms and language features required for controlling the application of aspects on the module level are still subject to research; it is currently not possible to restrict the application of aspects based on the module of a joinpoint.

AspectStratego keeps the pointcut declarations outside the aspect declarations, to signify that pointcuts may be shared between aspects. In object-oriented renditions of aspects, such as AspectJ, sharing of pointcuts between aspects is captured using inheritance: a subaspect inherits all pointcuts from its superaspect. The usefulness of shared pointcuts are demonstrated in Section 5.4.3.

5.4 Case Studies

This section motivates the use of AspectStratego with three case studies relevant to rule-based programming. The first example is a simple logging aspect which is included to show similarities and differences with the AspectJ language. The second is a dynamic type checker of terms realised entirely as an aspect. It shows how aspects may sometimes be used as an alternative to compiler extensions. The final case is a discussion of how aspects may be useful in expressing variation points when implementing generalised adaptable algorithms.

```
module simple-logger
strategies
 invoked(|s) = ![ "Rule '", s, "' invoked" ]
aspects
 pointcut log-rules(n) = rules(* ⇒ n)
 aspect simple-logger =
  before : log-rules(r) = log(|Debug, <invoked(|r)>)
  after fail : log-rules(r) = log(|Debug, <failed(|r)>)
  after succeed: log-rules(r) = log(|Debug, <succeeded(|r)>)
  after : log-rules(r) = log(|Debug, <finished(|r)>)
```

Figure 5.4: A complete logging aspect in AspectStratego. The definitions of `failed`, `succeeded` and `finished` are similar to `invoked`. The direction of information flow through the pointcut declaration arguments is somewhat uncommon: they specify information going *out* of the declaration.

5.4.1 Logging

Logging of program actions is often useful when developing software and is therefore a problem one wants to encode in a concise fashion. The program points one wants to trace frequently follow the program structure, for instance, the entry and exit of functions. In these cases, the established solution is to wrap the function definitions in syntactical or lexical macros which perform simple code composition. The numerous shortcomings of this technique, such as decreased code readability, lack of flexibility, interference with meta-tools (especially for documentation and refactoring) and typographic tedium, are all addressed by aspects. The aspect language also allows pervasive insertion of logging code in locations unanticipated by the original implementer such as inside rule conditions and failures deep inside the Stratego library.

The code in Figure 5.4 shows an aspect, called `simple-logger`, that may be used to insert logging code around all rules in a program by adding it to the `imports` list of the main module. The code transformations induced by the weaving are detailed in Section 5.5.

While the built-in `log` strategy provides the ability to set the logging level at runtime (e.g. only errors, and no warning and debug messages), a program with explicit `log` calls inserted into its strategy and rule definitions will always take a slight performance hit. Stratego, where the coding style encourages many and small rules and strategies, is sensitive to any such overhead even with aggressive inlining. Consequently, it is desirable to have the ability to easily remove most or all `log` calls before final deployment. Aspects make this trivial.

The application of one aspect may open up for further adaptation by another aspect. For example, the strategy `invoked` in Figure 5.4 may be the target for further aspects. Note that these second level — or "meta" — aspects pose a few potential problems with respect to weaving order that have not been resolved in the implementation yet. In the current implementation, aspects are weaved in the order of declaration. Consider the following definition of `ext-invoked`:

```
aspects
 pointcut invoked = calls(invoked)
 aspect ext-invoked =
  before : invoked = ...
```

If this aspect were to be weaved before `simple-logger`, it would have no effect, as `invoked` is not called anywhere at the time `ext-invoked` is weaved. As long as the user is aware of this, and manually linearises the dependency chain between aspects by declaring `ext-invoked` after `simple-logger`, the result will match the intention of the user.

5.4.2 Type Checking

Terms in Stratego are built with constructors from a signature, but the language does not enforce a typing discipline on the terms: it is a single-sorted rewriting language. Given the signature in Figure 5.1, a Stratego program may construct an invalid term, e.g. `!Plus(Int("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. To debug such problems, it is common to manually insert debug printing, or weave in a logger to generate a program trace for manual inspection. This form of manual verification is highly error-prone.

The Stratego/XT environment comes with format checking tools for this purpose. The tools can be applied to the resulting term of a Stratego program, checking it against a given signature. While all signature violations are caught by these tools, they cannot help in telling where in your program the actual problem is present as the check happens entirely after program execution. It is possible to use aspects to weave the format checker into the rules of our program at precisely the spots where one would like the structural invariants to hold. The `typechecker` aspect in Figure 5.5 makes use of the format checker functionality in Stratego/XT to pervasively weave format checking into all rules of a Stratego program. By modifying the `typecheck-rules` pointcut, the user can control the exact application of the type checker. Its usage is similar to the `simple-logger`: it must be imported, but, in addition, a `typecheck` strategy for the relevant signature must be declared in a `strategies` section:

```
typecheck(|t) = format-check-Imp(|t)
```

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

Figure 5.5: An aspect for weaving simple dynamic type of terms into rules.

Given the signature in Figure 5.1, the Stratego/XT format checker tools generate a Stratego module containing a complete format checker for that signature. The top level strategy for this format checker is named `format-check`. It may be applied to a term and checks if it is a valid (sub)term of that signature.

As with logging, introducing the checking aspect provides the user with a quick and concise mechanism to decide which parts (if any) of a program should be type checked. Its usage can be toggled both at compile- and runtime (the latter always incurs a small performance hit as previously discussed).

The aspect Figure 5.5 invokes the `typecheck` strategy. The argument t to `typecheck` is the pattern matched by the `typecheck-rules` pointcut, i.e. the pattern is extracted from the right-hand side pattern of a rule. In the case that `t` is a term (no variables), it can in theory be entirely checked at compile time as both the signature and the term are completely known to the compiler. In the case that t contains variables, the static parts may be checked at compile time, but the variable part must be evaluated at runtime.

The type checking aspect is only a partial replacement for a built-in type system. It performs no type inferencing and can therefore not eliminate redundant checks. The topic of typed, strategic term rewriting is discussed in [Läm03].

5.4.3 Extending Algorithms

The algorithm in Figure 5.1 is an instance of the more general data-flow problem of forward propagation examples of which are common subexpression elimination, copy propagation, unreachable code elimination and bound variable renaming. The algorithm can be factored into a language-specific skeleton and problem-specific extensions. The language-specific skeleton must account for control-flow constructs and scoping rules specific to a given language. In some cases, it is possible to abstract over subject language differences. Using aspects, additional flexibility is provided and the skeletons may more easily be reused for similar subject languages. Cases for additional language constructs may easily be added to the skeleton using before advice,

and non-applicable cases may be voided using around advice.

A *variation point* is a concrete point in a program where variants of an entity may be inserted. By providing clearly defined variation points, the skeleton is made adaptable to the specific propagation problem at hand.

Expressing Adaptable Algorithms

There are many well-known techniques for expressing adaptable algorithms. When providing an algorithm intended for reuse and adaptation by other programmers (users), the following properties of the technique become important:

- *adaptability*; one would like maximal freedom in which variation points one exposes to the users.
- *reuse*; the users of the algorithm should need to reimplement as little code as possible. This is especially important in the face of maintenance.
- *traceability*; when errors (either in the design or the implementation) are discovered in the algorithm, users should be offered an easy upgrade path. Ideally, the users should only need to replace the library file wherein the algorithm resides. This may not always be feasible, but, at the very least, the users should know which parts of their system may be affected by the error.
- *evolution*; one must be able to change the internals of the algorithm without disturbing the users.

Boilerplates One of the most popular, but least desirable, techniques for adaptation is *boilerplate* adaptation. In this approach, a code template is manually copied then modified to fit the situation at hand. The approach suffers from high maintenance costs due to inherent code duplication. It is especially problematic if the original template is later found to contain grave (security) errors since there is no traceability of where it has been used. On the other hand, it offers a very high degree of flexibility as all variation points may be reached. At its most extreme, boilerplate adaptation allows the applicant to gradually replace the entire algorithm.

Design Patterns Another, popular technique for reuse is the use of *design patterns* [GHJV95]. A design pattern is a piece of reusable engineering knowledge. For every case where a design pattern is applicable, it must be implemented from scratch by the programmer. In the recent years, much research has been into improving reuse of design patterns, either by providing direct language support [Bos98, Hed98] or by placing them in reusable libraries [AC98, HK02].

Hooks and Callbacks *Hooks* and *callbacks* are well-known techniques for exposing variation points through *overridable* stubs the user of a library or algorithm can extend. By calling registration functions, the user may add callbacks and hooks which are called at pre-determined locations in the algorithm or upon particular events in the program. As long as the contract between the algorithm and its callbacks is maintained, the algorithm internals may evolve separately from the adapted hooks and therefore offers good maintenance properties. Its drawbacks include the fact that not all variation points may be expressed as hooks, and that it is difficult to adapt an algorithm with different sets of hooks in multiple contexts within the same program. In Stratego, this can to some degree be solved using scoped dynamic rules. For other paradigms, function pointers, closures and/or objects allow multiple contexts to exist.

Higher-order Parameters In functional languages, it is common to expose variation points through *higher-order parameters*. The paper [OV05] describes an adaptable skeleton for forward propagation using this approach. The technique provides a precise way for exposing variation points which is both easy to use and allows the user to adapt the algorithm on a per-context basis within the same program. One drawback is the issue of "parameter plethora", i.e. the number of parameters users must deal with. In cases where the problem space is large, the algorithm often has many variation points yielding a long parameter list. A common solution to this problem is providing multiple entry points into the algorithm, each with an increasing number of parameters. Another is having parameters with default values where the language supports this.

Limitations

Boilerplates and design patterns are not really desirable given their poor support for code reuse and traceability. While the last two solutions discussed above offer both good reuse and traceability, they suffer from a few additional drawbacks. Over time, experience with the use of an algorithm may expose a need to extend it with further variation points unanticipated by the original implementer. Exposing a new variation point frequently results in a change in the algorithm interface, through the adding new higher-order parameters, hooks or new parameters to existing hooks. Backwards compatibility can normally be handled by writing wrappers mimicking the old interface which forwards to the new. The price is the burden of maintaining multiple versions of the same interface.

Another consideration when extending an algorithm is how to propagate the new variation point through its internals. Suppose in `prop-const` (Figure 5.1), one wanted to add the ability to transform the current term before recursively descending into the children. With a solution based on higher-order parameters, this transform parameter would have to be "threaded" through all `prop-*` strategies as a higher-order parameter,

and thus result in an intrusive rewrite.

A final consideration is who should be able to perform adaptation and extension of existing algorithms. It is normally not possible for the user of the algorithm to extend it outside the exposed variation points even if they can be clearly identified, unless the user has access to the source code, in which case the boilerplate technique may be resorted to.

Dealing with Evolution

This section demonstrates a solution to the extensibility problem for handling *unanticipated* variation points that is complementary to hooks and higher-order parameters. It uses the declarative features of aspects to clearly identify and name the variation points in the algorithm. The code in Figure 5.6 shows how some of the variation points already discussed have been exposed through pointcuts. The algorithm provider may decide to add some trivial points, `fail` in `forward-prop` and `id` in `prop-assign` to allow the pointcuts and advice to be expressed more clearly. These are not strictly necessary. The same joinpoints can be identified and used with only slightly more complicated pointcuts and advice and also by a user of the skeleton without involving the provider nor changing the code.

The `forward-prop` pointcut may be used to insert the transformation code before and after the propagator visits subterms of a given term. The `prop-*` pointcuts may be used similarly for inserting code before and after recursive descent into subterms of their respective language constructs. The `prop-rule` pointcuts are used for declaring which dynamic rule(s) to use for intersections and during traversal. Note that the pointcuts have the same names as the strategies they match inside. This makes it very clear to the user where the advice is applied. Admittedly, this is also a potential source of confusion as the same identifier may refer to either an aspect or a strategy/rule, depending on context. The pointcut namespace is kept separate from the other namespaces in Stratego (one for rules and strategies, another for constructor names) because the namespaces in Stratego are global and one-level. There is no hierarchy of namespaces (c.f. Chapter 2, Section 2.4.1).

By using these variation points exposed through aspects, the code in Figure 5.7 demonstrates how the skeleton may be instantiated with advice to obtain a constant propagator. After weaving, the result is the exact algorithm presented in Figure 5.1. `around` advice is used instead of `after` advice to properly parenthesise the expressions and control operator precedence. Consider the weaving of the `around` advice for `prop-while` pointcut. The pointcut matches the following joinpoint code:

```
/\* While(forward-prop,forward-prop)
```

By using the around advice, this expression is replaced with:

```
(While(forward-prop,id); EvalWhile <+ proceed)
```

```
module forward-prop
strategies
 forward-prop =
   fail <+ prop-assign <+ prop-if <+ prop-while
     <+ all(forward-prop)
 prop-assign =
   Assign(?Var(x), forward-prop ⇒ e) ; id
 prop-if =
   If(forward-prop, id, id)
   ; (If(id,forward-prop,id) /\ If(id,id,forward-prop))
 prop-while =
   ?While(e1, e2)
   ; (/\* While(forward-prop, forward-prop)))
aspects
 pointcut prop-rule(r) =
   (calls(dr-fork-and-intersect) ; args(_, _, r))
   + (calls(dr-fix-and-intersect) ; args(_, r))
 pointcut prop-rule = fails ; withincode(forward-prop)
 pointcut forward-prop = calls(all) ; withincode(forward-prop)
 pointcut prop-assign = calls(id) ; withincode(prop-assign)
 pointcut prop-if =
   calls(dr-fork-and-intersect) ; withincode(prop-if)
 pointcut prop-while =
   calls(dr-fix-and-intersect) ; withincode(prop-while)
```

Figure 5.6: Skeleton for forward propagation with variation points exposed as pointcuts. For a real language, the skeleton is often quite large and often difficult to construct. `s1 /Rule\ s2` is syntactic sugar for `dr-fork-and-intersect(s1, s2 | [ "Rule" ])`, and `/Rule\*` is sugar for `dr-fix-and-intersect`. In the above code, the `Rule` will be filled in later by aspects, thus the empty fork (`/\`) and fix (`/\*`) in `prop-if` and `prop-while`, respectively.

Since `proceed` invokes the original (matched) joinpoint code, the end result is the same code as found in `prop-const-while` in Figure 5.1, modulo the fact that the driving strategy is now named `forward-prop`. Using cloning and renaming, it is possible to derive a practically identical implementation. Additionally, the patterns and traversals may be adapted for additional signatures, thus easily instantiating the forward propagator for a family of subject languages.

Evaluation

The proposed solution is now evaluated based on the criteria set out above.

Adaptability Exposing variation points through pointcuts is more adaptable than higher-order parameters and hooks because it can be done without changing the algorithm itself. As long as the variation point can be picked out using a pointcut, an advice may be used to insert a callback into the algorithm at that point. This is easier with AspectStratego than many other aspect extensions since the data normally is passed through the algorithm as the *current term*. It is possible to use pointcuts to modify the current term before or after any strategy or rule invocation in the algorithm implementation. Aspects can be viewed as a complementary extension mechanism to callbacks/hooks since it may be used to add these. Similarly, the aspect technique is complementary to higher-order parameters. It is also possible to wrap the entry point to the algorithm in a reparametrised strategy.

Different levels of adaptability may be exposed using aspects. These these levels are expressed separately from the algorithm skeleton. By choosing between the available adaptation aspects, the user may select which sets of variation points he intends to deal with. Assuming white-box reuse, the user may add new variation points to the algorithm in this fashion.

Reuse Compared to design patterns and boilerplates, much better reuse is obtained. With a properly designed skeleton, the amount of code needed to adapt the algorithm is proportional to the extra functionality added.

Traceability It is directly evident from the code both which version of the skeleton that has been used and how it has been adapted (using which aspects). Traceability is therefore better than for boilerplates and patterns, and at the same level as parameters and callbacks.

Evolution As time goes by, new callbacks and higher-order parameters may easily be added to the skeleton using aspects. Further, aspects may be used internally to propagate the parameters to all sub parts of the algorithm implementation. Arguably, extra care must be taken to ensure that the semantics of the pointcuts are kept after

```
module forward-prop-usage
imports forward-prop
aspects
 aspect prop-const =
  around : prop-rule(r) = proceed([ "PropConst" ])
  around : prop-rule = PropConst
  around : forward-prop = (proceed ; try(EvalBinOp))
  before : prop-assign-ext =
    ?Assign(Var(x), e)
    ; if <is-value> e
       then rules(PropConst: Var(x) → e)
       else rules(PropConst:- Var(x)) end
  around : prop-if = EvalIf ; forward-prop <+ proceed
  around : prop-while =
    (While(forward-prop,id) ; EvalWhile <+ proceed)
```

Figure 5.7: Instantiation of the forward-prop to make the constant propagator in Figure 5.1, using aspects. Admittedly, the example is somewhat contrived, as these are variation points we normally would anticipate and explicitly parameterize easily.

an algorithm revision since they now are declared separately. This problem is no different from variation points exposed through higher-order parameters or hooks as long as the pointcuts are known to the revising party.

In the case where users have identified and extended variation points through their own pointcuts, the situation is more precarious. This is a known drawback of white box reuse.

A particularly attractive feature of aspects in the context of this dissertation is the way they enable the expression of language independent transformations. General algorithm skeletons may be implemented and adapted invasively when they are instantiated for new subject languages. To some extent, existing language-specific transformations may in some cases also be adapted to other, similar languages.

5.5 Implementation of the Weaver

The aspect weaver for AspectStratego is realised entirely inside the Stratego compiler as one additional stage in the front-end. It operates on the normalised AST where the module structure has been collapsed. All definitions from all included modules are thereby available for weaving. The weaver is implemented as traversals on the AST. The full pipeline for compiling – or weaving – the aspect extension into Stratego is

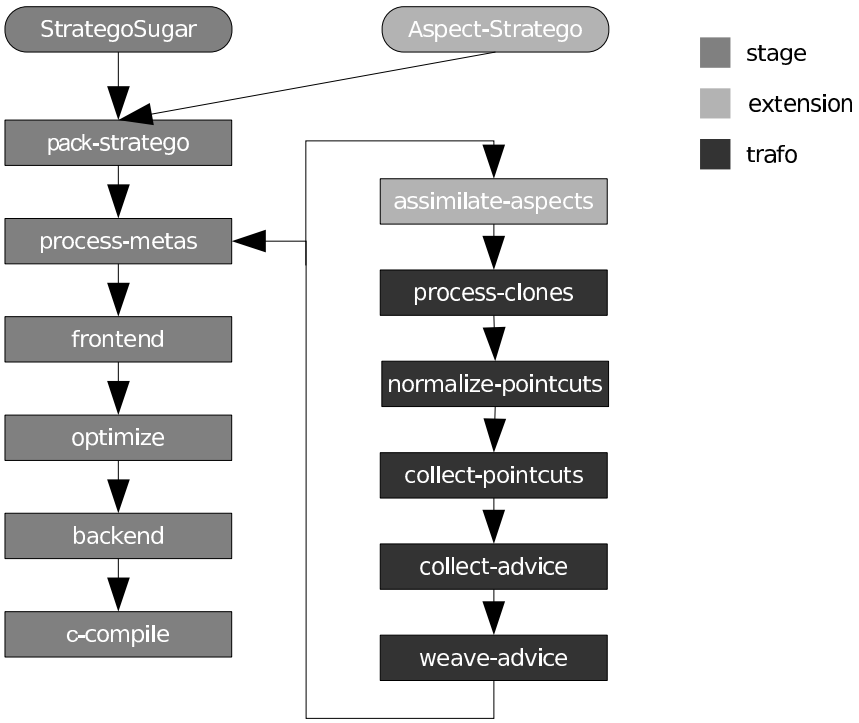

Figure 5.8: Pipeline for assimilating AspectStratego into StrategoCore

shown in Figure 5.8.

Cloning – All clone expressions are collected and the relevant definitions are duplicated and renamed in two-pass top-down traversal called `process-clones`. First, all the clone expressions are collected and then to all matches are found and duplicated.

Pointcut collection and evaluation (`collect-pointcuts` in the figure) is a top down traversal that collects all pointcut declarations. Every pointcut encountered is essentially a simple logical expression. These expressions are decomposed into conjunctive normal forms called *fragments*. A fragment contains one joinpoint 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)`. For each named pointcut, a dynamic rule is generated and used as a map from pointcut names to the fragment set for that pointcut.

Advice collection and evaluation (`collect-advice`) is a top down traversal that collects all advice declarations. When an advice is encountered, its associated pointcut is looked up and one dynamic rule is generated for each fragment of that pointcut. In a generated rule, the left-hand side matches the term in the Stratego AST corresponding to the fragment's joinpoint predicate. For example, `rules` match against the AST term for rule declaration, `RDefT`. The generated rule evaluates all joinpoint

```
EvalBinOp =
  log(|Debug, invoked("EvalBinOp")) ;
  if shadowed-EvalBinOp then
    if log(|Debug, succeeded("EvalBinOp")) then
      try(log(|Debug, finished("EvalBinOp")))
      else
      log(|Debug, finished("EvalBinOp")) ; fail
    end
  else
     // identical to the then-clause,
     // with succeeded replaced by failed
  end
```

Figure 5.9: The declaration of `EvalBinOp` from Figure 5.1 after weaving in the `simple-logger` aspect.

context predicates in its condition. These rules are applied later by the weaver. When a rule succeeds, it provides the weaver with the context information identified by its pointcut fragment and the advice body associated with that pointcut.

Weaving – The actual weaving is a bottom up traversal of the AST (`weave-advice`). It exhaustively attempts to apply all generated advice rules from the previous step. On any term where one or more rules match, their associated advice bodies are collected and applied in place.

5.5.1 A Weaving Example

By weaving the `simple-logger` aspect (Figure 5.4) into the module in Figure 5.2, all executions of `EvalBinOp` and `EvalIf` are logged. Weaving of this aspect on the rule `EvalBinOp` proceeds as follows.

First, the weaver shadows both declarations by adding a new unique prefix to the existing rule name. Then, a wrapper strategy from the template in Figure 5.10[2] is instantiated. It has the name of the original rule (`EvalBinOp`). The final result of this weaving for `EvalBinOp` is shown in Figure 5.9. The wrapper first executes the code from the before advice followed by the shadowed (original) code. If the shadowed code fails, the `after fail` advice is run followed by the `after` advice. The enclosing `try` and `if-then-else` are there to allow `after fail` and `after succeed` advice to change a failure into success or success into failure, respectively. `after` advice may not change failure/success but may replace the current term.

[2]Technically, the actual implementation uses the guarded choice operator. For readability reasons, the `if-then-else` is shown in the examples.

```
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
```

Figure 5.10: Template for advice weaving. Cursive identifiers are insertion sites for advice code. If a particular advice is not present in a joinpoint, it is replaced by an `id` (`after-fail` is replaced by `fail`).

5.5.2 Aspects as Meta Programs

When evaluating the pointcuts in the aspect compiler, there is a need to do interpretation of the pointcut expressions. This is realised as interpretive dynamic rules in the current implementation. Unfortunately, this leads to a rather rigid and tangled implementation where extending the language with new joinpoint (context) predicates becomes needlessly complex. It is conceptually much more appealing to view each advice as a small meta program to be executed on the AST. This meta program must be constructed at compilation time and can therefore not be a fixed part of the compiler. The current implementation can be seen as a manual specialisation of such a meta program where the dynamic parts are captured by dynamic rules.

Instead of inventing and maintaining a new interpreter for such meta programs, it is desirable to generate a small Stratego program for every meta program. This would be possible in a rewriting language with an open compiler or in a flexible, multi-staged language. The weaver would generate compiler extensions (meta programs), then execute these as part of the compilation process. This is now possible with the MetaStratego infrastructure, but the weaver has not yet been updated to take advantage of these developments.

5.6 Discussion

There are several documented examples of cross-cutting concerns found in the domain of rule-based programming. For example, the problem of origin tracking is documented in [vDKT93] and the problem of rewriting with layout in [vdBV00]. Both papers present interpreter extensions as the solution to their respective problems.

In [KL03], it is argued that both the above cases are instances of the more general problem of propagating term annotations – a separate concern which should be adaptable by the programmer. The solution proposed in [KL03] is to provide the programmer with declarative *progression methods* expressed as logic meta-programs.

It is realised as a research prototype in Prolog. The aspect extension also provides a mechanism for specifying cross-cutting concerns in a declarative and adaptable way, but the style proposed herein is very similar to the popular AspectJ language, although recast for Stratego.

Many other aspect extensions have been documented. The AspectS system for the Squeak dialect of Smalltalk [Hir03] describes a weaver which works entirely at runtime using the reflective features of the Smalltalk runtime environment. The Casear aspect extension for Java [MO03] brings runtime weaving to Java. In [LK97], the authors describe a small object-oriented language Jcore and its extension Cool for expressing coordination of threads. The two are composed using an aspect weaver. AspectC++ [SGSP02] is an aspect extension to the C++ language. An aspect extension for the functional language Caml is described in [HT05]. In [MRB+], the authors document an aspect extension to the GAMMA transformation language for multiset rewriting and demonstrates its use to express timing constraints and distribution of data and processes. AspectStratego attempts to solve many of the same problems as the languages above because these problems are found in many languages. In addition, this chapter also motivates how problems specific to rule-based programming, such as language independence, may have solutions based on aspects.

The implementation of aspect-weavers using rewriting has been documented in [AL00] for the context of graph rewriting and in [GR04] using term rewriting. In both cases, the subject languages were object-oriented. In [Läm99], the authors detail an aspect language for declarative logic programs with formally described semantics, and a weaver based on functional meta-programs. Reflective languages with meta programming facilities such as Maude [CDE+03] are alternative implementation vehicles for aspects. The appeal of aspects is their concise, declarative nature with their clearly defined goal: identify joinpoints for inserting code. This contrasts with the flexibility and complexity offered by general meta programming. The "general-purposeness" of meta programming may in fact often be a hindrance to users. Distilling the power of general meta programming into a concise, declarative aspect language may therefore be worthwhile. While this chapter also describes the implementation of an aspect weaver using a term-rewriting system, the subject language, Stratego, is not a declarative logic nor an object-oriented language. This gives rise to a different set of joinpoints than considered by the above references.

The algorithm extension technique described in Section 5.4.3 is an example of compile-time code composition and is thus somewhat related to techniques such as templates in C++. Unlike C++ templates, the AspectStratego composition language is purely declarative and new variation points can be exposed retroactively without reparameterizing.

5.7 Summary

This chapter presented an aspect extension for the Stratego term-rewriting language, combining the paradigms of aspect-oriented programming and strategic programming. The implementation of this language was discussed. Several examples of is applicability were given, including a flexible dynamic type checker of terms as a practical example of aspects as an alternative to the interpreter extensions in [vDKT93] and [vdBV00]. The chapter also demonstrated how aspects may be helpful in handling unanticipated algorithm extension using the technique of invasive software composition. This form of (potentially retroactive) parametrisation increases the genericity of (existing) implementations, and thereby improves language independence. Aspects may be regarded as a declarative mechanism for adding support for subject language families to transformation libraries and are therefore an attractive language abstraction for language independent transformations.