Proceedings of the First Domain-Specific Aspect Languages Workshop

ACM International Conference on Generative Programming and Component Engineering (GPCE 2006)

October 22-26, 2006
Portland, Oregon

Technical Report VUB-PROG-TR-06-33
Vrije Universiteit Brussel

Foreword

Although the majority of work in the AOSD community focuses on general-purpose aspect languages (eg. AspectJ), seminal work on AOSD proposed a number of domain-specific aspect languages, such as COOL for concurrency management and RIDL for serialization. A growing trend of research in the AOSD community is returning to this seminal work, motivated by the known advantages of domain-specific approaches, as argued by Mitchell Wand in his keynote at ICFP 2003. This workshop is conceived for researchers who are further exploring the area of domain-specific aspect languages, including language design, enabling technologies and composition issues.

This volume contains 6 papers and abstracts of invited talks. We hope the reader will find them useful for advancing their understanding of some issues in concerning the design of aspect languages.

The organizing committee:
Éric Tanter (University of Chile, Chile)
Thomas Cleenewerck (Vrije Universiteit Brussel, Belgium)
Johan Fabry (Vrije Universiteit Brussel, Belgium)
Anne-Françoise Le Meur (University of Lille, France)
Jacques Noyé (École des Mines de Nantes, France)

Technical Papers

A. H. Bagge, K. T. Kalleberg. *DSAL = library + notation: Program Transformation for Domain-Specific Aspect Languages*. ..... *Page 4*

J. Fabry, N. Pessemier. *KALA: A Domain-Specific Solution to Tangled Aspect Code*. ..... *Page 12*

D. Rebernak, M. Mernik, H. Wu, J. Gray. *Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammar*. ..... *Page 19*

S. M. Watt. *Post Facto Type Extension for Mathematical Programming*. ..... *Page 26*

Position Papers

Y. Coady, C. Gibbs, M. Haupt, J. Vitek, H. Yamauchi. *Towards a Domain-Specific Aspect Language for Virtual Machines*. ..... *Page 33*

D. Hutchins. *Partial Evaluation + Reflection = Domain-Specific Aspect Languages*. ..... *Page 37*

Submitted to DSAL'06

DSAL = library+notation:
Program Transformation for Domain-Specific Aspect Languages

Anya Helene Bagge Karl Trygve Kalleberg
University of Bergen, Norway
{anya,karltk}@ii.uib.no

Abstract

Domain-specific languages (DSLs) can greatly ease program development compared to general-purpose languages, but the cost of implementing a domain-specific language can be prohibitively high compared to the perceived benefit. This is more pronounced for narrower domains, and perhaps most acute for domain-specific aspect languages (DSALs).

A common technique for implementing a DSL is writing a software library in an existing programming language. Although this does not have the same syntactic appeal and possibilities as a full implementation, it is a technique familiar to most programmers, and it can be done cheaply compared to developing a full DSL compiler. Subsequently, the desired notation may be implemented as a simple syntactic preprocessor. The cross-cutting nature of DSALs, however, makes it difficult to encapsulate these in libraries.

In this paper, we show a technique for implementing a DSAL as a *library+notation*. We realize this by implementing the library in a program transformation system and the notation as a syntactic extension of the subject language. We discuss our experience with applying this technique to multiple kinds of DSALs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.11 [*Software Architectures*]: Langauges, Domain-specific architectures; D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators

General Terms Design, Languages

Keywords Domain-Specific Aspect Languages, Aspect-Orientation, Program Transformation

1. Introduction

The implementation of domain-specific abstractions is usually done by way of libraries and frameworks. Although this provides the semantics of the domain, it misses out on good notation and many optimisation opportunities. Implementing domain specific languages by adding notation (syntax) to a library, and then programming a simple compiler that translates from the notation into equivalent library calls is an easy and powerful technique, which is cost-effective in many larger domains. Both the libraries and the simple compiler can be implemented in general purpose languages without too much effort, and it is important to note that the library need not be implemented in the same language as the compiler. If the "library" language supports syntax macros, like Scheme [16], or has a sufficiently powerful meta-programming facility, like C++ templates [1], the translation task may be accomplished through the inherent meta-programming constructs of this language. Otherwise, a stand-alone preprocessor is commonly used. For example, adding complex numbers or interval arithmetic to Java, with an appropriate mathematical notation, can be accomplished by writing or reusing a Java library, and writing a simple translator from the mathematical notation into OO-style calls. The approach of adding notation to (object-oriented) libraries was explored in the MetaBorg project [12], where the subject language Java was extended in various ways using Stratego as the meta-programming language.

For domain-specific aspect languages, the translation story is different. Behind the notation visible to the programmer lie cross-cutting concerns which may reach across the entire program, possibly requiring extensive static analysis to resolve. The straight-forward translation scheme into library calls for the subject language is not applicable as we are no longer dealing with basic macro expansion. Instead, we shall view aspects as meta-programs that transform the code in the base program. These meta-programs may be implemented with transformation libraries in a transformation language (which may be different from the subject language). This allows us to consider DSALs as syntactic abstractions over transformation libraries, analogous to the way DSLs are syntactic abstractions over base libraries in the subject language. That is, we do not translate the DSAL notation into library calls in the subject language, but rather to library calls in the transformation language. Provided that the transformation language has a sufficiently powerful transformation library for the subject language, writing a transformation library extension for a domain-specific aspect is an easy task. We will demonstrate this technique by example, through the construction of *Alert*, a small error-handling DSAL extension to the Tiny Imperative Language (TIL).

The main contributions of this article are: A discussion of how the *library + notation* method for DSLs can be applied to DSALs, if the library is implemented in a meta-language; an example of the convenience of employing a program transformation language in the implementation of DSALs, compared to implementation in a general-purpose language;

and a discussion of our experience with this technique for several different subject languages and aspect domains.

The paper is organised as follows. We will begin by briefly introducing our DSAL example and the TIL language (Section 2), before we discuss the implementation of our DSAL using program transformation (Section 3). Finally, we discuss our experiences and related work (Section 4), then offer some concluding remarks (Section 5).

2. The Alert DSAL

Handling errors and exceptional circumstances is an important, yet tedious part of programming. Modern languages offer little linguistic support beyond the notion of exceptions, and this language feature does not deal with the various forms of cross-cutting concerns found in the handling of errors, namely that the choice of how and where errors are handled is spread out through the code (with `if`s and `try`/`catch` blocks at every corner), leading to a tangling of normal code and error-handling code. Also, the choice of how to handle errors is dependent on the mechanism by which a function reports errors—checking return codes is different from catching exceptions, even though both may be used to signal errors. Confusingly, even the default action taken on error depends on the error reporting mechanism, from ignoring it (for return codes and error flags) to aborting the program (exceptions).

The Alert DSAL allows each function in a program to declare its *alert mechanisms*—how it reports errors and other exceptional situations that arise, and allows callers to specify how alerts should be handled (the *handling policy*), independent of the alert mechanism. We use the word *alert* for any kind of exceptional circumstance a function may wish to report; this includes errors, but may also be other out-of-band information, such as progress reports. Typical examples of alert mechanisms are exceptions, special return values (commonly `0` or `-1`) or global error flags (`errno` in C and POSIX, for instance). Ways of handling alerts include substituting a default value for the alerting function's return code; logging and continuing; executing recovery code; propagating the alert up the call stack; aborting the program, or simply ignoring the alert.

The alert extension is a good example of a domain-specific aspect language. It allows separation of several concerns: the mechanism (how an alert is reported) is separated from the policy (how it is handled), and code dealing with alerts is separated from code dealing with normal circumstances. The granularity of the policies (i.e., to what parts of the code they apply) can be specified at different scoping levels, from expressions and blocks to whole classes and packages.

Separating normality and exceptionality has already been demonstrated with AspectJ [23], but the AspectJ solution is less notationally elegant, and fails to separate mechanism from policy (it only deals with exceptions).[1] Using domain-specific syntax makes the extension easier to deal with for programmers unfamiliar with the full complexity of general aspect languages. Our alert extension is described in full in [6]. Here, we will look at the implementation of a simplified version for the Tiny Imperative Language.

2.1 The TIL Language

The Tiny Imperative Language (TIL) is a simple imperative programming language used for educational [10] and comparison purposes in the program transformation community. The grammar for TIL is given in the appendix (Section A). A TIL program consists of a list of function definitions followed by a main program. TIL statements include the usual `if`, `while`, `for` and block control statements, variable declarations and assignments. Expressions include boolean, string and integer literals, variables, operator calls and function calls. We will use the name TIL+Alert for the extended TIL language.

[1] We are not experts on aspect orientation, but we believe that the full separation of concerns available with our alert system is difficult if not impossible to achieve with existing general aspect languages.

Alert declaration. *Alert declarations are given after the regular function declaration. Actual arguments and the function's return value are available in the alert condition expressions. Pre-alerts have a condition that is checked before a call to the function and typically involve checks on the arguments; post-alerts are checked after the call has returned, and typically involve the return code (accessible as the special variable* `value`*, legal only in alert conditions and handlers.).*

```
FunDecl AlertDecl       -> FunDecl
"pre" Exp "alert" Id    -> AlertDecl
"post" Exp "alert" Id   -> AlertDecl
"value"                 -> Exp
```

Figure 1. Grammar for TIL function declarations with alert extension.

2.2 Alert declarations and handlers

An *alert declaration* specifies a function's alert mechanisms. Our simple extension allows two ways of reporting alerts; via a condition which is checked before a call, or via a condition checked after a call. The pre-checks allows a function to report invalid parameters (before the call, avoiding the need for checks within the function itself), while the post-checks can be used for testing return values. The syntax for alert declarations is given in Figure 1. As an example, the following function definition declares that the function `lookup` raises the alert `Failed` if the return value is an empty string:

```
fun lookup(key : string) : string
  post value == "" alert Failed
begin ... end
```

The following declaration specifies that a `ParameterError` occurs if `f` is called with an argument less than zero, and that if the return value is `-1`, an `Aborted` alert was raised:

```
fun f(x : int) : int
  pre x < 0 alert ParameterError
  post value == -1 alert Aborted
```

A *handler declaration* specifies what action is to be taken if a given alert is raised in a function matched by its call pattern (the syntax is shown in Figure 2). The call pattern can be either `*` (all functions) or a list of named functions, possibly with parameter lists. This corresponds to the *pointcut* concept in AspectJ [3, 19]. The handler itself is a statement; it can reference the actual arguments of the call (if a formal parameter list is provided in the handler declaration), names from the scope to which it applies, and `value`—the return value of the function for which the handler was called. For example, this handler declaration specifies that the program should abort with an error message in case of a fatal error:

```
on FatalError in * begin
  print("Fatal Error!");
  exit(1);
end
```

Handlers. *A handler associates a statement with an alert condition; the statement is executed if the alert occurs. The* `use` *statement substitutes a value for the return value of the alerting function.*

```
"on" Id "in" {CallPattern ","} Stat -> Stat
"use" Exp ";"                       -> Stat
```

Call patterns. *A * matches a call to any function. The second form matches a call to a named function; the third form makes the actual arguments of the call available to the handler.*

```
"*"                    -> CallPattern
Id                     -> CallPattern
Id "(" {Id ","}* ")"   -> CallPattern
```

Figure 2. Grammar for handler declarations. The notation `{X Y}*` means `X` repeated zero or more times, separated by `Y`s.

The `use` statement is used to "return" a value from the handler; this value will be given to the original caller as if it was returned directly from the function called:

```
on Failed in lookup(k) begin
  log("lookup failed: ", k);
  use "Unknown";
end
```

The `on`-declaration is a statement, and applies to all calls matching the call pattern within the same lexical scope. If more than one handler may apply for a given alert, the most specific one closest in scoping applies.

TIL+Alert does not add anything that can not be expressed in TIL itself, at the cost of less notational convenience. For example, given the above alert and handler declarations, a call

```
print(lookup("foo"));
```

would need to implemented somewhat like

```
var t : string;
t = lookup("foo");
if t == "" then t = "Unknown"; end
print(t);
```

This cumbersome pattern should be familiar to many programmers (programming with Unix system calls, for instance, or with C in general): save the result in a temporary variable, test it, handle any error, resume normal operations if no error was detected or if the error was handled. Exceptions alleviate the need to check for errors on every return, but writing `try`/`catch` blocks everywhere a handler is needed is still cumbersome, and changing handling policies for large portions of code is tedious and error-prone.

3. Implementation of TIL+Alert

We have several possibilities when faced with the task of implementing a DSAL, or a language extension in general:

1. Compile to object code—write an entirely new compiler for the extended language.
2. Compile to unextended language—write an aspect-weaving preprocessor for an existing compiler.
3. Compile to aspect language—write a preprocessor for an existing aspect weaver.

The first choice is typically the most costly, and therefore also the least attractive. The second option is a common technique for bootstrapping new languages, and was used for both C++ and AspectJ. The third option is only possible if the subject language we are extending already supports a form of aspects which can be suitably used for writing implementing (most of) the semantics of our DSAL. We will discuss this option in more detail in Section 4.

DSALs are almost by definition extensions of existing languages, and we can therefore expect to have at least some language infrastructure. In other words, we need only consider the latter two situations above. In our experience, implementing the aspect extension as library + notation in a program transformation system is a very efficient approach in terms of development time.

3.1 DSAL = library + notation

We have said that (alert handling) aspects are meta-programs, then showed the programmer notation for these in Section 2.2 where we discussed the alert grammar. This covers the "notation" half of our equation. Now we will discuss how the semantics are implemented as a transformation library written in a program transformation system.

Stratego/XT [11] is our implementation vehicle of choice. Stratego is a domain-specific language for program transformation based on the paradigm of strategic programming [21] and provides many convenient language abstractions for our problem domain. The language is bundled with XT, a set of reusable transformation components and generators—in particular a formalism for defining language syntax, called SDF [26]—that support the development of language processing tools. In Section 4 we will discuss some of the benefits and drawbacks of using program transformation systems for implementing aspect weavers.

An existing language infrastructure for TIL exists that provides a grammar, a rudimentary compiler that does type checking and optimization, and finally a runtime that executes the compiled result. Together, these components make out a general-purpose transformation library for TIL. Using it, we can implement any program analysis and transformations on TIL programs [10]. The Alert grammar is implemented as a separate grammar module of about 30 lines of SDF code. Compositing this with the basic TIL grammar results in the complete syntax for the TIL+Alert language, c.f. the first step in Figure 3. We then use the TIL transformation library to implement a new Alert transformation library. Based on this, we can run meta-programs which perform the semantics of the alert constructs, i.e. the `on` and `pre`/`post` declarations: At compile-time, an abstract syntax tree for TIL+Alert is constructed and the corresponding meta-program for each alert construct is executed. Once all alert constructs in the program have been handled, the base program will have been rewritten. This completes the aspect weaving.

Ideologically, our approach can be considered an example of the "transformations for abstractions"-philosophy described by Visser [27] – we are effectively extending the open TIL infrastructure with transformations (our meta-programs) that provide new abstractions (the alerts). Next, we will describe the principles behind the implementation of the alert extension, and pay particular attention to the weaving done by the meta programs.

3.2 Type Checking

The constructs of the Alert language (`pre`, `post`, `on` and `use`) require their own type checking. To do this, we exploit the construction of the basic TIL type checker. It is a rule set. By adding new type checking rules to this set, we can easily

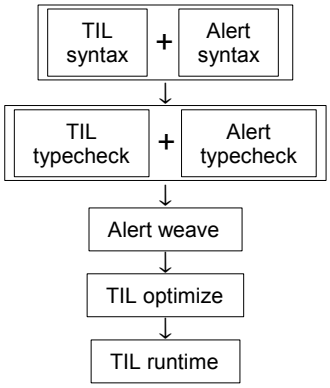

Figure 3. Implementation schematics. The *Alert weave* step implements the interpreter for the Alert meta-language and transforms a TIL+Alert program into a valid TIL program.

extend its domain (i.e. the ASTs it can process), as we do here for `use`. The following is a Stratego rewrite rule:

```
TypecheckUse: Use(e) -> Use(e'){t}
where  <typecheck-exp ; typeof> e => t
```

This rule, named `TypecheckUse`, says that if we are at a `Use` node in the AST with one subnode called `e` (this happens to be an expression), then we reuse the `typecheck-exp` function from the TIL library and annotate the `Use` node with the computed type `t`. The `;` operator works as function composition. The cases for `pre` and `post` are very similar. For type checking purposes, we define an `on` declaration to be a statement, thus having the `void` type. These few rules implement the "Alert typecheck" box in Figure 3.

3.3 Alert Weaving

The compilation flow in Figure 3 shows that after type checking, the DSAL meta-program parts of a TIL+Alert program are executed, effectuating the weaving. Once weaved, the Alert constructs are gone and the rest of the pipeline will process a pure TIL program. This program is optimised and compiled using unmodified steps of the TIL compiler.

The DSAL notation can be expanded using the simple translation scheme for we DSLs, described in the introduction, i.e. basic macro expansion, but with one crucial difference: whereas the DSL notation is expanded to library calls of a *subject* language library, the DSAL notation is expanded to library calls of a *transformation* language library, and the transformation language is generally different from the subject language. Here, TIL is our subject language and Stratego is our transformation language. Essentially, the DSAL notation is a syntactic abstraction over the Alert transformation library. This notation is embedded in the subject language (TIL), providing a distilled form of meta-programming inside TIL for managing the error handling concern.

When weaving Alert, we have to consider three constructs: the modified function definitions which now have `pre`/`post` conditions, the `on` handler declarations, and function calls. The code for the following cases are all part of the Alert transformation library where they are are implemented as Stratego rewrite rules. When the DSAL notation is expanded, it results in calls to these rules.

Pre/Post Conditions on Function Definitions Pre/post conditions are easy to process. They are merely markers, or annotations, on the functions. The expression of a pre/post condition can only be activated by an `on`-handler, so the meta-program processing the pre/post conditions has two tasks: first, to store the alert declaration for later use, and second, to remove it from the AST so that we may eventually reach a pure TIL AST. The following rewrite rule, `WeaveFunDef`, does this:

```
WeaveFunDef:
FunDef(x@FunDeclAlert(fd@FunDecl(n, _, _), _), body) ->
  FunDef(fd, body)
where rules( Functions: n -> x )
```

It takes a function definition (a `FunDef` node) that has a subnode which is a pre/post condition (a `FunDeclAlert`) and rewrites the `FunDef` node to a pure TIL `FunDef` by removing the `FunDeclAlert` node. Further, `WeaveFunDef` creates a new, dynamic rule called `Functions` that records a mapping from the name of this function to its complete pre/post alert declaration. A dynamic rule works exactly like a rewrite rule, but can be introduced at runtime, much like closures in functional programming languages. This is done with the `rules` construct. After `WeaveFunDef` has finished, the pre/post condition is removed, and the `Functions` rule can now be used as a mapping function from the name of a TIL+Alert function to its declaration.

In the code above, `_` is the wildcard pattern (matches anything) and v`@`$p(x)$ means bind the variable v to the AST matched by the pattern $p(x)$.

On The processing of `on` itself is also easy. Its node is removed from the AST and we add it to the current set of active `on`-handlers, maintained in the dynamic rule `On`. `On` maps from the name of an alert to the call patterns and handler for it.

```
WeaveOn: On(n, patterns, handler) -> None
where rules(On : n -> (patterns, handler))
```

Function Calls Rewriting function calls to adhere to the new semantics is the crux of the Alert DSAL, and is done by `WeaveFunCall`. This rule implements the following translation scheme. Consider the pattern for functions f in the following form, where f is the function name, f_i are the variable names, t_i are the corresponding types, tr is the return type, and the precondition is as explained earlier:

```
fun f(f0 : t0, ...) : tr
  pre exp alert signal
begin ... end
```

Whenever we see the declaration of an `on` handler, we need to process the subsequent calls in the same (static) scope, since these may now need to be transformed. We are looking for patterns on the form:

```
on signal in pattern handler;
...
z := f(e0, ...);
```

When we encounter an instance of this pattern, we may need to replace the call to f by some extra logic that performs the precondition check and, if necessary, executes the relevant `on`-handler according to the following call template[2].

```
z := begin
```

[2] The `begin`/`end` block here is called an expression block. It is effectively a closure that must always end in a `return`. It will be removed by a later translation step that lifts out the variables contained within it, finally giving a valid TIL program.

```
  var r : tr;
  var a_0 : t_0 := e_0; ...
  if exp then handler
  else r := f(a_0, ...) end
  return r;
end
```

`WeaveFunCall` will perform the aspect weaving. We will now describe the principles behind it, but not present the full source code, as this is available in the downloadable source code for TIL+Alert (see Section 5).

The weaving of `WeaveFunCall` can only happen at `FunCall` nodes, i.e. nodes in the TIL+Alert AST that are function calls. Assume `WeaveFunCall` is applied to a function call of the function f. First, it will check that f signals alerts by consulting the `Function` dynamic rule that was produced by `WeaveFunDef`. If indeed f has a declared alert, then the set of active `on` handlers for the current (static) scope is checked by consulting the `On` dynamic rule that was initialized by `WeaveOn`. Multiple `on` handlers can be active, so another Alert library function is used to resolve which takes precedence (the closest, most specific). Once the appropriate handler is found, the function call to f is rewritten according to the call template shown above, i.e. the FunCall node is replaced by an expression block (an `EBlock`) which does the precondition check before the call.

Extra care must be taken in the handling of variable names during this rewrite. The precondition expression is formulated in terms of the formal variable names of f, so we cannot insert that subtree unchanged. We must remap the variables, and this is done by a function called `remap-vars`. As the call template shows, for each formal parameter f_i of f, we create a local variable a_i that is assigned the actual value from the call site. We rename the variables in the precondition expression of f, from f_i to a_i, and insert the rewritten expression as *exp* in the call template.

3.4 Coordination

The meta-programs induced by the `on`, `pre` and `post` declarations are dispatched by a high-level strategy that can be likened to an interpreter for the Alert aspect extension. This strategy is implemented as a traversal over the TIL+Alert AST. It contains the logic responsible for translating the Alert notation into calls to the Alert transformation library, and in that capacity, it corresponds to the DSL macro expander. Its execution will coordinate the meta-programs for the various alert constructs. Once the traversal completes, all the Alert-specific nodes will have been excised from the tree, and the result is a woven TIL AST that can be optimized and run.

4. Discussion

While DSLs can often be implemented as rather simple macro expanders, the same translation scheme is apparently not applicable for DSALs. The cross-cutting nature of DSALs means that statements or declarations in a DSAL usually have non-local effects. A single line in the DSAL may bring about changes to every other line in the program, and this is not possible to achieve using macro expanders. However, the translation scheme offered by the macro expansion technique is appealing both because of its simplicity and its familiarity; we already have ample experience and tools which may be brought to bear if we could reformulate the DSAL implementation problem to be a DSL implementation problem. This is what our technique offers, by using a program transformation system to implement the library (semantics) for the DSAL notation (syntax). Here, we perform a brief evaluation of our approach.

4.1 Program Transformation

Program transformation languages are domain-specific languages for manipulating program trees. Stratego and other transformation language such as TXL [15] and ASF [25] all have abstract syntax trees as built-in data types, rewrite rules with structural pattern matching to perform tree modification, concrete syntax support and libraries with generic transformation functions. The advantage to using such languages for program transformation is that the transformation programs generally become smaller and more declarative when compared to implementations in general-purpose languages, be they imperative, object-oriented or functional.

High-level Transformations In our experience, when doing experiments with aspect language and aspect weaving, working with on high-level program representation such as the AST is often preferable to lower-level represention traditionally found in compiler-backends. The AST provides all the information from the original source code and is together with a symbol table a convient and familar data structure to work with. When working with ASTs, it is important for the transformation language to have good support for both reading and manipulating trees and tree-like data structures.

Generic Tree Traversals Many program transformation languages and functional languages, especially members of the ML family, have linguistic support for pattern matching on trees. We have already seen pattern matching in Stratego in the rewrite rules in Section 3. Using recursive functions and pattern matching, tree traversals are relatively simple to express, e.g.:

```
fun visit(Or(e, e)) = ..
  | visit(And(e, e)) = ..
```

In object-oriented (OO) languages, the Visitor pattern is a common idiom for tree traversal, but compared to pattern matching with recursion, it is very verbose. Both techniques perform poorly when the AST changes, however. Introducing a new AST node type requires changes to all recursive visitor functions, or in the OO case to the interface of the Visitor (and thus all classes implementing it). There is, however, an aspect-oriented solution to the cross-cutting-concern part of this problem [22].

Generic programming [20] in functional languages and generic traversals, as offered in Stratego, provide a solution. Generic traversals also allow arbitrary composition of traversal strategies.

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

This defines `bottomup` (post-order traversal) of a transformation `s` as "first, apply `bottomup(s)` recursively to all children of the current node, then apply the transformation `s` to the result". Once defined, this function can be used to succinctly program the variable renaming needed by the `WeaveFunDef` in Section 3.3:

```
remap-vars(|varmap) =
  bottomup(try(\ Var(n) -> Var(<lookup> (n, varmap)) \))
```

Syntax Analysis Support Program transformation languages typically come with parsing toolkits and libraries for manipulating existing languages, reducing the effort needed to create a language infrastructure. Also, there is often a tight

integration between the parser and the transformation language in transformation systems. Among other things, this allows expressing manipulations of code fragments from the subject language very precisely, using concrete syntax.

Rewriting with Concrete Syntax Another important task is tree manipulation. Rewrite rules provide a concise syntax and semantics for tree rewriting, but rewriting on ASTs can of course be expressed in any language. In program transformation languages, rewriting with concrete syntax, i.e. using code fragments written in the subject language is often provided, and this may improve the readability of rewrite rules considerably, e.g.:

```
Optimize: |[ if 0 then ~e0 else ~e1 end ]| -> |[ ~e1 ]|
```

Here, `~e0` and `~e1` are a meta-variables, i.e. variables in the transformation language (Stratego) and not the subject language (TIL).

Generic Transformation Libraries Libraries for language processing are not unique to program transformation systems, but transformation libraries often contain quite extensive collections of tree traversal and rule set evaluation strategies not found elsewhere. Also, some transformation systems provide generic, reusable functionality for data- and control-flow analysis, as well as basic support for variable renaming and type analysis. However, the libraries of transformation systems are often less complete than that of general purpose languages, when it comes to typical abstract data types.

Maturity and Learning Curve A clear disadvantage of contemporary program transformation systems is their relative immaturity when compared to implementations of mainstream, general-purpose languages. The compilers are usually slower, the development environments are not as advanced, and fewer options for debugging and profiling exist. Further, the same domain abstractions that make domain-specific transformation languages effective to use, also make them more difficult to learn, a tradeoff that must be evaluated when considering the use of a transformation language.

4.2 Program Transformation Languages for Aspect Implementation

The stance we take in this paper is that a aspect languages are a form of domain-specific transformation language; they provide convenient abstractions (join points, pointcuts, advice) for performing certain kinds of transformations (aspect weaving—dealing with cross-cutting concerns). They hide the full complexity of program transformation from programmers. Domain-specific aspect languages are even more domain-specific, and hide the complexities of general aspects from their users.

As domain-specific transformation languages, DSALs are conveniently implemented as libraries in a program transformation language. We make this claim based on our experience with the DSAL = library+notation method from constructing the following systems:

- A domain-specific error-handling aspect language [6]—a simplified version of this is used as an example in this paper. Our current implementation is for C, and is implemented in the Stratego program transformation language [11] using the C Transformers framework [9].
- A component and aspect language for adaptation and reuse of Java classes. An early version of this is described in [5]; it is implemented by translation to AspectJ [3, 19], using Stratego.
- AspectStratego [18]—an aspect-language extension to the Stratego program transformation language; implemented in Stratego itself, by compilation to primitive Stratego code.
- CodeBoost [7]—a transformation system for C++ that provides *user-defined rules*; an aspect language that allows users to declare library-specific optimization patterns inside the C++ code. The patterns are simple rewrite rules, executed at compile-time. User-defined rules is implemented with the library+notation technique, with the library written in Stratego.

Part of the design goals for many of these experiments was harnessing the expressive power of general program transformation systems into "domain-specific transformation languages" that the programmers of the subject languages could benefit from. In a word, these domain-specific transformation languages are DSALs. For most of our systems, the transformations underlying these extensions, i.e. the implementation of the DSAL semantics, are reusable Stratego libraries, and form the basis for further extensions and experiments.

Experiences One lesson learned from the construction of these DSALs is that good infrastructure for syntax extensions of the subject language is important. Reusing frontends from existing compilers usually preclude extending the syntax, as that would require massive changes to the frontend itself (and for mainstream languages, this is a substantial task). Implementing robust grammars for complicated languages like C++ and Java is infeasible, so language infrastructures provided by program transformation systems were of great help to us. Another lesson is that familiarity with language construction is crucial. Extending a subject language with an arbitrary DSAL may be very complicated, depending on what the DSAL is supposed to achieve. It may therefore be premature to expect regular developers to be able to design their own DSAL language extensions. This is often in more due to the complex semantics of the subject language itself, than the complexity of the DSAL.

4.3 Related Work

JTS, the Jakarta Tool Suite [8] is a toolkit for developing domain-specific languages. It consists of *Jak*, a DSL-extension to Java for implementing program transformation, and *Bali*, a tool for composing grammars. Jak allows syntax trees and tree fragments to be written in concrete syntax within a Java program, and provides abstractions for traversal and modification of syntax trees. Bali generates grammar specifications for a lexer and parser and class hierarchies for tree nodes, with constructor, editing and unparsing methods. Bali supports composition of grammars from multiple DSLs. DSL development with JTS is much like what we have described here; an existing language is extended with domain-specific syntax (in Bali), and a small tool is written (in Jak), translating the DSL to the base language.

XAspects [24] is a system for developing DSALs. It provides a plug-in architecture supporting the use of multiple DSALs within the same program. Declarations belonging to each DSAL are marked syntactically, picked up by the XAspects compiler and delivered to the plug-ins. The plug-ins then perform any necessary modification to the visible program interface (declared classes and methods). Bytecode is then generated by the AspectJ compiler; the plug-ins then have an opportunity to perform cross-cutting analysis and generating AspectJ code which is woven by the AspectJ compiler. Thus, implementation of a new DSAL is reduced to cre-

ating a plug-in which performs the necessary analyses and generates AspectJ code. Our method, with program transformation, can either complement XAspects, as a way of implementing XAspects plug-ins, or replace it, by developing a libraries for AspectJ manipulation in a program transformation language. The plug-in architecture of XAspects is appealing, as it forces possibly conflicting DSALs to conform to a common framework, making composition of DSALs easier. Both XAspects and our implementation can be seen as library+notation approaches. However, since domain-specific aspects in XAspects can only modify existing code using AspectJ advice and intertype declarations, there are limits to the invasiveness of the DSAL expressed with XAspects. Our implementation strategy has no such constraint since Stratego supports any kind of code modification.

The AspectBench Compiler [2] provides another open-ended aspect compiler, but is more focused on general aspect languages. It implements the AspectJ language, but is also intended as research platform for experimenting with aspect language extensions generally.

Logic meta programming (LMP) is proposed as a framework for implementing DSALs in [13], because expressing cross-cutting concerns using logic languages is appealing. We believe that our approach could be instantiated with an LMP system as well: the DSAL notation may be desugared into small logic meta-programs which perform the actual weaving. Depending on the logic language, constructing and compositing logic-based transformation libraries may be possible.

In [14], the authors argue that AOP is a general discipline that should be confine itself in a domain-specific language, but rather be addressed with a general, open framework for composing all kinds of aspects. Such an infrastructure, should it be constructed, would be an interesting compilation target to expand DSAL notation to.

Gray and Roychoudhury [17] describe the implementation of a general aspect language for Object Pascal using the DMS program transformation system. They conclude that since transformation systems often provide good and reusable language infrastructure for various subject languages, they are good starting points when developing new aspect extensions. We are of the same opinion, and advocate a disciplined approach where the aspect extensions themselves are implemented as reusable transformation libraries that may in turn be used a substrate for later extensions.

Assman and Ludwig [4] describe the implementation of aspect weaving using graph rewrite systems. The authors express the weaving steps in terms of graph rewrite rules, similar to how we describe them as tree rewrite rules. In principle, transformation libraries could be constructed from the sets of graph rewriting rules, but the rule set appears to always be evaluated exhaustively. This makes rule set composition (i.e. library extension) problematic, since two rule sets that are known to terminate may no longer terminate when composed. In Stratego, there is no fixed normalization strategy; the transformation programmer may select one from the library or compose one herself, which in practice adds a very useful degree of flexibility.

5. Conclusion

In this paper, we have discussed the *library+notation* method for implementing DSLs: building a library that implements the semantics of the domain, a syntax definition for the desired notation, and a simple translator that expands the notation into library calls. We showed how this method can also be used effectively for implementing DSALs by writing the library part in a program transformation system, expressing the notation as a syntax extension to a subject language, and translating the notation of the DSAL into library calls in the transformation system. This makes the DSAL a meta-program that is executed at compile-time, and that will rewrite the subject program according to the implemented DSAL semantics. Our illustrating examples were based around a small imperative language with an aspect extension for separately declaring error handling policies.

We argued that, based on our experience, program transformation systems are ideal vehicles for implementing such libraries because they themselves come with domain-specific languages and tools for doing language processing, which greatly reduces the burden of implementation when compared to general purpose languages.

The complete implementation of TIL+Alert is available at `www.codeboost.org/alert/til`.

Acknowledgments

Kalleberg is supported by the The Research Council of Norway (NFR) through the project PLI-AST. We would like to thank the anonymous reviewers for insightful feedback.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Meta-Programming*. Addison-Wesley, Boston, MA, USA, 2005. ISBN 0-321-22725-5.

[2] C. Allan, P. Avgustinov, A. S. Christensen, B. Dufour, C. Goard, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, and C. Verbrugge. abc: the AspectBench compiler for AspectJ – a workbench for aspect-oriented programming language and compilers research. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 88–89. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-193-7.

[3] AspectJ homepage. URL `http://eclipse.org/aspectj/`.

[4] U. Assmann and A. Ludwig. Aspect weaving with graph rewriting. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 24–36. Springer-Verlag, London, UK, 2000. ISBN 3-540-41172-0.

[5] A. H. Bagge, M. Bravenboer, K. T. Kalleberg, K. Muilwijk, and E. Visser. Adaptive code reuse by aspects, cloning and renaming. Technical Report UU-CS-2005-031, Utrecht University, 2005.

[6] A. H. Bagge, V. David, K. T. Kalleberg, and M. Haveraaen. Stayin' alert: Mouldable exception handling. In *Fifth International Conference on Generative Programming and Component Engineering (GPCE 2006)*. ACM Press, Portland, Oregon, October 2006.

[7] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75. IEEE Computer Society Press, Amsterdam, The Netherlands, September 2003.

[8] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8377-5.

[9] A. Borghi, V. David, and A. Demaille. C-transformers: A framework to write C program transformations. *ACM Crossroads*, 12(3), April 2006.

[10] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Tutorial, Examples, and Reference Manual*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, August 2005. (Draft).

[11] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006.

[12] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-831-9.

[13] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 110–127. Springer-Verlag, London, UK, 2002. ISBN 3-540-44284-7.

[14] C. A. Constantinides, A. Bader, T. H. Elrad, P. Netinant, and M. E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Comput. Surv.*, 32(1es):41, 2000. ISSN 0360-0300.

[15] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. Txl: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, 1991. ISSN 0096-0551.

[16] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992. ISSN 0892-4635.

[17] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-842-3.

[18] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In H. Cirstea and N. Marti-Oliet, editors, *Workshop on Rule-Based Programming (RULE'05)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Nara, Japan, April 2005.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming: 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, June 2001. ISBN 3-540-42206-4.

[20] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. *SIGPLAN Not.*, 38(3): 26–37, 2003. ISSN 0362-1340.

[21] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *Proceedings of Aspect-Oriented Software Development (AOSD'03)*, pages 168–177. ACM Press, Boston, USA, March 2003. URL `http://www.program-transformation.org/Transform/StrategicProgrammingMeetsAdaptiveProgramming`.

[22] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004. ISSN 0164-0925.

[23] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. *ICSE*, 00: 418, 2000. ISBN 1-58113-206-9.

[24] M. Shonle, K. Lieberherr, and A. Shah. XAspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37. ACM Press, New York, NY, USA, 2003. ISBN 1-58113-751-6.

[25] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002. ISSN 0164-0925.

[26] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[27] E. Visser. Transformations for abstractions. In J. Krinke and G. Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 3–12. IEEE Computer Society Press, Budapest, Hungary, October 2005.

A. TIL Grammar

Programs. A program is a list of function definitions, followed by a main program (a list of statements).

```
FunDef* Stat* -> Program
```

Functions. A function definition defines is function with a given signature (`FunDecl`) and body (a list of statements).

```
"fun" Id "(" {Param ","}* ")" ":" Type -> FunDecl
FunDecl "begin" Stat* "end"           -> FunDef
Id ":" Type                           -> Param
```

Statements:

```
"var" Id ";"                                -> Stat
"var" Id ":" Type ";"                       -> Stat
Id ":=" Exp ";"                             -> Stat
"begin" Stat* "end"                         -> Stat
"if" Exp "then" Stat* "end"                 -> Stat
"if" Exp "then" Stat* "else" Stat* "end"    -> Stat
"while" Exp "do" Stat* "end"                -> Stat
"for" Id ":=" Exp "to" Exp "do" Stat* "end" -> Stat
Id "(" {Exp ","}* ")" ";"                   -> Stat
"return" Exp ";"                            -> Stat
```

Expressions:

```
"true" | "false"      -> Exp
Id                    -> Exp
Int                   -> Exp
String                -> Exp
Exp Op Exp            -> Exp
"(" Exp ")"           -> Exp
Id "(" {Exp ","}* ")" -> Exp
```

Lexical syntax:

```
[A-Za-z][A-Za-z0-9]*    -> Id
[0-9]+                  -> Int
"\"" StrChar* "\""      -> String
~[\"\\\n] | [\\][\"\\n] -> StrChar
```

KALA: A Domain-Specific Solution to Tangled Aspect Code

Johan Fabry
INRIA Futurs - LIFL, Projet Jacquard/GOAL
Bâtiment M3
59655 Villeneuve d'Ascq, France
johan.fabry@lifl.fr

Nicolas Pessemier
INRIA Futurs - LIFL, Projet Jacquard/GOAL
Bâtiment M3
59655 Villeneuve d'Ascq, France
nicolas.pessemier@lifl.fr

1. Introduction

In multi-tiered distributed systems transaction management has long been a mainstay of concurrency management. Transactions were however originally conceived only for brief and unstructured database accesses. Because of this they are a poor match for applications that wish to access data in a more structured way, or for a relatively long time. Negative consequences of this mismatch are, for example, that transaction throughput is only optimal when each transaction has a very short life-time. The multiple shortcomings of classical transactions are recognized by an important body of work in the transaction management community. To address them, many advanced transaction models (ATMS) have been developed, including a formalism called ACTA [CR91]. Each of these advanced models addresses a subset of the known shortcomings of classical transactions.

As with classical transactions, advanced transaction management is a cross-cutting concern. We have therefore investigated how it can be modularized into an aspect and developed the domain-specific aspect language KALA [FD06]. KALA is based on the ACTA formalism, and KALA programs declare how a particular application uses an ATMS, as expressed in ACTA.

When performing this research, we encountered a problem in the aspect code itself. We found that because the ATMS concern is a complex concern it can be subdivided in multiple sub-concerns, and that the code for these sub-concerns cross-cut the aspect itself. This yields aspect code which itself tangles multiple concerns. We therefore termed this phenomenon *Tangled Aspect Code*.

In this paper we describe how KALA is able to address the problem of Tangled Aspect Code through the use of domain information. KALA was developed solely for the domain of ATMS, and with the intent to tackle this problem. As a result the modularization for sub-concerns offered by KALA is straightforward for the programmer and the composition of sub-concerns requires no programmer intervention.

2. KALA in a Nutshell

2.1 Advanced Transaction Management

As we have said above, a number of ATMS have been developed, each addressing a specific set of shortcomings of classical transactions. We do not give an overview of these models here, as this is outside of the scope of this paper. Instead we briefly discuss two well-known models: Nested Transactions [Mos81] and Sagas [GMS87]. These are illustrated in Figures 1 and 2, and we provide a short description of these models next.

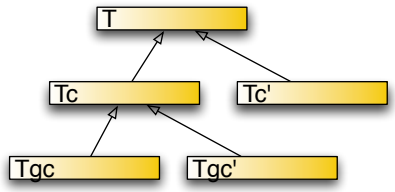

Figure 1. The Nested Transactions ATMS

Nested transactions [Mos81] is one of the oldest and easily the most well-known ATMS. It enables a running transaction T to have a number of child transactions Tc. Each Tc can *view* the data used by T. This is in contrast to classical transactions, where the data of T is not shared with other transactions. Tc may itself also have a number of children Tgc, forming a tree of transactions. When a child transaction Tc commits its data, this data is not written to the database, but instead *delegated* to its parent T, where it becomes part of the data of T. If a transaction Tx is the root of a transaction tree, *i.e.* it has no parent, Tx's data will be committed to the database when T commits. Lastly, if a child transaction Tc aborts, the parent T is unaffected. T is not required to also abort, *i.e.* when it ends it may choose freely to either commit or abort.

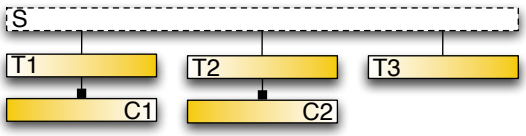

Figure 2. The Sagas ATMS

Sagas [GMS87] is, next to Nested Transactions, one of the oldest ATMS and also arguably one of the most referenced ATMS in the community. Sagas is tailored towards long-lived transactions. Instead of one long transaction T, a saga S splits T into a sequence of sub-transactions $T1$ to Tn. Each sub-transaction is a normal classical transaction and this sequence is executed completely before the saga commits. To abort or rollback a running saga S, the currently running sub-transaction Ti is aborted and the work of already committed transactions $T1$ to $Ti - 1$ has to be undone,

as their results have already been committed to the database. To allow this, the application programmer has to define for each subtransaction Ti a *compensating transaction* Ci that performs a semantical compensation action. To undo the work of $T1$ to $Ti-1$, $C1$ to $Ci-1$ are run by the runtime transaction monitor in inverse sequence, *i.e.* starting with $Ci-1$.

As said above, and illustrated by these two examples, different ATMS exhibit different concurrency management properties. This allows a given application to choose the advanced model that provides the best match to the concurrency management properties it requires. Also, if no matching model exists, it is possible to create a new model that provides the properties required by the application. However, as we argue in [Fab05] when using traditional software engineering approaches, there is only a small degree of separation of concerns between the ATMS concern and the other concerns present in the application.

2.2 ACTA and KALA

In general, to use transactions in an application, the developer needs to add *transaction demarcation code*, which is spread throughout the entire application. Using aspects, however, previous work has successfully achieved the modularization of the concern of classical transaction management [KG02, RC03, SLB02].

Advanced transaction models also suffer from the problem of cross-cutting demarcation code [Fab05]. Therefore, we created the domain-specific aspect language called KALA [FD06] to modularize advanced transaction management as an aspect for Java applications. KALA is based on the ACTA formalism for ATMS [CR91], which is accepted in the community as covering a wide field of advanced transaction models. In ACTA, extra properties are given to classical transactions, or properties of such transactions are modified, resulting in a collection of transactions that exhibits the behavior of an advanced model. The formalism declares three kinds of properties: *dependencies*, *views* and *delegation* which are declared between two transactions. The views and delegation properties correspond to viewing and delegation between them, which we have mentioned above when discussing nested transactions. Dependencies set relationships between two transactions and can be used to, for instance, sequence multiple transactions or trigger the beginning of a compensating transaction, which will be illustrated in Sections 4.2 and 6.1.1. KALA reifies the ACTA constructs of dependencies, views and delegation as the `dep`, `view` and `del` statements in the language. A full discussion of KALA is outside of the scope of this paper, instead we give a brief overview here. For a full description we refer to [FD06, Fab05].

A KALA program specifies what dependencies, views and delegations apply at the begin, commit and abort time of a transaction. As is the norm in multi-tier transactional systems, the life-cycle of a transaction coincides with the life-cycle of a method. The transaction begins when the method begins, commits when the method ends normally and aborts if the method ends with a (given type of) exception. All data accesses within this method (and within the methods called by this method) are included in the transaction. To identify this method, the signature of the method is used, possibly using wildcards, similar to AspectJ [asp06]. This yields the following overall structure of the KALA declarations for a method (square brackets indicate placeholders for actual KALA statements):

```
1 MethodSignature(ArgumentList){
2    [ preliminaries ]
3    begin { [ begin time properties ] }
4    commit { [ commit time properties ] }
5    abort { [ abort time properties ] }
6 }
```

Note that the depencency, view and delegation specifications inside a `begin`, `commit` and `abort` block are considered to happen in the same atomic action. Therefore the sequence of these statements within such a block is of no importance.

In order for dependencies, views and delegation to be applied to two transactions the KALA code needs to be able to refer to these transactions. This is performed through the use of a global naming service. Within KALA code, a local reference to such a name, *i.e.* an alias, is obtained through the `alias` statement. This statement takes the alias for that transaction, and a Java expression that evaluates to the key that is looked up in the name service. This expression has access to the actual parameters of the method and to aliases which have already been resolved. The alias `self` is always bound to the currently executing transaction. An alias placed in preliminaries is looked up immediately before the transaction starts, and is accessible thoughout the remainder of the KALA code for that method. Aliases placed in `begin`, `commit` and `abort` blocks are looked up at that moment in the life-cycle of the transaction, and are only accessible at that time. A transaction can be added to the naming service, *i.e.* given a global name, using the `name` statement. This statement takes an alias (which may be `self`), and a Java expression that evaluates to the key for the naming service. Note that, contrary to dependencies, views and delegation, the sequence of `name` and `alias` statements is important, as the expressions used in these statements have access to already resolved aliases.

In addition to naming, KALA also provides support for groups. Transactions can be added to a named group using the `groupAdd` statements. KALA makes no distinctions between transactions and groups of transactions, *i.e.* all statements can take groups or transactions as arguments[1].

KALA requires the programmer to perform manual memory deallocation for transactions (equivalent to the `free` statement in C++). This is performed through the `terminate` statement, which takes as argument the alias of the transaction or group to be freed. Termination can be performed at begin, commit or abort time of a transaction. If the transaction being terminated has not yet committed, it will be immediately forced to rollback.

Lastly, the preliminaries may contain an `autostart` statement. This statement specifies that a separate transaction needs to be started, in parallel to this transaction. The autostart specifies the signature of the method corresponding to this transaction, a list of actual parameters, and a KALA specification for this transaction. Autostarts are used, for example, within the KALA specification of a transaction Ti of a Saga S to specify the compensating transaction Ci. This specification then also contains dependencies at begin, commit and abort time of Ti that restrict Ci to only run if the Saga is aborted.

3. Tangled Aspect Code

3.1 Sub-concerns in the Aspect

If we consider various ATMS from a conceptual point of view, we find that these ATMS are not one monolithic block, but incorporate different design decisions. For example, consider how rollback is handled in the Sagas ATMS: compensating transactions are executed in the inverse sequence of the steps of the saga. Translated to application code, *i.e.* methods, each step corresponds to a method, as is each counterstep. If we consider conceptually the tasks that need to be performed by the demarcation code for such a step, we can infer that some parts of this code treat managing rollback of the saga. This code performs the work of defining and starting up compensating transactions, ensuring that these only begin when the

[1] The only exception being that a group cannot be the destination of a delegation operation.

saga aborts, and that they run in the right sequence. All of these low-level tasks comprise the code for one concern, which is managing rollback of the saga.

We can indeed consider management of rollbacks a true concern in this demarcation code, as it is a design decision of the ATMS that lies conceptually at a higher level of abstraction than the implementation details of the code, *i.e.* the various tasks of the code we identified above. This corresponds to the original consideration of a concern by Parnas [Par72] where he states that a module, *i.e.* a concern, corresponds to the implementation of a design decision. We claim that management of rollbacks is part of the design of the ATMS, as different implementations of this concern can be easily envisioned. For example, we could specify that compensating transactions run in the same sequence as the steps of the saga, or even let the compensating transactions run in parallel, to attempt to speed up saga rollback.

Sagas demarcation code will however contain more than code for rollbacks. In addition to this, the general structure of the Saga as a sequence of steps, where each step is itself a transaction, also needs to be defined. In other words, if we reflect on the various tasks performed by saga demarcation code, we find that this code treats two different sub-concerns: first the management of the *structure* of the overall transaction, and second the management of how *rollback* is performed.

This conceptual decomposition of an ATMS into different sub-concerns is not unique to the Sagas ATMS. We have performed a similar analysis of various ATMS, and found that these are also composed of multiple sub-concerns [Fab05]. In addition to the two sub-concerns of structure and rollback handling identified above, we have encountered the sub-concerns of *view management* and *delegation management*. Note that the list of four sub-concerns of ATMS is open-ended. Although we have identified these sub-concerns in many ATMS, it is possible that a new ATMS contains a sub-concern which we have not yet encountered.

To summarize, we should not consider an ATMS conceptually as one monolithic block, but rather as a composition of a number of sub-concerns. We have identified four such sub-concerns so far: the structure of the advanced transaction, how rollback is handled, the management of views and the management of delegation.

3.2 Tangled Aspect Code

If we wish to modify sub-concerns of an ATMS, or add implementations for new sub-concerns to an existing ATMS, the aspect that modularizes the ATMS must take into account this requirement. The aspect must be structured in such a way that modification of sub-concerns is easy, enabling easy creation of new ATMS through changes in the implementation of these sub-concerns. In other words, such a conceptual separation into modules should therefore ideally also be present in the KALA code. This would bring the well-known advantages of Separation of Concerns to the level of the aspect.

We find, however, that such a separation into multiple modules is absent from the KALA code. The reason for this is the primary decomposition inherent in KALA code. KALA code reflects the life-cycle of a transaction, and is therefore subdivided into three different phases: a begin phase, a commit phase and an abort phase. In contrast to this, the implementation of a sub-concern can affect multiple phases in the life-cycle, and in one phase the code for multiple sub-concerns can be present. For example, in Sagas the code for the rollback concern is contained in the begin and commit blocks of various steps, and in the commit and abort blocks of the top-level Saga, as we will see in Section 4.2. As a result, the aspect code for the ATMS concern tangles the multiple sub-concerns present in the ATMS being implemented.

We can see this tangling as a case of the tyranny of the dominant decomposition [TOHJ99]. The dominant decomposition in KALA is the life-cycle of the transaction in begin, commit and abort phases. The modularization of sub-concerns of an ATMS, however, is orthogonal to time. One sub-concern can act at multiple points in the life-cycle of a transaction. As a result, the sub-concerns cross-cut the dominant decomposition, leading to code which is scattered and tangled. In other words, a KALA program is a combination of different sub-concerns and we see that the code of these sub-concerns is tangled. We call this phenomenon, where the aspect itself is a tangled mess of sub-concerns, *tangled aspect code*.

4. Separate Definition of Concern Code

Instead of having an ATMS as a monolithic block, we want to apply the known benefits of separation of concerns [HVL95] to the process of creating and modifying an ATMS. Applying separation of concerns here, *i.e.* programming an ATMS in multiple modules, will greatly ease implementation and modification of this ATMS. This enables a new ATMS to be built, or an existing ATMS to be adapted. This in effect tailors an ATMS to best fit the transactional properties required by the application being developed.

We have seen above that these concerns cut across the dominant decomposition of KALA code, which is the life-cycle of a transaction. Therefore a separate modularization mechanism is required for these concerns. KALA contains such a mechanism, which allows separating the specification of the different concerns in a straightforward manner by writing them as separate KALA files. The composition of these modules into a complete specification is fully automatic, and is discussed in the next section. The straightforward modularization and automatic composition is possible because we used the properties of the domain when creating KALA.

In this section, we show how, applied to a given application, KALA can be used to define the different concerns of an ATMS separately. We assume here that an analysis has first been made of the different concerns present in the ATMS being used, as we have performed in Section 3. The different concerns identified in such an analysis, applied to an application, can then be written down separately in multiple KALA files, *i.e.* one file per concern. We show this by taking the Sagas ATMS we analyzed in Section 3.1, and writing KALA code for this ATMS.

As a concrete example of KALA code for the use of Sagas, we use the example of a bank transfer operation we introduced in [Fab05]. This is part of an application for bank cashiers, servicing customer at the teller window. The transfer operation is split in three steps: a `transfer`, a `printReceipt` and a `logTransfer` method, all called in sequence from a `moneyTransfer` method. The first step performs the actual money transfer, the second step prints out a receipt for the customer, and the third step updates the global log of the bank. Note that we do not include the Java code of the bank transfer operation here, as it is not relevant to this discussion.

We identified in Section 3.1 that the Sagas ATMS is comprised of two concerns: first the management of the structure of the overall transaction, and second the management of how rollback is performed. To have an implementation of these transactional concerns for the bank transfer operation, we now write KALA declarations for all four methods first for the structure concern, and second for the rollback concern.

4.1 Sagas: Structure

The first concern we implement here, is the structure of the saga. Recall that we identified this concern as the management of the overall structure of the saga, in which the steps perform their work. The structure concern codifies the subdivision of the saga into

multiple steps, allowing each step to obtain a reference to the top-level saga, and ensures that after the saga has ended cleanup work is performed.

Note that although we subdivide the discussion of the implementation of this concern into two parts, all the code for the structure concern is implemented in one file, as is indicated by the continuity in line number counting.

4.1.1 Saga Top-level

The first KALA declarations we show are for the top-level `moneyTransfer` method and are given below. This code registers itself in the naming service, such that the steps in the saga, shown later, can obtain a reference to the saga. At commit and abort time, the unique identifier of this saga is used to refer to a group name which is therefore guaranteed to be unique for this saga. In this group, the various steps of the saga will have registered themselves. As a result, termination of this group implies termination of all the steps of the saga, and together with termination of the saga itself ensures proper cleanup is performed.

```
1  Cashier.moneyTransfer
2     (Account src, Account dest , int amt) {
3     name(self Thread.currentThread());
4     commit { terminate("ID" + self + "Step");
5              terminate(self); }
6     abort {  terminate("ID" + self + "Step");
7              terminate(self); }
8  }
```

4.1.2 Saga Steps

The code of all the steps of the saga is virtually identical, the only difference being the identification of the method corresponding to each step. We therefore only show the code for the `logTransfer` step. Each of these steps first require a reference to the top-level transaction so as to, second, add itself to the group of steps. By adding itself to the group of steps, it ensures that it will be terminated when the saga ends, by the code either in line 5 or 7.

```
9  Cashier.logTransfer
10    (Account src, Account dest , int amt) {
11    alias (Saga Thread.currentThread());
12    groupAdd(self "ID" + Saga + "Step");
13 }
```

This concludes the code for the structure concern of the sagas ATMS, applied to the bank transfer example. This code implements the structure of the saga in multiple steps, with termination of the steps when the saga ends. The following concern will add the handling of rollbacks of this structure, yielding the behavior of the Sagas ATMS.

4.2 Sagas: Rollback Handling

The second concern which we implement here, is rollback handling for the saga. Recall that in order to rollback a saga, the currently executing step is aborted, and that all committed steps are compensated for by executing compensating steps in the reverse sequence of step execution.

The KALA code below is an implementation of the above concern, and is defined in a separate KALA file. Again, we subdivide the discussion of the implementation in multiple parts, and the line numbers show this code all belongs to one file.

4.2.1 Saga Top-level

The top level of the saga registers itself, as in the structure concern, because the steps and compensating steps will place dependencies on the sagas, as we see later. At commit and abort time, the group of compensating steps is aborted, similar to what is performed in the structure concern.

```
1  Cashier.moneyTransfer
2     (Account src, Account dest , int amt) {
3     name(self Thread.currentThread());
4     commit { terminate("ID" + self + "Comp");
5              terminate(self); }
6     abort {  terminate("ID" + self + "Comp");
7              terminate(self); }
8  }
```

4.2.2 Last Step

In the last step of the saga, to implement the rollback concern, a number of dependencies have to be set between the step and the saga when the step begins. To set these dependencies, in lines 12 and 13, a reference to the saga has to be obtained, which is performed in line 11. `Saga ad self` forces the Saga to abort if this transaction aborts. `self wd Saga` states that if the Saga aborts before this transaction ends, it is also forced to abort. `Saga scd self` ensures that the Saga does not commit before this transaction has committed.

```
9  Cashier.logTransfer
10    (Account src, Account dest , int amt) {
11    alias (Saga Thread.currentThread());
12    begin { dep(Saga ad self); dep(self wd Saga);
13            dep(Saga scd self); }
14 }
```

4.2.3 First and Second Step

The first step of the saga needs to declare the compensating transaction used when the saga rollbacks. It achieves this by using an `autostart` statement in lines 18 thru 22, which compensates a bank transfer simply by performing the inverse transfer operation. The secondary transaction registers itself under a unique name in line 21, so that in lines 23 and 26 a reference can be obtained to this transaction to set the required dependencies. Also, the compensating transaction adds itself to the group of compensating transactions in line 22, ensuring it is properly terminated in line 4 or 6, when the saga ends. The dependencies on the compensating transaction ensure that it does not begin unless this transaction has committed (`Comp bcd self`), only begins if the saga aborts (`Comp bad Saga`) and disallow it to abort in that case (`Comp cmd Saga`)

```
15 Cashier.transfer
16    (Account src, Account dest , int amt) {
17    alias (Saga Thread.currentThread());
18    autostart (transfer
19       (Account src, Account dest, int amt)
20       (dest, src, amt) {
21          name(self "ID" + Saga + "Comp");
22          groupAdd(self "ID" + Saga + "Comp"); });
23    begin {  alias (Comp "ID" + Saga + "Comp");
24             dep(Saga ad self); dep(self wd Saga);
25             dep(Comp bcd self); }
26    commit { alias (Comp "ID" + Saga + "Comp");
27             dep(Comp cmd Saga);dep(Comp bad Saga);}
28 }
```

The second step of the saga is highly similar to the first step of the saga, the only differences being a different `autostart`, and dependencies being placed on the previous compensating transaction, as can be seen below. A reference to the compensating transaction of the first step of the saga is obtained in line 32. This allows the `CompPrev wcd Comp` dependency to be placed in line 42, ensuring that the previous compensating transaction begins after this has

ended. In other words, this determines the sequence in which the compensating transactions will run.

```
Cashier.printReceipt
   (Account src, Account dest, int amt) {
   alias (Saga Thread.currentThread());
   alias (CompPrev "ID"+Saga+"Comp");
   autostart (printTransferCancel
      (Account src, Account dest, int amt)
      (src, dest, amt) {
          name(self "ID" + Saga + "Comp");
          groupAdd(self "ID" + Saga + "Comp"); });
   begin {  alias (Comp "ID" + Saga + "Comp");
            dep(Saga ad self); dep(self wd Saga);
            dep(Comp bcd self); }
   commit { alias (Comp "ID" + Saga + "Comp");
            dep(CompPrev wcd Comp);
            dep(Comp cmd Saga);dep(Comp bad Saga);}
}
```

This completes the KALA code of the concern of rollback handling for the bank transfer operation using the sagas ATMS. As there are no more concerns in this ATMS, this concludes the KALA code for this example.

4.3 Conclusion

In this section we have shown how the implementation of a chosen ATMS for a given application is modularized using KALA code. In KALA, each module is implemented in a separate file. As an example we have shown a bank transfer operation that uses the Sagas ATMS. We have taken the decomposition of the Sagas ATMS, perfomed in Section 3.1, which identified two concerns, and given the KALA code for each of these concerns.

This modularization frees us from having to write tangled aspect code, which brings the benefits of separation of concerns to the process of defining an ATMS as an aspect. Instead of having to write aspect code which itself is tangled with multiple concerns, with all the impediments this entails, we now cleanly separate each concern in a separate KALA module.

5. Composing KALA Code

Given a definition of an ATMS concern in multiple modules, these need to be composed to form the complete KALA program. Conceptually, the concerns are combined before the ATMS aspect is woven, because it is the combination of these concerns that forms the complete definition of the ATMS. The KALA weaver, therefore, is not built to weave each concern of an ATMS separately into the base code. Instead all KALA modules are combined into one KALA file, describing how the ATMS is used, and this full description is woven into the base code[2].

Because of the domain-specific nature of KALA we were able to fully take into account the properties of the domain, yielding a composition mechanism that requires no programmer intervention.This is mainly due to the inherent composition properties of the ACTA model, which were taken into account when designing the KALA language. As a result, composition of multiple KALA modules is straightforward. In fact, composing multiple specifications in essence boils down to a simple merge, as we show here.

Conceptually, different KALA specifications declare that different actions need to take place at a given time in the life-cycle of a transaction: before the transaction begins, at begin time, at commit time or at abort time. In the composed file, therefore, for each of these moments in the life cycle all the actions defined for that point need to be performed. In other words, all the declarations that pertain to one moment in the life-cycle of the transaction have to be gathered into one block of the resulting specification.

The sequence of statements for naming and grouping within this composition matters, however, as an alias referred in a KALA statement needs to have been previously looked up. Therefore, when composing multiple KALA specifications, the partial ordering of naming and grouping statements within each KALA file needs to be preserved in the global file.

Considering in more detail the `begin`, `commit` and `abort` blocks of KALA code, we can state that the sequence of the code for setting dependencies, placing views, performing delegation and termination, however, is irrelevant. This is because, as said in Section 2.2, these are considered to happen in the same atomic action of begin, commit or abort. Therefore, when composing a number of `begin`, `commit` or `abort` blocks for the same method, their dependency, view, delegation and terminate statements can be simply joined into one sequence which respects the partial ordering of names and groups. The same observation holds for `autostart` statements, as their sequence in the KALA code also is of no importance. All `autostart` statements for one method are placed before the begin block of the composed KALA specification.

We can implement the above composition by a simple merge, the implementation of which is outlined next. Given that we have a number of KALA specifications for one method and we need to generate an output file:

1. Start the output file with the method signature suffixed with {.
2. For each specification, take the sequence of top-level declarations and add them to the output file.
3. Write the start of a `begin` block to the output file.
4. For each specification take the sequence of begin declarations and add them to the output file.
5. Write the close of the `begin` block, and the start of the `commit` block to the output file.
6. For each specification take the sequence of commit declarations and add them to the output file.
7. Write the close of the `commit` block, and the start of the `abort` block to the output file.
8. For each specification take the sequence of abort declarations and add them to the output file.
9. Write the close of the `abort` block and the closing } to the output file.

There is one downside, however, to this simple merging, which is name clashes: multiple modules should not define the same names. If these modules redefine the name with the same target, as in line 3 of both modules in the sagas example in Section 4, this is not an issue. But if multiple modules define the same name for a different target this will lead to wrongly placed dependencies, views, delegation, and so on, yielding faulty code. Conceptually, this issue can, however, be easily solved through a renaming or a merge of names. Therefore we do not provide an outline of such an implementation here.

The above is all which is required to compose multiple KALA modules into one full program. Thanks to the properties of the domain, which were taken into account when designing KALA, we have a straightforward composition mechanism that requires no programmer intervention.

[2] Although this aspect code will be tangled aspect code, this is not an issue since this code is but an intermediate representation which is not presented to a programmer.

6. Building a New ATMS: Cooperating Nested Transactions

In this section we show how we can use KALA to define a new ATMS to fit a given application or class of applications. The goal is to achieve an ATMS in which the transactional properties better align with the transactional properties of the (class of) application(s). We show this by creating a new ATMS, which we call Cooperating Nested Transactions, that aims to achieve the highest possible performance for computations that are hierarchically structured.

Before we introduce Cooperating Nested Transactions, we first give a definition of the Nested Transactions ATMS in multiple KALA modules. Second, we show how we can easily modify this definition to yield the Cooperating Nested Transactions ATMS.

In this section, we do not provide example applications to which the KALA code is applied. This is because as we solely wish to concentrate on the implementation of the ATMS, without considering how this ATMS is used by an application. We will use placeholder code, which is marked like this, when referring to base-level entities, such as method signatures. When these ATMS are used for a given application, this placeholder code needs to be replaced by the appropriate code for that application.

6.1 Nested Transactions

In Section 3.1 we established that Nested Transactions is composed out of four different concerns: structure, handling of rollbacks, view management and delegation of operations. We now write KALA code for each of these concerns separately.

6.1.1 Structure

The structure of Nested Transactions is not fixed statically as in Sagas, instead of this, at runtime a tree structure of transactions is built. Each transaction that forms a part of the tree structure is solely responsible for itself. Given such a tree structure, built at runtime, there is however one restriction: a parent may not commit before all its children have ended. Therefore a commit dependency `cd` needs to be placed between a parent and each of its children. This requires that each child obtain a reference to its parent before placing this dependency, as shown in line 4 of the code below. We achieve this by first letting each transaction name itself (line 2), so that it can be referred to by its children, and second letting each transaction obtain a reference to its parent by performing a lookup in line 3.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    begin {  dep(parent cd self); }
5    commit { terminate(self); }
6    abort {  terminate(self); }
7 }
```

6.1.2 Rollback Handling

When rolling back a transaction which is a part of a tree of nested transactions we need to ensure that if this transaction aborts, all its children also abort. This is implemented first by letting each child add itself to a group associated with the parent in line 4 of the code below, and second by letting each transaction terminate its children when aborting, in line 6 of the code below. Having each child add itself to the group associated with the parent, however, also implies that each parent needs to also clean up this group when committing, which is performed in line 5.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    groupAdd(self "ID" + parent + "Children");
5    commit { terminate("ID" + self + "Children"); }
6    abort {  terminate("ID" + self + "Children"); }
7 }
```

6.1.3 Delegation

Upon commit of a child its work is delegated to the parent, which is performed in line 4 of the KALA code below. Again this requires a reference to the parent, which in turn requires that each transaction register itself.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    commit { del(self parent); }
5 }
```

6.1.4 View Management

Thirdly, in Nested Transactions, a child has a view on the intermediate results of its parent, which is achieved by setting the view at begin time in line 4 of the code below.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    begin { view(self parent); }
5 }
```

This concludes the definition of the Nested Transactions ATMS. We now show how we can straightforwardly modify this ATMS to better fit a particular class of applications.

6.2 Cooperating Nested Transactions

One of the advantages of using the multi-tiered architecture in a large-scale distributed system is the ability of this architecture to provide a faster response time of the middle tier through load balancing. We can use parallelization on multiple servers to perform sub-computations of a given algorithm in parallel, but we want the entire computation to be performed as a single transaction to prevent data inconsistency. In a hierarchically structured computation, we can have sub-computations as nested sub-transactions of the main algorithm, and distribute sub-transactions over multiple servers, to be performed in parallel.

We can consider using Nested Transactions as an ATMS for this application: as sub-computations are sub-transactions they will preserve data consistency, and can access the data of the parent. Also, a failure in the sub-computation will not necessarily imply that the entire computation is lost. This allows graceful recovery of errors in the computation, without needlessly losing work. Having sub-computations performed in parallel, however, may entail that each of these sub-computations needs to be able to access the other computations' intermediate results, as they are supposed to cooperate, in parallel, to achieve the overall goal. This is not possible when using nested transactions and therefore, we have adapted the Nested Transactions ATMS to allow sharing between multiple sub-transactions, yielding a new ATMS: *Cooperating Nested Transactions* (CNT).

In CNT, all siblings of a parent transaction have access to each other's intermediate results though a view relationship. This, however, has an impact when aborting a child transaction. The siblings which have seen the inconsistent data of this child and have not committed are also considered to be inconsistent and should abort. This only applies to the sibling transactions that run simultaneously, in parallel, with the aborting sub-transaction.

Siblings that have committed before the aborter are not aborted, and siblings that start after the abortion need not abort. This limits the lost work in such cases to only include sibling sub-transactions which run at the same time as the aborting sub-transactions. This is an advantage of using CNT over running the entire computation in one transaction. If we would do this and a sub-computation aborts, automatically all of the work of the entire computation would be lost. With CNT, only the work of the sub-computations simultaneously running is lost.

We have implemented CNT in KALA by taking the implementation of Nested Transactions and modifying the concerns of view management and rollback handling. This illustrates one of the benefits of applying separation of concerns at the level of the ATMS definition, easing modification of an ATMS as only the code for the changing concerns needs to be considered, as we show next.

6.2.1 View Management

In CNT, all children of a given transaction can see each other's intermediate results. To implement this, in the code below, views are set from this transaction to all siblings, and the reverse, in line 7. This, however, requires each child of a transaction to add itself to the group of children of the parent, performed in line 4, and that a reference to be obtained to this group, in line 5. Also, when a transaction ends, the group of children of this transaction has to be removed by the system, which is performed in lines 8 and 9 of the code below.

```
1  packageName.className.methodName(parameterList) {
2     name(self name expression);
3     alias(parent extends expression);
4     groupAdd(self "ID" + parent + "Children");
5     alias(siblings "ID" + parent + "Children");
6     begin {  view(self parent);
7        view(self siblings); view(siblings self); }
8     commit { terminate("ID" + self + "Children"); }
9     abort {  terminate("ID" + self + "Children"); }
10 }
```

6.2.2 Rollback Handling

When performing rollback, siblings of the erroneous transaction should also abort, as they have seen the intermediate state of the aborting transaction. We cannot modify transactions that have already committed, but we can abort all currently running siblings of the aborting transaction, which is performed by the `terminate` statement in line 7.

```
1  packageName.className.methodName(parameterList) {
2     name(self name expression);
3     alias(parent parent expression);
4     groupAdd(self "ID" + parent + "Children");
5     commit { terminate("ID" + self +  "Children"); }
6     abort {  terminate("ID" + self + "Children");
7              terminate("ID" + parent +"Children"); }
8  }
```

7. Conclusion

KALA was designed to enable the modular specification of ATMS, avoiding the need to write tangled aspect code. In this paper we introduced how KALA enables the application of separation of concerns in the process of defining an ATMS.

In KALA, each concern can straightforwardly be written in a separate module and the composition does not require any programmer intervention. This is thanks to the domain-specific nature of KALA, where the properties of the domain were extensively taken into account when defining the modularization and composition mechanism.

We have shown how two existing ATMS can be programmed as KALA modules, namely Nested Transactions and Sagas. Furthermore, we described how a new ATMS: Cooperating Nested Transactions was created by modifying a number of modules from an existing ATMS, in this case Nested Transactions. This shows the benefit of modularization in KALA code, *i.e.* applying separation of concerns when defining an ATMS.

Acknowledgments

Thanks to Denis Conan for fruitful discussions when considering the topic of Tangled Aspect Code and thanks to Theo D'Hondt for supporting this research.

References

[asp06] The AspectJ project, 2006. http://eclipse.org/aspectj/.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112, 1991.

[Fab05] Johan Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde (PROG), July 2005.

[FD06] Johan Fabry and Theo D'Hondt. KALA: Kernel aspect language for advanced transactions. In *Proceedings of the 2006 ACM Symposium on Applied Computing Conference*, 2006.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[HVL95] Walter L. Hürsh and Cristina Videira Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[KG02] Jörg Kienzle and Rachid Guerraoui. AOP: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag, 2002.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

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

[RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[SLB02] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM, 2002.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars

Damijan Rebernak Marjan Mernik
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, 2000 Maribor, Slovenia
{damijan.rebernak, marjan.mernik}@uni-mb.si

Hui Wu Jeff Gray
University of Alabama at Birmingham
Department of Computer and Information Sciences
1300 University Blvd, Birmingham, AL 35294, USA
{wuh, gray}@cis.uab.edu

Abstract

The emergence of crosscutting concerns can be observed in various representations of software artifacts (e.g., source code, models, requirements, and language grammars). Although much of the focus of AOP has been on aspect languages that augment the descriptive power of general purpose programming languages, there is also a need for domain-specific aspect languages that address particular crosscutting concerns found in software representations other than traditional source code. This paper discusses the issues involved in the design and implementation of domain-specific aspect languages that are focused within the domain of language specification. Specifically, the paper outlines the challenges and issues that we faced while designing two separate aspect languages that assist in modularizing crosscutting concerns in grammars.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics, syntax; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, data type and structures, frameworks, inheritance patterns; D.3.4 [*Programming Languages*]: Processors—compilers, debuggers, interpreters, parsing, preprocessors, compiler generators

General Terms Algorithms, Design, Languages.

Keywords Aspect-Oriented Programming, Domain-Specific Languages, grammars, language specification, join point models

1. Introduction

Over the past decade, many aspect-oriented languages have been proposed, designed and implemented (e.g., AspectC [2], AspectC# [3], and AspectJ [4]). However, the majority of these efforts are devoted to general-purpose aspect languages (GPALs), despite the fact that preliminary work in AOP had its genesis with domain-specific aspect languages (DSALs) [23]. A DSAL is focused on the description of specific crosscutting concerns (e.g., concurrency and distribution) that provide language constructs tailored to the particular representation of such concerns. Examples of DSALs include [9, 12, 30, 32]. In comparison, a GPAL is an aspect language that is not coupled to any specific crosscutting concern and provides general language constructs that permit modularization of a broad range of crosscutting concerns. The majority of DSALs have been developed for languages that are general-purpose programming languages (GPLs) [30]; i.e., the aspect-language is focused on a specific concern, but it is applied to a GPL such as Java or C++. The scope of this paper is focused on the concept of a DSAL that is applied to a domain-specific language (DSL) [24]; i.e., the aspect language is focused on a specific concern and applied to a DSL that also captures the intentions of an expert in a particular domain. The distinction is highlighted by the partitioning of the aspect language (which can be either a GPAL or a DSAL) from the associated component language (which can be either a GPL or DSL). In the DSAL/DSL combination explored in this paper, a different join point model was needed. This paper discusses several issues associated with DSALs applied to DSLs, rather than GPLs.

The focus of this paper is in the well-established domain of programming language definition and compiler generation. Historically, the development of the first compilers in the late-fifties were implemented without adequate tools, resulting in a very complicated and time consuming task. To assist in compiler and language tool construction, formal methods were developed that made the implementation of programming languages easier. Such formal methods contributed to the automatic generation of compilers/interpreters. Several concepts from general programming languages have been adopted into the formalisms used to specify languages, such as object-oriented techniques [27]. To achieve modularity, extensibility and reusability to the fullest extent, new techniques such as aspect-orientation are being used to assist in modularizing the semantic concerns that crosscut many language components described in a grammar [15, 20, 21, 28, 36].

Within a language specification, modularization is typically based on language syntax constructs (e.g., declarations, expressions, and commands). Adding new functionality to an existing language sometimes can be done in a modular way by providing separate grammar productions associated with the extension. For example, additions made to specific types of expressions within a language can be made by changing only those syntax and semantic productions associated with expressions. In such cases, a new feature does not crosscut other productions within the language specification. However, there are certain types of language extensions (e.g., type checking and code generation) that may require changes in many (if not in all) of the language productions represented in the grammar. Because language specifications are also used to generate language-based tools automatically (e.g., editors, type checkers, and debuggers) [16], the various concerns associated with each language tool are often scattered throughout the core language specification. Such language extensions to support tool generation emerge as aspects that crosscut language components [36]. As such, these concerns often represent refinements

over the structure of the grammar [5]. This paper shows how application of aspect-oriented principles toward language specification can assist in modularizing the concerns that crosscut the language grammar.

This paper describes two approaches to integrate AOP with specifications that describe language grammars. Although the approaches are both focused on the common domain of language specification, the two resulting aspect languages apply to different compiler generators; namely, LISA [26] and ANTLR [1]. LISA relies on attribute grammars and ANTLR uses syntax-directed translation. Furthermore, LISA specifications enable higher modularity, extensibility, and reusability through concepts such as multiple attribute grammar inheritance and templates [25]. These differences among LISA and ANTLR contribute to proposing two DSALs that are quite different.

The organization of the paper is as follows. Challenges associated with the design and usage of GPALs and DSALs are discussed in Section 2. The various issues that are encountered when developing domain-specific join point models are presented in Section 3. In Sections 4 and 5, two separate DSALs for language definition (namely, AspectLISA and AspectG) are described. Related work is summarized in Section 6 followed by concluding remarks in Section 7.

2. Challenges Facing General-Purpose and Domain-Specific Aspect Languages

A DSL is a programming language for solving problems in a particular domain that provides built-in abstractions and notations for that domain. DSLs are usually small, more declarative than imperative, and more aligned to the needs of an end-user than general-purpose languages. Use of DSLs has been adopted for a variety of applications because of opportunities for systematic reuse and easier verification [24]. Because of these benefits, DSLs have become more important in software engineering [10, 14]. Moreover, DSLs offer possibilities for analysis, verification, optimization, parallelization, and transformation of DSL code, at a level of specialization not available with general-purpose code.

Similar conclusions can be drawn among GPALs and DSALs. Although GPALs are useful, certain crosscutting concerns are simply best described using DSALs [9, 30]. More importantly, domain-specific analysis and verification can be described by DSALs to prevent the occurence of subtle errors [31]. Yet in other cases, adding new aspects might produce inefficient code and domain-specific optimization is needed [19]. Furthermore, current GPALs (e.g., AspectJ) are not expressive enough to separate all concerns (e.g., structure-shy concerns [30]).

Clearly, in order to address fully the problem of separation of concerns, domain-specific solutions are needed. This has been observed also by other researchers. Gray recognized that specific domains will have numerous dominant decompositions and hence different crosscutting concerns [13]. Consequently, different aspect weavers will be required, even at various levels of abstraction (e.g., models). Hugunin defined four key areas of research that can improve the power and usability of AOP [18]: 1) improved separate compilation and static checking, 2) increased expressiveness for pointcuts, 3) simpler use of aspects in specialized domains, and 4) enhanced usability and extensibility of AOP development tools. Currently, AspectJ has initial support, but not completely sufficient, for particular domains by use of abstract aspect libraries. Not surprisingly, domain-specific aspects are one of the key future research areas in AOSD.

Several of the challenges of using GPALs can be overcome by DSALs. However, DSALs also have their own drawbacks. The most notable challenges of DSALs are: the cost of DSAL development and maintenance, inter-operability with other tools, and user training. One of the most formidable challenges is the extra effort required to design and implement a DSAL. Without an appropriate methodology and tools, the associated costs of introducing a new DSAL can be higher than the savings obtained through usage. With respect to DSLs, there are several techniques available to assist in implementation [24], such as: compiler/interpreter, embedding, preprocessing, and extensible compiler/interpreter. In addition, several tools [22, 26] exist to facilitate the DSL implementation process. We believe that such tools can also assist in DSAL design and implementation.

Another disadvantage of DSALs is that some domains have concerns that require several different DSALs to be developed. In such cases, several DSALs have to coordinate with each other and also interact with a component language. This imposes additional challenges in the design and implementation of DSALs, as well as in user training. It could be argued that it is not feasible to introduce many DSALs because it could overload the ability of the programmers to learn many different languages. However, conscious language design enables programmers to program at much higher abstraction levels and with less code. Conversely, programmers need to write more low-level code without DSALs [30].

3. Domain-Specific Join Point Models

When designing a new DSAL, a completely different join point model (JPM) might be needed as an alternative to the JPM used by a GPAL like AspectJ. The main issues in designing a JPM for a DSAL include:

- What are the join points that will be captured in the DSAL?
- Are the DSAL join points static or dynamic?
- What granularity is required for these join points?
- What is an appropriate pointcut language to describe these joinpoints?
- What are advice in this domain?
- Is extension/refinement only about behavior, or also structure?
- How is information exchanged between join points and associated advice (context exchange)? Is parameterization of advice needed?

In specific domains such as context-dependent computing (e.g., service-oriented and ubiquitous computing), AOP needs to address context passing concerns. Several specific approaches have been proposed such as: contextual pointcut expressions [8], temporal-based context aware pointcuts [17], and context-aware aspects [33]. Such concerns are more easily addressed through DSALs than GPALs [7]. A DSAL designer must also consider the issue of aspect ordering (i.e., how inter-aspect dependencies are handled) and if there is a need to dynamically add/remove aspects during the execution.

Another issue to be considered in the design of a DSAL is the degree that abstraction, reusability, modularity, and extensibility are needed to specify a crosscutting concern that is domain-specific. An abstraction is an entity that embodies a computation [35]. The abstraction principle shows that it is possible to construct abstractions over any syntactic class, provided the phrases of that class specify some kind of computation (e.g., function abstraction, procedure abstraction, and generic abstraction). A GPL provides a large set of powerful abstraction mechanisms, whereas a DSL strives to offer the correct set of predefined abstractions. This is reasonable because a GPL cannot possibly provide the right abstractions needed for all possible applications. Because a DSL has a restricted domain, it is possible to provide some, if not all, of the

desired abstractions. Many DSLs do not provide general-purpose abstraction mechanisms because it is often possible to define a fixed set of abstractions that are sufficient for all the applications in a domain. Hence, we can expect that some DSALs will use fixed and predefined pointcuts and advice with limited possibility for general abstraction. On the other hand, a DSAL that is applied to a general-purpose component language should have more sophisticated constructs for pointcuts and advice. For example, pointcuts should be generic, reusable, comprehensible and not tightly coupled to an application's structure. Tourwe et al. proposed an annotation of inductively generated pointcuts as a solution to this problem [34].

4. AspectLISA

This section describes our investigation into applying aspects to our own language definition tool called LISA. The first subsection introduces LISA and is followed by a discussion of how AspectLISA extends a LISA language specification through aspects that crosscut the language definition.

4.1 LISA: A Domain-Specific Component Language

LISA [22] is a tool that automatically generates a compiler and other language related tools from formal language specifications. The LISA specification notation that is used to define a new language is based on multiple attribute grammar inheritance [25], which enables incremental language development and reusability of specifications. The LISA specification language consists of regular definitions, attribute definitions, rules (which are generalized syntax rules that encapsulate semantic rules), and methods on semantic domains.

The lexical part of a new language definition is denoted by the reserved word **lexicon**. From this part, LISA generates Java source code that implements a scanner for the defined lexicon. Tokens are defined using named regular expressions. Each regular expression has a unique name and can be extended or redefined in a derived language. In the example below, we have two regular definitions: `Commands` and `ReservedWord`. Reserved word **ignore** is used to define characters and tokens that are ignored by the scanner (i.e., not included in the token list). The syntax and semantic parts of a language specification are encapsulated into generalized LISA rules (denoted by the reserved word **rule**). LISA follows the well-known standard BNF notation for defining the syntax of a programming language. Context-free productions are specified in the rule part of a language definition (e.g., `START ::= begin COMMANDS end`). Generalized LISA rules serve as an interface for language specifications and may be extended through inheritance. A new language specification inherits the properties of its ancestors and may introduce new properties that extend, modify or override its inherited properties. The semantic part of a language specification is defined by an attribute grammar. Semantic actions must be provided for every production in the **compute** part of a context-free production. To pass values in the syntax tree, non-terminals have attributes. Semantic rules (i.e., attribute calculations) are defined in Java (i.e., the right-hand side of the semantic equation).

In order to illustrate the LISA specification language, the definition of a toy language for robotic control is given below. The robot can move in different directions (left, right, down, up) and the task is to compute its final position. An example of the program is **begin** `up right up` **end** with the meaning {`outp.x=1, outp.y=2`}.

```
language Robot {
lexicon {
  Commands left | right | up | down
  ReservedWord  begin | end
  ignore [\0x0D\0x0A\ ]   // skip whitespaces
}
attributes Point *.inp, *.outp;

rule start {
 START ::= begin COMMANDS end  compute {
   START.outp = COMMANDS.outp;
    // robot position in the beginning
   COMMANDS.inp = new Point(0, 0); };
 }

rule moves {
 COMMANDS ::= COMMAND COMMANDS  compute {
   COMMANDS[0].outp = COMMANDS[1].outp;  // propagation of position
   COMMAND.inp = COMMANDS[0].inp;        // to sub-commands
   COMMANDS[1].inp = COMMAND.outp; }
 | epsilon  compute {  // epsilon (empty) production
   COMMANDS.outp = COMMANDS.inp; };
 }

rule move {
 // each command changes one coordinate
 COMMAND ::= left  compute {
   COMMAND.outp = new Point((COMMAND.inp).x-1,(COMMAND.inp).y); };
 COMMAND ::= right  compute {
   COMMAND.outp = new Point((COMMAND.inp).x+1,(COMMAND.inp).y); };
 COMMAND ::= up  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y+1); };
 COMMAND ::= down  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y-1); };
 }
}
```

From this language specification, LISA generates highly efficient Java source code that represents the scanner/parser/compiler of the defined language. Additional information about LISA (including software, tutorial, and examples), can be found on LISA's web page [22] and in [25, 26].

4.2 AspectLISA: A Domain-Specific Aspect Language

In language specification there are situations when new semantic aspects crosscut basic modular structure. For example, some semantic rules have to be repeated in different productions; i.e., the introduction of an assignment statement requires variables, which imply definition of the environment and its propagation in all the defined productions. We also identified some other crosscuting concerns, as described in Section 1.

In consideration of the questions stated in Section 3 regarding the JPM for specific domains, join points in AspectLISA are static points in a language specification where additional semantic rules can be attached. These points can be syntactic production rules or generalized LISA rules. A set of join points in AspectLISA is described by a pointcut that matches rules/productions in the language specification. To define a pointcut in AspectLISA, two different wildcards are available. The wildcard '..' matches zero or more terminal or non-terminal symbols and can be used to specify right-hand side matching rules. The wildcard '$*$' is used to match parts or whole literals representing a symbol (terminal or non-terminal symbol). Some examples of pointcut specifications are shown below:

`*.* : * ::= .. ;`
matches any production in any rule in all languages across the current language hierarchy

`Robot.m* : * ::= .. ;`
matches any production in all rules which start with `m` in the `Robot` language

`Robot.move : COMMAND ::= left ;`
matches only a production `COMMAND ::= left` in the rule `move` of the `Robot` language

Pointcuts in AspectLISA are defined using the reserved word **pointcut**. Each pointcut has a unique name and a list of actual parameters (terminals and non-terminals) that denote the public interface for advice. An example of a pointcut that identifies all productions with `COMMAND` as the left-hand non-terminal is:

```
pointcut Time<COMMAND> *.move : COMMAND ::= * ;
```

In AspectLISA, advice are parameterized semantic rules written as native Java assignment statements that can be applied at join points specified by a pointcut. Advice defines additional semantics (extension/refinement) and does not impact the structural (syntax) level of a language specification. In AspectLISA, there is only one way to apply advice on a specific join point due to the fact that attribute grammars are declarative. The order of semantic rules is calculated during compilation/evaluation time when dependencies among attributes are identified. Therefore, applying advice before/after a join point is not applicable. For the same reason ordering of different aspects is not necessary.

Suppose that the Robot language needs to be extended to incorporate the concept of time. An example advice for time calculation that is applied on join points specified by pointcut `Time` is:

```
advice TimeSemantics<C> on Time { C.time=1; }
```

After weaving takes place, the semantic function `COMMAND.time=1;` is added to all productions within rule `move`. In addition, advice in AspectLISA can have an **apply** part for applying predefined semantic patterns such as: value distribution, list distribution, value construction, list construction, bucket brigade, and propagate value. These semantic patterns are common in attribute grammars and represent fixed abstractions. For example, to propagate environment attributes over an entire evaluation tree, the semantic pattern `bucketBrigadeLeft` should be used (`inEnv` and `outEnv` are attributes used to store and propagate the environment).

```
pointcut All<> *.* : * ::= .. ;
advice EnvProp<> on All apply bucketBrigadeLeft(inEnv, outEnv) { }
```

Modularity, reusability and extensibility of language specification have been much improved in LISA using multiple attribute grammar inheritance [25]. In AspectLISA, pointcuts and advice are also subjects of inheritance. All pointcuts of predecessors can be used in all ancestors. Pointcuts with the same signature (name and parameters) as in ancestors can be used but cannot be extended in inherited languages. Such pointcuts are overriden by default. Advice inherited from ancestors using the **extends** keyword must be merged with the advice in the specific language. If advice exists in the inherited parent language, then the semantic functions of the advice must be merged; otherwise, advice are simply copied from the inherited language to the current language. Advice can also override the semantics of its parent using the keyword **override**.

An example of inheritance on advice is shown below. Note that the pointcut on which this advice is applied is inherited.

```
advice extends TimeSemantics<C> {
C.time=1.0 / C.inspeed; C.outSpeed = C.inspeed; }
```

4.2.1 Aspect Weaving in AspectLISA

The crucial part of every aspect-oriented compiler is an aspect weaver that is responsible for appending advice code into appropriate places described by pointcuts.

Weaving takes place after the initial phase of LISA's compiler, which is responsible for parsing the LISA source and generating the necessary data structures for pointcuts and advice. The main weaving algorithm is described by Algorithm 1.

Algorithm 1 Main weaving algorithm

```
method weaveAll(lastLanguage)
// lastLanguage is last language in hierarchy
Language_list ← allDefinedLanguages
for all L ∈ Language_list do
  L ← nextElement(Language_list)
  // if L is not part of language hierarchy the weaving
  // in that Language is not necessary
  if L is reachable from lastLanguage then
    weave(L)
  end if
end for
```

Weaving starts at the first (parent) language (component) defined by the developer and follows its hierarchy. The same algorithm is applied to each language specification over the entire hierarchy of languages. The weaving procedure for each user-defined language is described by Algorithm 2. Note that the lookup method (*pointcutLookup(A, L)*) works the same as in most compilers for object-oriented languages.

Algorithm 2 Weaving algorithm for one Language

```
method weave(L)
Advice_set ← getAllAdvice(L)
for all A ∈ Advice_set do
  A ← nextElement(Advice_set)
  // advice must not be overriden by none of its successors
  if A is not overriden then
    // find appropriate pointcut in current or parent languages
    pointcut ← pointcutLookup(A, L);
    // find all production rules that match pointcut
    productionRules ← findProductions(pointcut, L);
    for all prodRule ∈ productionRules do
      prodRule ← nextElement(productionRules)
      // substitute formal parameters of advice with actual
      // parameters of pointcut and apply semantic functions
      // to the production
      addSemanticsToRule(prodRule, A, L)
    end for
  end if
end for
```

5. AspectG

This section describes our second investigation into a DSAL for language specification. The first subsection introduces ANTLR as the DSL representing the component language. The second subsection provides a discussion of AspectG, which is our DSAL that weaves crosscutting concerns into ANTLR grammars.

5.1 ANTLR: A Domain-Specific Component Language

ANTLR (ANother Tool for Language Recognition) is a parser generator that provides a framework for constructing various programming language related tools (e.g., recognizers, compilers, and translators) from grammatical specifications [1]. The ANTLR specification language is based on EBNF notation and enables syntax-directed generation of a compiler. The tokens comprising the lexical part of the grammar for the new language are defined using named regular expressions. The parser representing the semantic part of the language specification is defined as a subclass of the grammar specification and encapsulates semantic rules within each

grammar production. The semantic actions within each production rule are written in a GPL (e.g., Java, C#, C++, or Python). The Robot language described in Section 4 has been rewritten in ANTLR and partially provided below. This simple example illustrates the ANTLR specification language with semantic rules defined in Java. AspectG follows the DSL implementation pattern using a pre-processor that serves as a compiler and application generator to perform a source-to-source transformation (i.e., the DSL source code is translated into the source code of an existing GPL). The ANTLR specification of the Robot language translates Robot code into the equivalent Java code (e.g., Robot.java) that can be compiled and executed on the Java Virtual Machine.

```
// The following class represents the Robot parser in ANTLR
class P extends Parser; {FileIO fileio=new FileIO();}
root:(
       BEGIN
       {
       fileio.print("public class Robot");
       fileio.print("{");
       fileio.print("public static void main(String[] args) {");
       fileio.print("int x = 0;");
       fileio.print("int y = 0;");
       }
      commands
      END EOF!
       {
       fileio.print("System.out.println(\"x coord= \" + x +
                    \" \" + \"y coordinator= \" + y);");
       fileio.print("   }");
       fileio.print("}");
       fileio.end();
       }
      );
commands:(  command commands
         |
          );
command :(
          LEFT  {fileio.print("x=x-1;");
                 fileio.print("time=time+1;");}
         |RIGHT {fileio.print("x=x+1;");
                 fileio.print("time=time+1;");}
         |UP    {fileio.print("y=y+1;");
                 fileio.print("time=time+1;");}
         |DOWN  {fileio.print("y=y-1;");
                 fileio.print("time=time+1;");});

// The following class represents the Robot lexer in ANTLR
class L extends Lexer;
BEGIN : "begin";
END   : "end";
LEFT  : "left";
RIGHT : "right";
UP    : "up";
DOWN  : "down";
// whitespace
WS  :   (   ' '
        |   '\t'
        |   '\r' '\n' { newline(); }
        |   '\n' { newline(); }
        )       {$setType(Token.SKIP);} ;
```

From the above language specification, ANTLR generates Java source code representing the scanner and parser for the Robot language. Additional information about ANTLR can be found on the ANTLR web page [1].

5.2 AspectG: A Domain-Specific Aspect Language

In our past work [36], we noticed that crosscutting concerns emerged within the grammar of the language specification. In particular, the implementation hooks for various language tools (e.g., debugger and testing engine) required modification to be made to every production in the grammar. Manually changing the grammar through invasive modifications proved to be a very time consuming and error prone task. It is difficult to build new testing tools for each new language of interest and for each supported platform because each language tool depends heavily on the underlying operating system's capabilities and lower-level native code functionality [29].

We developed a general framework called the DSL Testing Tool Studio (DTTS), which assists in debugging, testing, and profiling a program written in a DSL. Using the DTTS, a DSL debugger and unit test engine can be generated automatically from the DSL grammar provided that an explicit mapping is specified between the DSL and the translated GPL. To specify this mapping, additional semantic actions inside each grammar production are defined. A crosscutting concern emerges from the addition of the explicit mapping in each of the grammar productions. The manual addition of the same mapping code in each grammar production results in much redundancy that can be better modularized using an aspect-oriented approach applied to grammars. In the case of generating a debugger for the Robot language, the debug mapping for the Robot DSL debugger was originally specified manually at the Robot DSL grammar level shown below. For example, line 12 to line 18 represents the semantic rule of the LEFT command. Line 12 keeps track of the Robot DSL line number; line 14 records the first line of the translated GPL code segment; line 16 marks the last line of the translated GPL code segment; line 17 and line 18 generate the mapping code statement used by the DTTS. These semantic actions are repeated in every terminal production of the Robot grammar.

```
10  command
11   :( LEFT {
12       dsllinenumber=dsllinenumber+1;
13       fileio.print(" x=x-1;");
14       gplbeginline=fileio.getLinenumber();
15       fileio.print(" time=time+1;");
16       gplendline=fileio.getLinenumber();
17       filemap.print("mapping.add(newMap(" + dsllinenumber +
                        ",\"Robot.java\"," +
18                    gplbeginline + "," + gplendline + "));");}
19     |RIGHT {
20       dsllinenumber=dsllinenumber+1;
21       fileio.print(" x=x+1;");
22       gplbeginline=fileio.getLinenumber();
23       fileio.print(" time=time+1;");
24       gplendline=fileio.getLinenumber();
25       filemap.print("mapping.add(newMap(" + dsllinenumber +
                        ",\"Robot.java\"," +
26                    gplbeginline + "," + gplendline + "));");}
```

The same mapping statements for the RIGHT command appear in lines 20, 22, and 24 to 26. Although the Robot DSL is simple, it is not uncommon to have grammars with hundreds of production rules. In such cases, much redundancy will exist because the debug mapping code is replicated across each production. Of course, because the debug mapping concern is not properly modularized, changing any part of the debug mapping has a rippling effect across the entire grammar. An aspect-oriented approach can offer much benefit in such a case. We have created AspectG as a tool to help us manage crosscutting concerns in ANTLR language specifications.

The AspectG pointcut model can match on both the syntax of the grammar and the semantic rule within each production (written in Java). Join points in ANTLR are static points in the language specifications where additional semantic rules can be attached. A set of join points in AspectG is described with pointcuts that define the location where the advice is to apply. A wildcard can be used within the signature of a pointcut. The wildcard '$*$' matches zero or more terminal or non-terminal symbols to represent a set of qualified join points. Some examples of pointcut specifications are shown below:

`*.*;` *matches any production in the entire Robot language*
`command.*;` *matches any production in a command production in the Robot language*

Pointcuts in AspectG are defined using the reserved word **pointcut** and two keywords (e.g., **within** and **match**). The **within** predicate is used to locate grammar productions at the syntax level and **match** is used to define the location of a GPL statement within a semantic rule. Each pointcut has a unique name and a list of actual parameter signatures (terminals and non-terminals) and semantic rules. Considering the following pointcut:

```
pointcut productions(): within(command.*);
```

The pointcut called `productions` is defined with the wild card `command.*` and matches each join point that is a command production in a grammar (e.g., RIGHT). As an example of a pointcut that combines both predicate types, consider the following:

```
pointcut count_gpllinenumber(): within(command.*) &&
match(fileio.print("time=time+1;"));
```

The pointcut `count_gpllinenumber` is a pattern specification corresponding to command productions having a semantic action with a statement matching the signature `fileio.print("time=time+1;")`. The advice in AspectG is defined in a similar manner to AspectJ, which brings together a pointcut that selects join points and a body of code representing the effect of the advice [4]. The advice are semantic rules written as native Java statements that can be applied at join points specified by pointcuts. Unlike LISA, in ANTLR the order of GPL statements in semantic rules is very important. Therefore, in AspectG the ability to apply advice before/after a join point is necessary, as shown in the example below.

```
before(): productions() { dsllinenumber=dsllinenumber+1;}
after(): count_gpllinenumber() {
gplendline=fileio.getLinenumber();
}
```

The **before** advice defined on the `productions` pointcut means that before the parser proceeds with execution of each command production, the DSL line number is incremented (i.e., `dsllinumber=dsllinenumber+1;`). The **after** advice associated with the `count_gpllinenumber` means that line numbers for the GPL are updated (i.e., `gplendline=fileio.getLinenumber();`) after the parser matches a timer increment (i.e., `fileio.print ("time=time+1;");`).

The changes in terms of aspects automatically propagate into the generated parser through the modified grammar productions. After weaving a grammar aspect and parsing the Robot DSL code, the new ANTLR grammar can generate the mapping information that contains the information needed by the DTTS (i.e., each Robot DSL code statement line number along with its corresponding generated Java statement line numbers is recorded in the grammar).

5.2.1 Aspect Weaving in AspectG

Unlike AspectLISA's compiler approach, AspectG uses a program transformation system (specifically, we use DMS - the Design Maintenance System [6]) to perform the underlying weaving on the language specification. The AspectG abstraction hides the details of the accidental complexities of using the transformation system from the users; i.e., a user of AspectG focuses on describing the crosscutting grammar concerns at a higher level of abstraction using an aspect language, rather than writing lower level program transformation rules [36]. In AspectG, each of the crosscutting concerns is modularized as an aspect that is weaved into an ANTLR grammar using parameterized low-level transformation functions.

We have developed four weaving functions to handle four different types of join points that may occur within a grammar. The four possible join points provided by AspectG are: before a semantic action; after a semantic action; before a specific statement that is inside a semantic action; and, after a specific statement that is inside a semantic action. These joint points are represented in AspectG by **before** and **after** keywords within the context of a semantic action or specific statement. Weaving takes place after the initial phase of AspectG's compiler, which is responsible for parsing the AspectG specification and generating the program transformation rules. The generated program transformation rules provide bindings to the appropriate weaving function parameters corresponding to the pointcut and advice defined in the aspect language. Algorithm 3 describes the weaving procedure for AspectG. Note that the algorithm requires two parameters (advice and a join point) and weaves the advice parameter into the join points designated by the pointcut predicate. Additional technical details are provided in [11].

Algorithm 3 AspectG weaving

```
for all jp ∈ pointcuts_list do
  for all a ∈ advices_list do
    if jp's name equals a's pointcut name then
      weave(jp, a)
    end if
  end for
end for
```

The actual weaving of the language specification is done by the DMS program transformation engine according to the different program transformation rules generated by the AspectG compiler. The weave method first looks for all potential pointcut positions in the semantic sections of a grammar. The weaver then back tracks to the pointcut's ancestor node type and value in the syntax level to filter out the unqualified pointcut positions. Finally, the advice is inserted in the correct position of the grammar specification using the ASTInterface API provided by DMS, which provides methods for modifying a given syntax tree to regenerate a new tree structure.

6. Related work

AspectASF [21] is a simple DSAL for language specifications written in the ASF+SDF formalism. Only rewrite rules are supported. Therefore, join points in AspectASF are static points in equation rules describing semantics of the language. The pointcut pattern language in AspectASF is a simple pattern matching language on the structure of equations where only labels and left-hand sides of equations can be matched. Pointcuts can be of two types: entering an equation (after a successful match of left-hand side) and exiting an equation (just before returning the right-hand side). Advice specify additional equations that are written in the ASF formalism. The AspectASF weaver transforms the original language specifications by augmenting the base grammar with new concerns (i.e., additional equations are appended to appropriate places in the grammar).

An early approach of aspect-orientation in language specifications is presented in the compiler generator system JastAdd [15]. The JastAdd system is a class weaver: it reads all the JastAdd modules (aspects) and weaves the fields and methods into the appropriate classes during the generation of the AST classes. This approach does not follow the conventional join point model where join points are specified using a pointcut pattern language. However, it can be seen as inter-type declarations in AspectJ where join points are all non-anonymous types in the program and pointcuts are the names of classes or interfaces. Hence, JastAdd uses implicit joint points while AspectLISA and AspectG use explict joint points described by pointcuts. Moreover, JastAdd does not enable inheritance on advice and pointcuts as AspectLISA does.

7. Conclusion

Domain-specific aspect languages (DSALs) represent a focused approach toward providing a language that allows a programmer or end-user to define a specific type of concern. DSALs can be contrasted with general-purpose aspect languages (GPALs) that provide a more general language for capturing a broader range of crosscutting concerns. Within the research on DSALs, much of the application is centered on specific concerns for a language like Java or C++. This paper differs from the scope of general research by describing our investigation into DSALs for DSLs such as language specification.

The paper summarized the challenges of DSAL development and presented two separate case studies of different DSALs applied to two different languages. Future work includes new pointcut predicates that assist in specifying the control flow within a grammar. Such a predicate would allow aspects associated with various forms of run-time analysis to be specified and captured.

References

[1] ANTLR – ANother Tool for Language Recognition. http://www.antlr.org, 2006.

[2] AspectC. http://www.aspectc.org/, 2006.

[3] AspectC#. http://www.castleproject.org/index.php/aspectsharp, 2006.

[4] AspectJ. http://eclipse.org/aspectj/, 2006.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[6] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformation for practical scalable software evolution. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 625–634, 2004.

[7] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of Joint European Software Engineering Conference (ESEC)*, pages 88–98, 2001.

[8] T. Cottenier and T. Elrad. Contextual pointcut expressions for dynamic service customization. In *Dynamic Aspects Workshop (DAW)*, pages 95–99, 2005.

[9] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 69–77, 2005.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[11] DSL Testing Tool Studio. http://www.cis.uab.edu/wuh/ddf/, 2006.

[12] J. Fabry and T. Cleenewerck. Aspect-oriented domain-specific languages for advanced transaction management. In *Proceedings of International Conference on Enterprise Information Systems (ICEIS)*, pages 428–432, 2005.

[13] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, pages 87–93, October 2001.

[14] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software factories: assembling applications with patterns, models, frameworks and tools*. Wiley Publishing, 2004.

[15] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[16] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using LISA. *IEE Proceedings - Software Engineering*, 152(2):54–69, April 2005.

[17] C. Herzeel, K. Gybels, and P. Costanza. A temporal logic language for context awareness in pointcuts. In *ECOOP Workshop: Revival of Dynamic Languages*, 2006.

[18] J. Hugunin. The next steps for aspect-oriented programming languages. In *NSF Workshop on Software Design and Productivity*, 2001.

[19] J. Irwin, J. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments*, 1997.

[20] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. *Electr. Notes Theor. Comput. Sci*, 147(1):5–30, 2006.

[21] P. Klint, T.van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical report, CWI, 2004.

[22] LISA. http://marcel.uni-mb.si/lisa, 2006.

[23] C. Lopes. *Aspect-Oriented Programming: A Historical Perspective*. In Aspect-Oriented Software Development, R. Filman, T. Elrad, M. Aksit, S. Clarke (eds.), Addison-Wesley, 2004.

[24] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[25] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, 2000.

[26] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In *Proceedings of International Conference on Compiler Construction (CC)*, pages 1–4, 2002.

[27] M. Mernik, X. Wu, and B. Bryant. Object-oriented language specifications: Current status and future trends. In *ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004.

[28] D. Rebernak, M. Mernik, P. R. Henriques, and M. J. V. Pereira. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. In *Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 44–61, 2006.

[29] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley and Sons, 1996.

[30] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 28–37, 2003.

[31] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An aspect-based composition tool for real-time systems. In *Real-Time Applications Symposium*, pages 58–69, 2003.

[32] D. Suvee, W. Vanderperren, and V. Jonckers. Jasco: An aspect-oriented approach tailored for component based software development. In *Proceedings of International Conference on Aspect-oriented Software Development (AOSD)*, pages 21–29, 2003.

[33] E. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *Proceedings of International Symposium on Software Composition*, pages 227–249, 2006.

[34] T. Tourwe, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse. Inductively generated pointcuts to suppport reafctoring to aspects. In *AOSD Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2004.

[35] D. A. Watt. *Programming Language Concepts and Paradigms*. Prentice-Hall, 1990.

[36] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 1370–1374, 2005.

Post Facto Type Extension for Mathematical Programming

Stephen M. Watt

Department of Computer Science
University of Western Ontario
London ON, Canada N6A 5B7
watt@csd.uwo.ca

Abstract

We present the concept of *post facto extensions*, which may be used to enrich types after they have been defined. Adding exported behaviours without altering data representation permits existing types to be augmented without renaming. This allows large libraries to be structured in a clean, layered fashion and allows independently developed software components to be used together. This form of type extension has been found to be particularly useful in mathematical software, where often new abstractions are applicable to existing objects. We describe an implementation of post facto extension, as provided by *Aldor*, and explain how it has been used to structure a large mathematical library.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Methodologies; D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Special-purpose algebraic systems

General Terms Design, Languages

Keywords Aldor, Axiom, Aspect-oriented programming, Symbolic computation, Computer algebra

1. Introduction

As software libraries are extended and combined, it is often desirable to view values of pre-existing types as instances of more general abstractions defined later. This leads either to defining a host of conversions, or to re-writing libraries. Conversions may be either implicit or explicit, but in either case the programmer must be aware of them and use compilers that can optimize them. Re-writing libraries to endow pre-existing types with later-defined semantics is time-consuming and decreases modularity.

This paper presents a programming language solution to this problem. The solution, "*post facto extension*," is a specialized instance of what is today known as aspect-oriented programming, and it has proven highly effective in structuring mathematical libraries for *Aldor* [1, 2, 3, 4]. Language support for post facto extension can be readily added to object oriented or abstract datatype programming languages without the complexity of full support for general aspect-oriented programming.

This work has been motivated by the design of software for computer algebra, an area concerned with answering mathematical problems in terms of symbolic expressions rather than numbers. In mathematics, as in software development, one of the principal activities is that of generalization. By expressing problems more abstractly, it is possible to make greater re-use of previous work. From this point of view, it is quite natural to view previously defined quantities as special instances of newly defined abstractions.

A simple example illustrates this point: Suppose we are developing a library, and one of the types is `Integer`. We provide this type with the basic arithmetic operations, $+$, $-$, $\times$, $=$, $<$, *etc*. Later, `gcd` and `lcm` are added to the library. Should the type `Integer` be modified to export these operations, or should they remain as independent functions? If additional arithmetic types are added, it may be desirable to add a `Ring` abstraction, from which all types with suitable arithmetic can inherit. Types that could provide the `Ring` interface include square matrices, polynomials, quotient fields, complex numbers and the integers. Should `Integer` be modified to export `Ring`? If `Integer` *is not* modified, then `Integer` values cannot be used where elements of a `Ring` are required. It is then necessary to introduce a new type and provide conversions. As more abstractions are added, many conversions are used either explicitly or implicitly and code becomes cumbersome and inefficient. If `Integer` *is* modified, then a new dependency is created and previously complete components must be re-tested. It is also possible that adding this behaviour to the type will break third-party uses of the library. When more than one library is involved, this problem is exacerbated.

The problem of dealing with new abstractions for existing objects is by no means restricted to mathematical computation. It arises whenever multiple libraries provide different functionality for basic types in an object-oriented environment. An example would be when different libraries provide string manipulation, regular expression matching and higher-level text operations. Situations such as this are now well-understood in the aspect-oriented programming community. We have found this problem to be particularly acute in the construction of computer algebra software: *In this setting, it is the usual case that most basic types are instances of many later-defined abstractions.* Moreover, since there is wide agreement on numerous mathematical abstractions, it is quite natural to expect the objects of one library to simply work in other libraries.

The situation where we first noticed the problem was in the construction of libraries for the *Axiom* [5] computer algebra system. Given that the basic arithmetic types needed to participate in advanced mathematical operations, there seemed to be no way to de-

DSAL 2006 Portland OR, USA.

fine a fixed core language with a few basic types and an evolving library of advanced functionality. During this period, the present author was responsible for the design and implementation of the programming language to be used for libraries to extend the *Axiom* system. We were thus in the fortunate position to consider programming language solutions to problems that arose in library design.

Our solution to the problem of dealing with new abstractions for existing components relies on a key observation: *Although it is desirable to add new interfaces to mathematical types after they have been defined, it is usually not desirable to change the representation of values.* We found it was almost always the case that any new operations required by the new interfaces could be defined in terms of existing exported behaviour without any change to the object representation. This led to the idea that existing values could participate in new interfaces without any changes to the objects at all. Instead, higher-order operations on the types could add the desired behaviours. This is the basic idea of what we call "post facto extension" of types.

We have explored this idea in our design of the *Aldor* programming language and have found it to be quite effective in cleanly structuring complex mathematical libraries with many rich relations among the types. In *Aldor*, the expression of post facto extensions is quite simple. From the programmer's point of view, there is little required to use them effectively. We believe that these ideas may prove useful in areas outside of mathematical programming and therefore should be more widely known.

This paper presents the main ideas of post facto type extension and describes how it has been used to structure complex libraries: Section 2 outlines the main ideas of *Aldor* and its type system. Section 3 describes structural problems that were observed in building mathematical libraries for *Aldor*. Section 4 then presents our solution, *post facto extensions*. Section 5 explains some of the ways that post facto extensions can be used in structuring large libraries. We present our conclusions in Section 6.

2. *Aldor* and Its Type System

Aldor [1, 2, 3, 4] is a programming language originally intended to provide compiled libraries for computer algebra. The design of the language tries to balance high-level mathematical expressivity with the possibility of compilation to efficient machine code so large symbolic and numeric problems can be treated. There are several aspects to the *Aldor* language that are intended to provide support for mathematical programming, but which are somewhat unusual. We outline these below.

Types and functions are first-class values. This means that they may be created and used dynamically, providing representations for mathematical sets and functions.

The type system has two levels. Each value belongs to some unique type, known as its *domain*, and the domains of values can be delcared statically. Domains themselves belong to the domain `Type`. Domains may additionally belong to type *categories* that specify additional properties. In particular, categories may specify that a domain must export certain operations or that some operations have default implementations. Categories fill the role of interfaces or abstract base classes of other languages, and may be viewed as sub-types of the domain `Type`. Category membership can be asserted at compile time and tested at run time.

The language is not object-oriented. There are a number of aspects of object-oriented programming that make it awkward to use in an algebraic setting:

The first problem is that object-oriented languages favour a programming style where objects maintain state and the execution of a program consists of calling methods to change that state. Mathematical programming is more suited to a functional style, where one works with values and functions compute new values based on their arguments and where values are seldom, if ever, modified.

The second problem is that, in an object-oriented world, binary operations do not inherit in a natural way. In mathematics it is quite common to have functions that are homogeneous on their arguments, for example $+$, $\times$, $-$, $=$ and $<$. To illustrate the difficulty with object-oriented inheritance, suppose we have a base class B with a method plus, used as `a.plus(b)` to add a value of type B to an object of type B and yielding new value of type B. That is, `plus` : $B \times B \to B$. If class D is derived from class B then it will have `plus` : $D \times B \to B$. This problem was already noted by Barbara Liskov as arising in the design of *CLU* [6] and is cited as one of the reasons that the language was based on abstract data types rather than objects.

The third problem is related to the second. Class-based inheritance does not provide sufficient static type checking for multiple-argument functions. To illustrate, suppose that a base class B provides the abstraction of multiplication $\times : B \times B \to B$ and that classes D_1 and D_2 are derived from it. We wish to ensure statically that the multiplications $D_1 \times D_1$ and $D_2 \times D_2$ are allowed, but that the multiplications $D_1 \times D_2$ and $D_2 \times D_1$ are disallowed. In an object-oriented world, however, the inhomogeneous multiplications would be allowed by virtue of the inherited multiplication defined in class B.

An example can illustrate this last point. In *Aldor* one can define a category `Semigroup` to capture the abstraction of a homogeneous multiplication, and the domains `DoubleFloat` and `Permutation` could be declared to belong to this category.

```
Semigroup:   Category == with { *: (%, %) -> % }
DoubleFloat: Join(Semigroup, ...) == ...
Permutation: Join(Semigroup, ...) == ...
```

This causes the declared domains to export a suitable multiplication. For example, `DoubleFloat` will have an exported operation "`*`" that takes two `DoubleFloat` values and returns a third. If `x` and `y` are declared to be of type `DoubleFloat` and `p` and `q` are declared to be of type `Permutation`, then it will be possible to multiply `x*y` and `p*q`, but not `x*p`. This difference may be summarized by the following relations:

$$\left.\begin{array}{lllll} \texttt{x, y} & \in & \texttt{DoubleFloat} & \subset & \texttt{Semigroup} \\ \texttt{p, q} & \in & \texttt{Permutation} & \subset & \texttt{Semigroup} \end{array}\right\}\text{OOP}$$

$$\left.\begin{array}{lllll} \texttt{x, y} & \in & \texttt{DoubleFloat} & \in & \texttt{Semigroup} \\ \texttt{p, q} & \in & \texttt{Permutation} & \in & \texttt{Semigroup} \end{array}\right\}\text{Aldor}$$

In an object orient world `x` and `p` belong to a common inherited class, but in *Aldor* they do not.

Dependent types are fully supported. *Aldor* obtains the capabilities of object-oriented programming through the use of dependent types. Tuples may have components whose value determines the type of other components and mappings may have return types that depend on the values of parameters. As an example, consider the following declaration:

```
f: (n: Integer, m: SquareMatrix(n, Integer))
   -> List IntegerMod(n)
```

Here we suppose that `SquareMatrix(n, Integer)` is the type of $n \times n$ square matrices with integer entries and that `IntegerMod(n)` is a type representing the integers modulo n. The *types* of the second argument and of the return value of `f` depend on the *value* of the first argument. If the first argument is 3, then the second argument must be a 3×3 matrix and the result will be a list of integers modulo 3.

Dependent types are particularly useful when some of the components are themselves types. For example, we may define

```
prodl: List Record(S: Semigroup, s: S) == [
   [DoubleFloat, x],
   [Permutation, p],
   [DoubleFloat, y]
]
```

Here each element of the list consists of a type and a value belonging to that type. By specifying that the type belong to a particular category, we are able to determine statically what operations are supported on the values. In *Aldor*, use of dependent types and inheritance in the category hierarchy take the place of objects and inheritance in the class hierarchy.

Parametric polymorphism is provided by category- and domain-producing functions. With types as first class values and dependent types fully supported, functions producing types take the place of templates in other languages. For example we may write

```
define Module(R: Ring): Category == Ring with {
    *: (R, %) -> %
}

Complex(R: Ring): Module(R) with {
    complex: (%, %) -> R;
    real: % -> R;
    imag: % -> R;
    conjugate: % -> %;
    ...
} == add {
    Rep == Record(real: R, imag: R);
    ...
}
```

Here, `Module` is a function that take a type parameter, `R`, belonging to the category `Ring` and returns a category as its result. The form `Ring with {*: (R, %) -> R}` constructs the category to be returned as being the category `Ring` extended with one additional operation. The symbol `%` in the category expression refers to the domain that exports the category. If `D: Module(T)`, then `D` exports `*: (T, D) -> D`. The keyword `define` affects the publicly visible information about `Module` that will be visible about compilation units. It allows not only its type, `(R: Ring) -> Category`, but also its value, `(R:Ring) +-> Ring with {*:(R,%) -> %}`, to be publicly visible.

The second definition declares `Complex` to have a dependent mapping type, `(R: Ring) -> Module(R) with...`. That is, `Complex` takes a type-valued parameter `R` that belongs to the category `Ring` and returns a type-valued result that belongs to the category `Module(R) with...`. The body of the function definition (the part after "==") is a form that constructs a domain.

Category- and domain-producing expressions may be conditional. *Aldor* provides conditional inheritance, allowing types to be formed differently according to run-time conditions. For example, we may write

```
UnivariatePolynomial(R: Ring): Module(R) with {
    coeff: (%, Integer) -> R;
    monomial: (R, Integer) -> %;

    if R has Field then EuclideanDomain;
    ...
} == add {
    ...
}
```

That is, if the type parameter `R` to `UnivariatePolynomial` not only belongs to the category `Ring` but also belongs to the category `Field`, then the type `UnivariatePolynomial(R)` also belongs to the category `EuclideanDomain`.

Post facto extensions. *Aldor* allows domains to be extended to belong to new categories after they have been initially defined. These allow domains to be defined in a layered fashion, separating issues and eliminating dependencies, while providing rich function. These are the focus of the present paper and are described in more detail in Section 4.

Aldor grew out of an earlier language by Jenks and Trager [7] that already used the idea of domains and categories. This language was the original library language for the *Axiom* system (then known as *Scratchpad II*), and inspired a number of other projects for computer algebra languages, including *Newspeak* [8] and *Views* [9].

3. Problems in Library Design

We now describe a certain problems that we observed in building the first *Aldor* libraries. We describe some of these problems using the terminology of *Aldor*, but their translation into other programming languages should be straightforward. Later we show how these problems are solved with post facto extensions.

Old domains and new categories. We can now revisit the example of the introduction using more precise language: In building libraries for mathematical computation, it is quite normal to define new categories and to find that existing domains could be made to belong to them. Many of the most basic types, such as `Integer`, `IntegerMod(p)`, `Fraction(R)`, `Complex(R)`, `Matrix(n,m,R)` and `UnivariatePolynomial(R)`, have a wealth of mathematical properties and are often potential instances of newly defined categories. The same thing is true for floating point types if one is willing to overlook the fact that they are not exactly associative.

In building the basic *Aldor* libraries, there was the choice of whether to modify these basic domains to belong to all the applicable categories defined in the standard libraries, or whether to maintain modularity. On the one hand, even within the standard libraries, modularity was desirable. Certain types, such as `Integer` and `Boolean` must be known in the language definition and it would be injudicious to therefore have to fix an intricate hierarchy of algebraic categories as part of the basic language. Even if these basic domains were modified to belong to all the applicable categories in the standard libraries, then the problem of membership in categories from new libraries would still exist. On the other hand, if the basic domains were not made to belong to the categories of the standard libraries, then values belonging to these domains could not be used by any of the advanced functions. The solution of having a basic and an elaborated version of each type would lead to code filled with distracting explicit conversions or subtly dangerous implicit conversions.

Difficulties with multiple libraries. Commonly, application must work with objects that inherit from base classes or interfaces from independent libraries. There is the problem, however, that objects returned by methods of one library are not suitable for use in calls to methods of other libraries. One solution is for the application to build its objects as compound structures containing component objects from the separate libraries. In this case conversions and constructors are used to move between the types. Sometimes an application will define a new base class for its hierarchy that inherits from both libraries as a way to deal with this situation. Then clients of the application that require yet other third libraries must repeat the process. This is really just another instance of old types lacking new interfaces. except in this case the interfaces come from separate libraries and there is no real possibility of integrating the set of types.

Large dependency sets in libraries. In many programming languages dependencies can arise among components because types refer to each other in their definitions. In *Aldor* and certain other languages, dependencies can also arise because types refer to each other in their *type*. We give a simple example. Suppose we have the following declarations:

```
define AbelianGroup: Category == with {
    +: (%, %) -> %;
    *: (Integer, %) -> %;
    ...
}

define DifferentialRing: Ring with {
    diff: % -> %;
}

Integer: Join(DifferentialRing, ...) == ...
```

That is, the domain `Integer` is declared to (trivially) belong to the category `DifferentialRing` so that it be possible to construct differential operators and other structures with integer coefficients. The problem is that the type `Integer` appears in the definition of `AbelianGroup`. Because of this, all domains that belong to `AbelianGroup` have an indirect and undesired dependency on `DifferentialRing`.

In compiling programs we may wish to verify that expressions have well-defined type, to verify that types are well-formed and to perform type inference. The dependencies that arise through the types of types can lead to large systems requiring fixed-point analysis. This not only imposes technical constraints on the type system, it also requires careful compiler design to avoid long compilation times for simple programs.

This form of dependency has been seen to be a practical problem. In the design of the Axiom system, basic mathematical types were endowed with all appropriate advanced algebraic interfaces. This led to an almost complete inter-dependency among the interface specifications. Doing a complete type checking of the library interfaces took several days, and this led to a reluctance to modify the library.

Complex conditionalization. While conditional category membership is one of the more useful features of the *Aldor* language and its predecessors, it is also subject to difficulties when new categories are used. We illustrate this with the domain-constructing function `DirectProduct(n, S)` which constructs the type of n-tuples of values from the type `S`.

```
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new:       Tuple S -> %;

    if S has Semigroup then Semigroup;
    if S has Monoid then Monoid;
    if S has Group then Group;
    ...
    if S has Ring  then Join(Ring, Module(S));
    if S has Field then Join(Ring, VectorField(S));
    ...
    if S has DifferentialRing then DifferentialRing;
    if S has Ordered then Ordered;
    ...
} == add {
    ...
}
```

Here we see that the set of categories satisfied by `DirectProduct(n, S)` depends very much on the categorical properties of the argument `S`. The direct product inherits from many, but not all, of the categories satisfied by `S`. For example, if `S` is a `Monoid`, then so is `DirectProduct(n,S)`. The same is true for many other categories. Sometimes `DirectProduct(n,S)` does *not* belong to the categories satisfied by `S`. For example, if `S` is a `Field` then `DirectProduct(n,S)` is not. Sometimes the opposite is true: sometimes `DirectProduct(n,S)` belongs to *additional* categories by virtue of the categorical properties its argument. This occurs, for example, when `S` is a `Ring`.

This example serves to make two points: First, we see that the categorical properties of a domain-constructing function can be quite complex and depend very much on the specific nature of the type constructor — it is not possible to describe this behaviour with a few simple universal rules. Second, we see that certain constructors are open-ended in their conditionalization requirements — whenever new categories are added to the environment, it is likely the constructor should be augmented.

4. Post Facto Extensions

Our solution to the problems we have outlined is to provide a mechanism for domain-valued expressions to have their meaning augmented with additional categories. This is achieved by allowing names bound to domains and domain-producing functions to have additional definitions and by providing rules by which the multiple meanings visible in a given scope are to be combined. We explicitly note that the representation of values belonging to the augmented domains does not change. All that is different is that the domain to which they belong is made to belong to additional categories, and consequently support more operations.

Extending domains. If D is a domain-valued constant, then its meaning may be extended with a definition of the form

`extend` D`:` C `==` E

This declares the D to belong to the category given by C in addition to whatever other categories it belongs in the current scope. In general, belonging to this new category may require D to provide new exports. The expression E gives the implementation of these new exports in terms of previously exported operations. The keyword `extend` is required so that the definition is not taken to be an independent, overloaded meaning.

To illustrate, the domain `Integer` may be made to belong to the category `DifferentialRing` by providing the following extension

```
extend Integer: DifferentialRing == add {
    diff(n: Integer): Integer == 0;
}
```

Separately, `Integer` may be made to belong to the category `ConvertibleTo(MathML)` by providing the extension

```
extend: Integer: ConvertibleTo(MathML) == add {
    convert(n: Integer): MathML == mi(n)
}
```

Named domains can in this way have different behaviours added as needed. When an existing domain is used with a new library, then a set of extensions can be provided to make the domain belong to whichever categories are desired. The programmer is free to organize the extensions in any suitable manner. In a scope where a domain-valued constant is used, its type is taken to be the `Join` of all the categories of the visible extension definitions and its value is taken to be the `add` of all the expressions from the extension definitions. That is, if the visible definitions are

```
N: C0 == A0;
extend N: C1 == A1;
...
extend N: Cn == An;
```

then the domain used will be formed as

```
N: Join(C0,C1,...,Cn) ==
A0 add A1 add ... An add {}
```

Extending functions. Domain-producing functions may be extended by providing an additional function definitions, marked with `extend`. An extension of a domain-producing function must have arguments with the same domains as the corresponding arguments of the original function. Normally, however, one or more of the arguments will have different subtype properties. (This includes the case where domain-valued arguments are declared to belong to different categories.) The declared return type of the extension function is taken to be an additional category to which the resulting domain will belong. To illustrate, we rewrite the `DirectProduct` example using extensions as shown in Figure 1. It would not normally be the case that the extensions would be given together as shown here. More often the extensions would either be placed together with the category definitions or be grouped in some way (*e.g.* extensions necessary to make domains of library 1 work with library 2).

In a scope where a domain-producing function constant is used, the original function value and all of the visible extensions are combined to produce the function that is actually used. This allows proper behaviour of function-valued names. So, for example, extended domain-producing functions may be passed as parameters, saved as values *etc*, and later used as desired.

If the visible function definition and extensions for F are

```
F(a1: T01,...,ak: T0k): R0 == A0
extend F(a1: T11,...,ak: T1k): R1 == A1
...
extend F(a1: Tn1,...,ak: Tnk): Rn == An
```

this is equivalent to the definition

```
F(a1:Meet(T01...Tn1),...,an:Meet(T0k...Tnk)): with {
    if a1 ∈ T01 and ... and ak ∈ T0k then R0;
    if a1 ∈ T11 and ... and ak ∈ T1k then R1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then Rn;
} == add {
    if a1 ∈ T01 and ... and ak ∈ T0k then A0;
    if a1 ∈ T11 and ... and ak ∈ T1k then A1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then An;
}
```

Here the symbol "$\in$" is interpreted to be a subtype test for the corresponding base domain. In particular, when $\mathtt{T}_{ij}$ is a category "$\in$" means "`has`." If $\mathtt{T}_{0i} = \mathtt{T}_{1i} = \cdots = \mathtt{T}_{ni}$ then the i-th test can be omitted.

These rules are applied recursively, with suitable interpretation of `Meet`, `Join` and `add`, so that curried domain-producing functions are handled naturally.

Implementation. In *Aldor* data values are not necessarily self-identifying, but each expression has a unique well-defined domain. Operations on data values are in principle extracted during execution from these domain objects and it is the compiler's responsibility to ensure that all the necessary domain objects are available at known locations at run-time. Post-facto extension of domains is implemented by constructing composite domain objects. Post-facto extension of functions is implemented by combining functions as described above. One of the most important aspects of the implementation of post facto extension is the static optimization of extension compositions, determining which functions should be called during execution. This enables a number of further optimizations, resulting in relatively efficient code.

Post facto extension can also be implemented in an object-oriented environment. In this case it is necessary to modify data structures representing class objects. These classes (including virtual function tables) are usually accessed through the member objects so there is the added complexity of matching the lifetime of the post facto extensions with their scope.

Relation to other work. Our design of post facto extensions makes use of the idea of mixins, from the Flavors system [10], applied to type-producing functions. This allows a separation of concerns in the creation and use of first class type objects, as described in [2]. The result gives a specialized form of what has come to be known as aspect-oriented programming [11], applicable to parameterized and non-parameterized types. If we view contant domains as nullary domain-producing fuctions, we may view post-facto extension as providing scoped point cuts associated to domain constructors. In the non-parametric case, a similar effect can be achieved with open classes [12]. The use of type categories in *Aldor* allows the compiler to perform various optimizations, as described in [1], to eliminate function look-up and perform in-lining where possible, taking into account post facto extensions.

5. Use in Library Design

We now have a dozen years' experience in the use of post facto extensions for structuring mathematical libraries for *Aldor*. This section describes some of the ways in which we have found it to be useful.

Uniform treatment of raw types and object types. Many programming languages make a distinction between "raw types" and "object types." This distinction does not exist in *Aldor*. The *language* defines a set of standard types and the *library* endows them with operations. All basic domains are initially defined simply as data representations. All primitives are given as independent operations, provided by the `Machine` package. For example, we have

```
Boolean:       Type == add {};
Integer:       Type == add {};
DoubleFloat:   Type == add {};
...
```

and the `Machine` package provides primitives for arithmetic on values of these types. Later, these types are extended by the standard library to have a richer structure.

Layering large libraries. We have found it useful to be able to build large libraries in layers, with fewer dependency cycles. In bootstrapping the Standard *Aldor* Library, we have the following layers:

1. *Basic types without operations.* The basic types are simply declared to be types, and the data representation is implicit in the use of available machine primitives on these types.
2. *Basic types with representation.* The basic types are extended to themselves export the relevant primitives. They may now be treated as types with an opaque representation. As the basic types are extended with operations, they may refer to each other in the signatures of their exports. For example, we may have

```
extend Boolean: with {
    =: (%, %) -> Booelean;
    convert: % -> String;
    ...
} == ...

extend Integer: with {
    =: (%, %) -> Boolean;
    <: (%, %) -> Boolean;
    convert: % -> String;
    ...
} == ...
```

```
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new:       Tuple S -> %;
} == add { ... }

extend DirectProduct(n: Integer, S: Semigroup): Semigroup == ...
extend DirectProduct(n: Integer, S: Monoid): Monoid == ...
extend DirectProduct(n: Integer, S: Group): Group == ...
...
extend DirectProduct(n: Integer, S: Ring): Join(Ring, Module(S)) == ...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
extend DirectProduct(n: Integer, S: DifferentialRing): DifferentialRing == ...
extend DirectProduct(n: Integer, S: Ordered): Ordered == ...
...
```

Figure 1. `DirectProduct` defined using extensions

```
extend String: with {
    =: (%, %) -> Boolean;
    #: % -> Integer -- Length.
    ...
} == ...
```

These may be compiled without having to resort to a multi-type fixed-point determination in type inference.

3. *Definition of constructed types.* The library uses the primitive types to construct a richer set of useful types, such as linked lists, hash tables, I/O abstractions, *etc*.
4. *Types with useful categories.* The Standard Library defines a number of categories, and the basic and constructed types are extended to belong to them as appropriate.

The Algebra Library is built on top of the Standard Library as follows:

5. *Mathematical categories.* The Algebra Library defines a rich categorical structure with categories corresponding to many of the standard algebraic abstractions. These include abstractions for the concept of group, ring, euclidean domain, field, module, algebra, *etc.*
6. *Extension of the basic types.* The arithmetic types of the Standard Library are extended to belong to all the appropriate categories from the Algebra Library.
7. *Definition of mathematical domains.* The library defines a set of common mathematical domains, such as polynomials, matrices, quotients, finite fields and so on.

A number of more sophisticated mathematical libraries are built on top of the Algebra Library, and these extend the types of the Standard Library and Algebra Library, as appropriate.

This layering allows the elimination of cyclic dependencies in the design of the libraries and allows the libraries to be built and tested in a modular fashion. It does this without compromising the rich set of behaviours desired for the basic types.

Combined use of multiple libraries. With post facto extensions it is quite easy to use multiple, independently developed libraries without a host of data conversions. The application programmer decides which categories from the various libraries will be important and extends the necessary types to export them. Having done this, the values computed by one library may be readily used in the other libraries without conversion.

Separation of concerns. With post facto extensions it is straightforward to separately implement various independent aspects of domains. This is one of the standard goals of aspect-oriented programming. For example, one set of extensions can provide algebraic algorithms, while another set of extensions provide translations to TEX and a third set of extensions provide translations to MathML. The code for each set of extensions can be separately developed, tested and maintained.

Adding callback algorithms to parameters. One of the difficulties with generic programming is that there are often specialized algorithms that apply over certain domains. In C++ this is handled by template specialization and is resolved statically. However, in *Aldor* types may be constructed dynamically so we need some other mechanism to access specialized algorithms. Post facto extension, combined with conditional category tests, allows generic code to use special purpose algorithms, when applicable, without revising library components.

We illustrate this point with an example from linear algebra. Such a package can be defined generically over any commutative ring. More efficient algorithms may be used, however, when the ring is known to be an integral domain or a field. We may thus assemble these algorithms into a package as follows:

```
LinearAlgebra(R:CommutativeRing, M:MatrixCategory R):
with {...} == add {
    local Elim: LinearEliminationCategory(R, M) == {
        R has Field =>
            OrdinaryGaussElimination(R, M);
        R has IntegralDomain =>
            TwoStepFractionFreeGaussElimination(R,M);
        DivisionFreeGaussElimination(R, M);
    }

    determinant(m:M):R == determinant(m)$Elim;
}
```

Certain coefficient rings may support efficient specialized algorithms. For example, we may want to compute over the integers using Chinese remaindering. However, we do not want to have to modify the `LinearAlgebra` package whenever a new method is incorporated into the library. We therefore define a category that a ring can implement to provide linear algebra algorithms over itself:

```
LinearAlgebraRing: Category == with {
    determinant: (M:MatrixCategory %) -> M -> %;
    rank:        (M:MatrixCategory %) -> M -> Integer;
    ...
}
```

We make one modification to the `LinearAlgebra` package to take advantage of special-case algorithms carried in a `LinearAlgebraRing` view: we replace the `determinant` function with

```
determinant(m:M):R == {
    if R has LinearAlgebraRing then
        determinant(M)(a)$R;
    else
        determinant(m)$Elim;
}
```

When we have special algorithms for some domain, we extend the domain to know about them:

```
extend Integer: LinearAlgebraRing == add {
    determinant(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingDeterminant(M, m);
    rank(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingRank(M, m);
    ...
}
```

Whenever we use the `LinearAlgebra` package, it will use the designated algorithm even if the coefficient ring is determined dynamically.

The technique of using post facto extensions to endow domains with special- case algorithms has been used in the in the construction of the $\sum^{IT}$ library [13]. The notion of rings knowing how to perform operations in structures over themselves has been explored earlier in relation to composing factorization algorithms [14].

6. Conclusions

We have examined a number of problems that arise in the construction of software libraries. These are all related to question of whether existing types should be updated with new abstractions as libraries grow or are used together. We have noted that the problem is particularly accute in computer algebra, where it is quite usual that pre-existing types can satisfy newly defined abstractions. We have observed, however, that in mathematical programming new abstractions do not usually require any change in data representation in order to be applied. Based on this observation, we have proposed the notion of *post facto extension* of types and of type producing functions. This provides a specialized instance of aspect-oriented programming that has proven particularly effective for mathematical computing.

Acknowledgments

The author would like to thank Samuel S. Dooley for his assistance with the first implementation of post facto extensions in the $A^\sharp$ compiler at IBM Research. He would also like to express his gratitude to the late Manuel Bronstein for the `LinearAlgebraRing` example.

References

[1] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S.: *A first report on the $A^\#$ compiler*, Proc. 1994 International Symposium on Symbolic and Algebraic Computation, pp. 25–31, ACM Press.

[2] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S.: *AXIOM Library Compiler User Guide*, The Numerical Algorithms Group Ltd, Oxford 1994.

[3] Watt, S.M., *Aldor*, in *Handbook of Computer Algebra*, Grabmeier, Kaltofen and Weispfenning (editors), Springer Verlag 2003, pp. 265–270.

[4] *Aldor User Guide*, `http://www.aldor.org` (2003).

[5] Jenks, R.D., Sutor, R.S.: *Axiom: The Scientific Computation System*, Springer Verlag (1992).

[6] Liskov, B. A history of CLU, Proc. Second ACM SIGPLAN conference on History of programming languages, pp. 133–147, ACM Press (1993).

[7] Jenks, R., Trager, B.: *A language for computational algebra*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, pp. 6–13, ACM Press.

[8] Foderaro, J.K.: *The Design of a Language for Algebraic Computation*, Ph.D. Thesis, UC Berkeley, 1983.

[9] Abdali, S.K., Cherry, G.W., Soiffer, N.: *A Smalltalk system for algebraic manipulation*, Proc. 1986 Object Oriented Programming Systems Languages and Applications, pp. 277–283, ACM Press.

[10] Moon, D.A.: *Object-oriented programming with flavors*, Proc. 1986 Object Oriented Programming Systems Languages and Applications, pp. 1–8, ACM Press.

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J.: *Aspect-Oriented Programming*, Proc. European Conference on Object-Oriented Programming, LNCS 1241 pp. 220-242, Springer Verlag (1997).

[12] Millstein, T., Chambers, C.: *Modular Statically Typed Multimethods*, Proc. European Conference on Object-Oriented Programming, LNCS 1628 pp. 279-303, Springer Verlag (1999).

[13] Bronstein, M.: *$\sum^{IT}$: A strongly-typed embeddable computer algebra library*, Proc. DISCO'96, LNCS 1128 pp. 22-33, Springer Verlag.

[14] Davenport, J., Gianni, P., Trager, B.: *Scratchpad's view of algebra II: a categorical view of factorization*, Proc. 1991 International Symposium on Symbolic and Algebraic Computation, pp. 32–38, ACM Press.

Towards a Domain-Specific Aspect Language for Virtual Machines

– Position Paper –

Yvonne Coady Celina Gibbs
University of Victoria, Canada
ycoady@cs.uvic.ca, celinag@uvic.ca

Michael Haupt
Darmstadt University of Technology / Hasso Plattner Institute for Software Systems Engineering, Germany
michael.haupt@hpi.uni-potsdam.de

Jan Vitek Hiroshi Yamauchi
Purdue University, USA
{jv,yamauchi}@cs.purdue.edu

Abstract

High-level language virtual machines, e. g., for the Java programming language, offer a unique and challenging domain for aspects. This position paper motivates the need for an aspect-oriented language designed precisely for this domain. We start by overviewing examples of some of the crosscutting concerns we have refactored as aspects in VMs, and then demonstrate how mainstream aspect-oriented programming languages need to be augmented in order to elegantly implement these and similar concerns. We believe current join point and advice models are not expressive enough for this domain. Predominantly this is due to the fact that the concept of a point in the execution of the VM requires the ability to explicitly specify subtle issues regarding system state and services. Finally, the paper outlines, based on a design view on virtual machines, the shape of a possible domain-specific aspect language for the implementation of such systems.

1. Introduction

A virtual machine—virtual machines in the context of this paper are always *high-level language virtual machines* [12] such as the Java virtual machine—can be seen as providing several services to the application it runs, such as memory management, execution (interpreted or JIT-compiled), adaptive optimisation, thread management, synchronisation, and so forth. These services often interact closely, and these interactions' work flows are non-trivial.

When adopting a view that regards each such service as being a *concern*, the crosscutting nature of the services and their interactions becomes perceivable. Further considering that future VMs could customise services on the basis of application-specific behaviour [18], it becomes clear that virtual machines call for employing aspect-oriented programming in their implementations.

As an example, take the cooperation of the *execution*, *organiser* and *controller* services in the Jikes RVM's [2, 3, 13] adaptive optimisation system [4]. The *execution* service is that part of the VM actually running an application; albeit it is not explicitly modelled as a dedicated service in the Jikes RVM, viewing it as such supports our notion of VMs as collections of services. *Organiser* services are responsible for collecting performance data and issuing optimisation suggestions. The *controller* gathers such suggestions and decides on whether they should be put into action.

In this setting, an organiser observing call edge hotness may be signalled by the execution service that a particular call edge has been executed so-and-so many times, and the organiser may issue an optimisation request when that call edge exceeds a certain threshold. Though optimisation based on edge hotness is straightforward to describe, its current implementation requires using several threads and queues for their communication. The logic cuts across the virtual machine and is expressed only in an implicit way, by means of attaching queues to threads appropriately.

Several attempts have been made to actually utilise AOP in implementing virtual machine services. We focus on two particular examples in this work within Java-in-Java virtual machines. The first example is *GCSpy*, a heap visualiser [16], which has been introduced to the Jikes RVM using AspectJ [14, 5]. The second example is an implementation of *software transactional memory* (STM) in the OpenVM [15].

Observing the examples' utilisations of AOP constructs, it is interesting to see that, in both cases, the implementors had to go to considerable length to realise their particular VM services as crosscutting concerns. The observation has led to the recognition of some shortcomings of existing AOP languages that in turn call for dedicated modelling mechanisms for crosscutting in the virtual machine implementation domain.

The structure of this position paper is as follows. The next Section will briefly describe the aspect-oriented realisation of GCSpy in the Jikes RVM and of STM in the OpenVM, respectively. Section 3 will describe the identified shortcomings. Section 4 will present an initial proposal for some characteristics of a domain-specific aspect language for virtual machine implementations. Not all of the proposed language mechanisms are necessarily *specific* to the virtual machine implementation domain: some merely introduce a higher level of abstraction over generally applicable AOP language mechanisms, albeit in a way allowing for the declarative expression of domain-specific requirements. Section 5 summarises the paper and outlines future work directions.

2. Case Study: GCSpy and STM

Our experience with aspects in the VM domain include two research systems developed in Java, the Jikes RVM and the OpenVM. In both cases, we used standard AOP mechanisms provided by AspectJ, as described in the high-level overview in the subsections that follow.

DSAL '06 October 23, 2006, Portland, Oregon, USA.

2.1 The GCSpy Aspect

GCspy is a heap visualisation framework designed to visualise a wide variety of memory management systems. A system as complex as a VM benefits greatly from non-invasive, pluggable tools that provide system visualisation while minimising the effects on that system. Visualisation tools such as GCSpy inherently have many fine-grained interaction points that span the system they are visualising, lending themselves to an aspect-oriented implementation.

GCSpy-specific code interacts with the base RVM in order to establish two things: (1) to gather data before and after garbage collection, and (2) to connect a GCSpy server and client-GUI for heap visualisation. Details associated with the GCSpy aspect are overviewed in the table below. The code touches 12 classes, has a 1:1 ratio of pointcuts to advice, and uses a collection that spans before/after/around advice and can be further characterised as a 'heterogenous' concern [8].

GCSpy in RVM	
Classes Involved	12
Pointcuts	15
Before Advice	5
After Advice	6
Around Advice with Proceed	2
Around Advice without Proceed	2
LOC in Aspect	126

2.2 The STM Aspect

Historically, concurrent access to shared data has been controlled using mutual-exclusion locks and condition variables where critical sections are identified with a language-specific demarcation. Software transactional memory (STM) provides a declarative style of concurrency control, allowing programmers to work with the high level abstraction of software-based transactions. The abstraction replaces critical sections with transactions, which can be in one of several possible states and can be manipulated with a set of well-defined operations.

Modifying the OpenVM to support STM involved a series of changes that lent themselves to an aspect-oriented implementation. The details of this AspectJ implementation are overviewed in the table below. The code touches 13 classes, has a 1:1 ratio of pointcuts to advice, and employs the heavy use of non-proceeding around advice.

STM in OpenVM	
Classes Involved	13
Pointcuts	22
Before Advice	1
After Advice	3
Around Advice with Proceed	4
Around Advice without Proceed	14
LOC in Aspect	114

3. AOP in VMs: Current Shortcomings

Modularity within the implementation of a VM is generally challenging [6]. Implementations tend to rely on the "black art" of fine-grained optimisation within a predominantly monolithic system. But more recent implementations of virtual machines have shown that modularity and performance can indeed co-exist, and bode well for a future were JVMs can be more easily customised according to application-specific needs [18].

In previous work, we have demonstrated how the Jikes RVM's modularity can be enhanced even with a naïve implementation of aspects, and how these aspects impact system evolution [11]. Here, we consider a more qualitative assessment of the representations of the aspects themselves, and the ways in which AspectJ could be augmented to better support the needs of crosscutting concerns in this domain. In our experience, we have determined the need to explicitly define principled points in the execution of the VM in terms of a combination of current system state and a composition of system services. The following subsections consider the ways in which domain-specific needs outstrip current join point, pointcut and advice models for AspectJ [14].

3.1 Join Points in VMs

The AspectJ joint point model was designed according to principled points in the execution of a program, such that join points remain stable under a stable interface. Similarly, a characteristic required of domain-specific join points is stability across inconsequential changes as well as being understandable to a typical VM programmer. It is our experience that the types of join points—both static and dynamic—exposed by most existing join point models are not sufficient for expressing the special needs of crosscutting concern composition in virtual machine implementations. We believe a domain-specific aspect language for VMs could adhere to the stability characteristic offered by traditional join points, while augmenting the model with further support for meaningful and much needed service composition. We believe one of the interesting challenges in this work is that, in the domain of VMs, this requires simultaneous attention to both higher-level abstractions and lower-level details.

For example, the current AspectJ-based implementation of GCSpy and STM services both require a largely 1:1 pointcut to advice ratio as shown in Sec. 2. There is little redundancy of advice as they apply to specific points in the execution of the system. As a result, the aspects are relatively large and arguably difficult to understand from a high-level perspective. Though they improve the ability for developers to reason about their internal structure and external interaction, the improvement is arguably less compelling than a higher-level representation may be able to achieve. To get a sense of what the AspectJ-based implementation looks like in terms of implementation, four fine-grained pointcut/advice pairs are required just to ensure GCSpy starts properly when the VM boots. A similar phenomenon exists in the STM aspect. An abstracted view of the GCSpy code follows.

```
before(): execution(* Plan.boot()) {
  Plan.objectMap = new ObjectMap();
  Plan.objectMap.boot();
}

before(VM_Address ref):
 args(...) && execution(* Plan.postCopy(...)) {
    Plan.objectMap.alloc(...);
}

before(VM_Address original):
 args(...) && execution(* Plan.allocCopy(...)) {
    Plan.objectMap.dealloc(...);
}

void around(VM_Address ref,..) :
 args(...) && execution(* Plan.postAlloc(...)) {
    if (allocator == Plan.DEFAULT_SPACE
        && bytes <= Plan.LOS_SIZE_THRESHOLD) {
      Plan.objectMap.alloc(...);
    else
      proceed(...);
}
```

We consider the possibility to define such a concern as a higher-level abstraction, at the granularity of a service. We envision this to include startup parameters that would specify information such as whether it is started in a separate thread, whether the application triggers it, or other threads under certain circumstances trigger it. This high-level abstraction shields the VM programmer from knowledge of the fine-grained points at which the service interacts with other services, shifting the complexity to the domain-specific aspect language processor. In the case of the startup of GCSpy, the GCSpy service interacts with the VM boot service. An example of GCSpy's interaction with the VM boot service is further illustrated in Sec. 4.

In terms of requirements along the lines of a finer granularity than currently allowed by AspectJ's pointcut model, another problem exhibited in both the GCSpy and STM aspects is the aggressive refactoring of the VM code they crosscut in order to expose appropriate join points. In [17], Siadat *et al.* provide results that suggest an intolerable amount of refactoring to expose sufficient join-points in systems code. Refactorings resulting from naïve aspect-oriented implementations yielded either (a) empty methods, or (b) new methods that do not necessarily enhance the system. They even break modularity by introducing methods for which no abstraction is required. We would prefer ways to accomplish more explicit fine-grained inter-service relationships within VMs. Our experience in this domain has lead us to conclude that current join point models are not fine-grained enough to be highly effective within VMs.

3.2 Pointcut Descriptors

Pointcut descriptors determine whether a given join point matches a point in the execution of the VM. In our experience, virtual machine services may expose some points where other services may interact and often inherently require access to dynamic, often shared, VM system state. For example, the execution concern may expose a point indicating that a certain call edge hotness has exceeded a given threshold, which might be interesting for the adaptive optimisation concern. Similarly, the generalisation of memory management systems monitored by GCSpy, or controlled by STM, also lend themselves to this scenario, where one service is interested in points in the execution of the composed system only if system state is appropriate.

The circumstances in which an optimisation request is to be generated are cumbersome to express using the pointcut language present in AspectJ. The pointcut must match *only* if the call edge hotness actually *exceeds* the threshold; as long as a call edge's invocation count is less than the threshold, the optimisation aspect is in the "do not optimise" state, which it leaves in the moment the count exceeds the threshold. As soon as another, greater threshold is exceeded, the aspect may choose an even higher level of optimisation for the call edge. This basically constitutes a stateful aspect [10] and could be expressed using the appropriate means, e. g., tracematches [1].

We would like to stress that this kind of pointcut is not specific to the domain of virtual machine implementations and hence does not actually constitute a need for a *domain-specific* language construct. It is rather the case that stateful aspects of this kind characteristically occur in the domain. Still, a declarative way of expressing them, other than a generic one as seen in the tracematch syntax, may be more viable by means of introducing greater abstraction.

3.3 Advice

Advice supply a means of specifying code to run at a join point. It is important for a domain-specific language to carefully consider the nature of the VM domain. It is unacceptable, within this domain, to introduce possibly prohibitive performance penalties, or dramatic increases in the system's memory footprint. Returning to the adaptive optimiser example mentioned above, if the execution service signals sufficient call edge hotness, the optimising service actually should not necessarily kick in immediately. In this case, it could harm the VM's performance to optimise every single call edge as the execution service sees it fit for being optimised. The VM should instead wait until there are ample time and resources available. Usually, this is implemented using separate threads and a queue storing requests (as in the Jikes RVM; cf. above). The two concerns interact in a detached way; they utilise *asynchronous* advice [7], as met in, e. g., the AWED language [9].

Asynchronous advice cannot be expressed directly using simple AspectJ mechanisms, as the required queues and associated state have to be introduced as explicit data structures. Although there is no such concept as an asynchronous advice in traditional AOP advice models, we believe this to be highly desirable in VMs. Given that there are very likely many VM services that do not interact synchronously, modeling such services as crosscutting concerns calls for providing a mechanism allowing for such definitions.

4. A DSAL for Virtual Machines

Based on our experience with aspects in VMs so far, we believe a domain-specific aspect language for virtual machines must address the following issues:

1. Many common services in a VM can be structured as crosscutting concerns. An according DSAL should provide a high-level view on VM abstractions and their corresponding implementation that allows for expressing virtual machine services as modules of their own, explicitly specifying their characteristics and relations to other services. The abstractions should be generalisable across multiple VMs, enabling the services to be generalisable as well.
2. Existing join point models are not sufficient to express the rationale and type of interaction between the concerns found in a VM. A DSAL for VMs should allow for exposing types of join points met in virtual machines, for expressing them in appropriate pointcuts, and for specifying advice that should run at those points in meaningful ways (synchronously/asynchronously).

Regarding the first issue, a VM service should be expressible as a single, configurable module in terms relative to core VM abstractions. For a simple example, a completely asynchronous service (running in a dedicated thread) could be succinctly expressed like this:

```
detached service AdaptiveOptimiser { ... }
```

or advice associated with a boot-time sequence might be expressed like this:

```
service GCSpy {
  during(VMbooting): { ... }
  ...
}
```

with the DSAL weaver knowing about the places to join to in terms of boot-time logic, as signified by `during`. Of course, there may be cases where it is necessary to specify the order of service startups, in case of possible conflicts.

Each service should also make clear which points it exposes to others, establishing a clear interface for crosscutting behaviour. For example, in the case of the STM aspect, it may be possible to apply either optimistic or pessimistic concurrency control strategies depending on the level of conflict in the system. The points at which these different concurrency control services could be applied should be exposed by the STM service.

As for adaptive optimisation, the circumstance that a method `m2()` should be inlined in `m1()` when it has been called therefrom

more than 100 times could be expressed like this, as part of an organiser service:

```
detached service CallEdgeOrganiser {
  whenever(VM_Method m1, VM_Method m2):
    edgehotness(m1,m2) exceeds 100 {
      inline(m2, m1);
    }
}
```

In this example, the `whenever` advice type means asynchronous advice execution, and the `exceeds` comparison operator implies that the `edgehotness` value must exceed the given threshold for the pointcut to match. The `edgehotness` value, by the way, is exposed from the execution service.

5. Summary

Our experience with aspects in VMs leads us to believe that this domain could benefit greatly from VM-specific AOP mechanisms. In this paper, we have argued this point based on our sample aspects in the RVM, the OpenVM, and additionally reasoning about a common optimisation scenario. We also have described the ways we believe the join point model in AspectJ could be augmented to suit this domain. Based on these observations, we have proposed to regard crosscutting concerns in virtual machines as a special domain of aspects, requiring support in the form of dedicated language expressions.

Future work in this area will focus on a close examination of crosscutting in high-level language virtual machines. Based on results from this analysis, a more complete critique of existing AOP models will be formulated, along with a more detailed version of the proposed domain-specific aspect language.

References

[1] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proc. OOPSLA 2005*, pages 345–364. ACM Press, 2005.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño—a compiler-supported java virtual machine for servers. ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99), May 1999.

[3] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA 2000 Proceedings*, pages 47–65. ACM Press, 2000.

[5] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.

[7] M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In *Proc. GPCE 2003*, volume 2830, pages 169–188. Springer, 2003.

[8] A. Colyer and A. Clement. Dlarge-scale aosd for middleware. In *AOSD '04: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[9] R. Douence, D. Le Botlan, J. Noye, and M. Sudholt. Concurrent aspects. In *Proc of GPCE*. Springer, 2006.

[10] R. Douence, P. Fradet, and M. Sudholt. A framework for the detection and resolution of aspect interactions. In *Proc of GPCE*, pages 173–188. Springer, 2002.

[11] C. Gibbs, R. Liu, and Y. Coady. Sustainable system infrastructure and big bang evolution: Can aspects keep pace? In *Proc. ECOOP 2005*. Springer, 2005.

[12] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan-Kaufmann, 2005.

[13] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[15] OpenVM Home Page. `http://ovmj.org/`.

[16] T. Printezis and R. Jones. Gcspy: an adaptable heap visualisation framework. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343–358. ACM Press, 2002.

[17] J. Siadat, R. Walker, and C. Kiddle. Optimization aspects in network simulation. In *AOSD '06: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 122–133. ACM Press, 2006.

[18] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 49–60. ACM Press, 2004.

Partial Evaluation + Reflection = Domain Specific Aspect Languages

DeLesley Hutchins
LFCS, University of Edinburgh
D.S.Hutchins@sms.ed.ac.uk

Abstract

Domain-specific languages (DSLs) are typically implemented by code generation, in which domain-specific constructs are translated to a general-purpose "host" language. Aspect-oriented languages go one step further. An aspect weaver doesn't just generate code, it transforms code in the host language. In both cases, one of the major challenges in building and using the DSL is achieving good integration between the code generator and the host language. Generated code should be type safe, and any errors should be reported before generation.

Partial evaluation and multi-stage languages are excellent tools for implementing ordinary DSLs which satisfy these requirements. Combining partial evaluation with reflection could potentially yield a system which is strong enough to perform aspect weaving as well. This paper discusses some of the technical hurdles which must be overcome to make such a combination work in practice.

1. Introduction

A domain-specific language (DSL) differs from an ordinary library because the functionality provided by a DSL cannot be easily encapsulated behind ordinary functions and classes. Efficiency is the usual culprit behind this failure of encapsulation. A clean interface may introduce a layer of *interpretive overhead* which is unacceptable. To reduce such overhead, DSLs are often implemented by means of code generation, in which domain-specific constructs are translated or compiled to a general-purpose "host" language.

Like DSLs, aspect-oriented programming (AOP) addresses a failure of encapsulation. In the case of AOP, encapsulation fails because *cross-cutting concerns*, which are logically separate in the high-level design of a program, become tangled together in the source code. An aspect language allows such concerns to be specified separately, and then *weaves* the aspects together to generate a complete program [12].

Whereas DSLs are primarily concerned with *generating* code, aspect languages are primarily concerned with *transforming* code. These two tasks are qualitatively different. A code generator does not need to understand the full syntax and semantics of the host language. Many successful code generators are little more than glorified macro systems — they manipulate blocks of code as untyped syntax trees, or even ASCII text.

DSAL Workshop, GPCE 2006 Oct 23d 2006, Portland OR

A code transformer, on the other hand, must parse and understand the code that it transforms. For example, in order to perform aspect weaving, the AspectJ compiler must correctly distinguish between field and method signatures, and keep track of the inheritance relationships between classes.

Nevertheless, the distinction between code generation and code transformation is not black and white. Indeed, a domain-specific aspect language (DSAL) can be usefully defined as a DSL which performs both code generation and weaving. It may generate code for domain-specific constructs, and then weave that code into an existing general purpose program. RIDL and COOL follow this pattern [15].

Most programmers would agree that writing good code is difficult. Experience with DSLs has shown that writing good code generators is even more difficult. As a result, a great deal of research has focused on developing toolkits which simplify the task of writing generators. This paper discusses some of the issues involved in building a toolkit for constructing DSALs, which must perform both code generation and weaving. One of the main challenges for such a toolkit is achieving close integration between the DSAL and the host language.

With any DSL, generated code should be correct and type-safe, and any errors should be reported *before* code generation, so that the user does not need to read generated code. In a DSAL, good integration is even more important, since the weaving process must directly modify constructs in the host language. Ideally, the weaver should respect the semantics of the host language, so that weaving does not create unexpected changes in behavior.

Partial evaluation and multi-stage languages are excellent tools for writing DSLs which are well-integrated with their host language [11] [19]. Combinining partial evaluation with reflection could potentially yield a system which is strong enough to perform aspect weaving as well. However, there are a number of technical hurdles to overcome before this mechanism can be applied in the real world.

2. Language Integration in DSLs

Parser generators such as `yacc` and `antlr` are an old and familiar example of DSLs. A parser generator takes a grammar definition as input, and generates a program which parses the grammar. The following is an ANTLR grammar rule:

```
expr returns [int v]          { int e1, e2;  }
   :  v=literal
   |  e1=literal "+" e2=expr  { v = e1 + e2; }
   ;
```

The important thing to notice about this example is that it contains a mixure of domain-specific code, and code in the host language (here Java or C++). ANTLR does not attempt to parse or un-

derstand code in the host language, it simply copies the raw ASCII text. Any syntax errors or type errors will not be discovered until after the code has been generated. Manipulating source code as ASCII text can also introduce subtle scoping and naming errors, as was frequently encountered in early, "unhygienic" macro systems. This is an example of poor integration with the host language.

On the other hand, ANTLR does perform domain-specific analysis of the code that it generates. Because code generation is done ahead of time, it can detect ambiguous grammars, missing rules, and similar problems.

Parser combinators represent an alternative approach [14]. The following is code written in Haskell. (Unlike ANTLR, there is no syntax sugar, so it is somewhat harder to read).

```
add e1 _ e2 = e1 + e2

expr :: Parser Char Int
expr =  literal
    <|> (pSucceed add) <*>
        literal <*> (pSym '+') <*> expr
```

With combinators, each grammar rule is actually given a type in the host language; `Parser Char Int` is the type of an object which parses a character string to return an integer. Small parsers are combined using the `<|>` and `<*>` operators to create larger parsers. These operators are statically typed, so any type errors will be discovered before composition, rather than after. Instead of representing host-language expressions as ASCII text, Haskell uses higher-order functions.

Haskell is a particularly good language for writing DSLs, because it is possible to manipulate pieces of code as first-class values, much like Lisp and Scheme. It is easy to wrap any expression up as a function, and then pass it as an argument to the code generator. Unlike Lisp and Scheme, Haskell is statically-typed. Static typing confers an important benefit: if a DSL code generator is well-typed, *then the code that it generates is also guaranteed to be well-typed*.

As a result of these features, parser combinators are well-integrated with Haskell. Unfortunately, they also have a disadvantage. The parser is not generated until run-time, so there is a layer of interpretive overhead. Type errors are detected at compile-time, but domain-specific errors, such as ambiguous grammars, will not be detected until run-time.

2.1 Partial Evaluation and multi-stage languages

Partial evaluation is an old technique which attempts to overcome these problems. A partial evaluator fuses a compiler with an interpreter. Offline partial evaluation, which is the kind commonly used in practice, works by labeling every expression in a program as either "static" or "dynamic". This process is known as *binding-time analysis*. Static expressions are evaluated at compile-time by the interpreter, while dynamic expressions are compiled to machine code, which will be evaluated at run-time [11].

Multi-stage languages operate on a similar principle [19]. A multi-stage language allows a block of code to be "quoted", which means that the evaluation of the code is delayed. A dual "unquoting" mechanism forces the immediate evaluation of particular subexpressions. The net effect is similar to partial evaluation – expressions are labeled as either "evaluate now" (static) or "evaluate later" (dynamic).

These two mechanisms are designed to eliminate the interpretive overhead associated with a DSL. In the case of parsers, the production rules for a particular grammar are statically defined. A partial evaluator would thus evaluate the `<*>` and `<|>` operators at compile-time. Any composition errors (such as those caused by an ambiguous grammar) would also be detected at compile-time.

2.1.1 A Toolkit for building DSLs?

Put together, the tools described above offer some hope of a universal toolkit for building DSLs which are *well-integrated* with their host language, and which *respect the semantics* of that language. The key ingredients are the following:

- First-class functions, which can be used to manipulate code as values.
- Static typing, which guarantees that the generated code will be type-safe.
- Partial evaluation or staging.

(Partial evaluation automatically respects the semantics of the host language, because the "semantics" of a language is just a description of the evaluation rules for that language.)

2.1.2 Limitations and partially static data

Unfortunately, partial evaluation does have some limitations in practice, which have been described extensively in the literature [10]. The amount of speedup, and the nature of the generated code, depend closely on two things: the way in which the DSL interpreter was written, and the precise algorithm used for binding-time analysis. Aggressive analyzers can locate more static data, but the evaluator may then fail to terminate.

Even when binding-time analysis works properly, a partial evaluator will only rewrite program terms according to the reduction rules of the host language. There are a number of other program transformations (such as deforestation [21]) which are both semantics-preserving, and which yield substantial speedups, but these are beyond the reach of partial evaluators.

One of the most serious problems is that partial evaluators must label data structures as either "static" or "dynamic". Interpreters for real-world DSLs often manipulate data structures that are "partially static", containing a mixture of static and dynamic information. A simple evaluator is forced to label such structures as "dynamic", which means that they will be unable to remove much of the interpretive overhead.

Complex data structures must be factored into static and dynamic parts. This can be done either by rewriting the interpreter, using more sophisticated binding-time analysis, or both. "Tag removal" in strongly typed languages is a special case of the problem, and one which is particularly difficult to solve [20].

3. Aspect Weaving

In the DSL examples above, "good integration" with the host language means that expressions in the host language can be wrapped up and passed to the code generator in a type-safe manner. Aspect weavers go far beyond this, because they must parse, understand, and modify constructs in the host language.

Reflection, which is found in many OO languages, including Smalltalk, CLOS, and (to some extent) Java, allows ordinary code to inspect and/or modify existing classes. Class declarations are known at compile-time, and so they constitute static data. Combining reflection with partial evaluation extends the range of generators which can be produced.

3.1 Introspection

The reflection facilities provided by Java are *introspective*. It is possible to inspect, but not modify, the structure of a program. In particular, Java supports the following operations:

- It is possible to find the class of an object at run-time.
- It is possible to query the class to find method names and signatures.

- It is possible to call a method whose name and signature is not known until run-time.

This kind of reflection is useful for generating "boilerplate" code, which must peform the same task in the same way on a wide variety of data types. In functional programming circles, boilerplate generation is called "generic programming" [13] [7]. Examples of "boilerplate" include:

- Comparing two objects for equality. (Compare the values of all fields.)
- Serializing an object, or converting it to string. (Serialize all fields.)
- Generic traversals and rewrites of complex data, such as XML.

Programmers tend to avoid reflection where possible because its performance is abysmal. However, if the class of an object is statically known, then it is possible to partially evaluate reflective calls. Partial evaluation eliminates the method lookup code, and transform reflective method calls into ordinary method calls [4].

Consider the following example, which is taken from [4]:

```
static void printFields(Object anObj) {
  Field[] fields = anObj.getClass().getFields();
  for (int i = 0; i < fields.length; i++)
    System.out.println(fields[i].getName() +
      ": " + fields[i].get(anObj);
}
```

If the class of anObj is statically known, then getFields () can be evaluated at compile-time. Once fields is known, the loop will be further unrolled, producing a piece of code with no reflective calls.

Note that the class of anObj may be known even if its value is not, in the following two situations. First, if the type of anObj at partial-evaluation time is C, where C is a final class, then the run-time class of anObj is guaranteed to be C. Second, the run-time class of anObj is C if the value of anObj is the dynamic expression new C(...).

Exploiting this information requires a very sophisticated partial evaluator. In addition to labeling expressions as static or dynamic, it must label their *type* as either static or dynamic. Partial evaluation must be integrated with the type system.

3.2 Extensible classes

Smalltalk and CLOS not only allow classes to be inspected, they allow existing classes to be modified, or new classes to be created on the fly [3] [18]. In a statically typed language like Java, creating a new class would be a reflective operation, because classes are not ordinary objects. Since Smalltalk and CLOS are dynamic languages, creating new classes on the fly is standard practice.

By itself, introspection is limited because it can only be used to generate code inside methods. However, if classes are objects, then code generators can create the methods and classes themselves as well.

In principle, this mechanism is powerful enough to do the kind of aspect weaving found in AspectJ. An aspect weaver would first inspect the set of currently defined classes, and then modify those definitions as appropriate. Unfortunately, there are some serious technical problems that need to be overcome in order to make weaving work with a partial evaluator.

3.2.1 Classes Should be Immutable

Aspect-weaving is ordinarily thought of as an operation which modifies existing classes. Unfortunately, modification is a side-effect which changes the heap. Dealing with a mutable heap makes things much more difficult, because everything stored on the heap becomes partially static data — the Achilles Heel of partial evaluators.

Fortunately, this problem is easily solved. The solution is to treat an aspect as a function which takes an *immutable* set of classes as input, and yields a new set of transformed classes as output. This is the strategy used by feature-oriented programming, and there are several additional reasons to prefer it.

Lopez-Herrejon has argued that treating aspects as functions makes it easy to control the order of multiple transformations, and enables *step-wise refinement* [16].

Perhaps even more importantly, different parts of a program may need to use different sets of class extensions, which mean that the original definitions must be preserved. Bergel's class boxes [2], which are an extension to Smalltalk, allows classes to be extended only within a particular scope.

3.2.2 Type Safety

In a statically-typed language, classes denote types, so any transformation which affects class signatures (such as AspectJ's inter-type declarations) will affect the well-typedness of program code. Aspect weaving must thus be done before type checking, because it will invalidate any type judgements that were previously made. Unfortunately, doing aspect weaving first creates two problems. First, the weaver must manipulate code that may not be correct, which is especially problematic if it is combined with a partial evaluator that relies on correct type information. Second, type errors will appear in the generated code, where they are more difficult to fix.

One solution to this problem is to use a type system based on *virtual classes*, as found in the gbeta language [5]. Type judgements in gbeta are made under the assumption that the full definition of a virtual class is not statically known. Virtual classes can thus be extended without invalidating previous type judgements.

The use of virtual classes places a strong restriction on the aspect-weaver: the weaver cannot perform arbitrary transformations; it must only generate subclasses. The question of how to enforce this restriction in a fully reflective environment is an open problem.

Even with a restricted weaver, providing extensible classes within a type-safe language requires a type system much more powerful than Java's — one which is based upon dependent types [8] [9] [6]. The only alternative to heroic type hackery is to use a host language which is not statically typed. Dynamic languages do not solve the underlying the problem, though, because we would still like the assurance that weaving will not introduce type errors; that's part of what it means to respect the semantics of the host language.

3.2.3 The DEEP programming language

I have taken a few steps towards a DSAL toolkit in the design of the DEEP programming language [8]. DEEP is a formal language calculus which integrates dependent types, singleton types, and partial evaluation. By combining these three mechanisms together, DEEP can deal with partially static data.

The basic idea behind the DEEP type system is that the type of an expression should hold whatever information is known about that expression at compile-time. In the case of static data, the type of an expression will be a *singleton type* representing its value. For example, the type of `(1 + 2)` is `3`. The type system may partially evaluate a term in order to assign an accurate type to that term.

For expressions which are not statically known, DEEP uses dependent types, which contain a mixture of both types and values. For example, if `myList` is a list of length `3` (a dependent type) then `length(myList)` will have type `3`, even if the elements of the list are not statically known. The advantage of this scheme is that it is a good way to deal with partially static data; the disadvantage is that programming with dependent types can be notoriously tricky.

The DEEP language is based on prototypes rather than classes. At compile-time, a prototype is treated as a type for the purpose of static type checking. At run-time, a prototype is just an ordinary object: it can be stored in a field, or passed as an argument to a function. The prototype model allows classes to be created and manipulated by ordinary code, just like in CLOS and Smalltalk, without sacrificing static type safety. This model is intended to simplify the task of writing code generators.

Finally, DEEP supports *deep mixin composition* of modules. Deep mixin composition is an extension of inheritance which allows a group of classes to be encapsulated in a module, and then extended as single unit. Classes keep the same name within the module, so it appears to client code as if the classes have been updated in place. This form of composition is the same as that found in feature-oriented programming [1], multi-dimensional separation of concerns [17], and virtual classes in gbeta [5].

Deep mixin composition has some of the capabilities of aspect-weaving, but not all. It is possible to add "before" and "after" code to individual methods, but it is not possible to quantify over methods and classes, or to write general-purpose transformations. Quantification requires reflection, which DEEP does not currently support.

Summary.
To summarize, DEEP provides the following:

- A statically typed language with a powerful type system.
- A partial evaluator which supports partially-static data.
- First-class functions, classes, and modules.
- Deep mixin composition.

However, the DEEP calculus currently does *not* support reflection of any kind. The lack of reflection means that it is not possible to write generic boilerplate code, or to do general-purpose aspect weaving.

Adding simple introspection would be easy enough, but it is not sufficient for true aspect-weaving. Ideally, reflection should be integrated with the mechanism for mixin composition, so that the type system can guarantee that a particular class extension generates a proper subtype. Doing this in a way that is both type-safe, and sufficiently flexible for DSALs, remains an open problem.

4. Conclusion

Combining partial evaluation with first-class functions is an excellent way to write code generators for DSLs. Such generators are type-safe and well-integrated with the host language. Adding simple reflection to this mix allows the automatic generation of "boilerplate" code.

It may be possible to write full-blown aspect-weavers by combining partial evaluation with both reflection and first-class classes. However, if it is possible to extend classes in arbitrary ways, then the resulting transformations may not be type-safe. The mechanism for extending classes safely (i.e. inheritance) should be integrated with reflection, and the best way to do this is not obvious.

References

[1] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Proceedings of ICSE*, 2003.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3):107–126, 2005.

[3] D. Bobrow, R. Gabriel, and J. White. CLOS in Context: The Shape of the Design Space. *Object-Oriented Programming – The CLOS Perspective*, 1993.

[4] M. Braux and J. Noye. Towards partially evaluating reflection in java. *Proceedings of Partial Evaluation and Program Manipulation*, 2000.

[5] E. Ernst. Higher order hierarchies. *Proceedings of ECOOP*, 2003.

[6] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. *Proceedings of POPL*, 2006.

[7] R. Hinze. A new approach to generic functional programming. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 2000.

[8] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. *Proceedings of OOPSLA*, 2006.

[9] A. Igarashi and B. Pierce. Foundations for virtual types. *Proceedings of ECOOP*, 1999.

[10] N. Jones. Mix ten years later. *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 24–38, 1995.

[11] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. *Proceedings of ECOOP*, 2001.

[13] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

[14] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. *Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht*, 2001.

[15] C. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, 1997.

[16] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. 2006.

[17] H. O. Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer.*, 2000.

[18] F. Rivard. Smalltalk: a Reflective Language. *Proceedings of Reflection*, 96:21–38, 1996.

[19] W. Taha. A gentle introduction to multi-stage programming. *Domain-Specific Program Generation*, pages 30–50, 2003.

[20] W. Taha, H. Makholm, and J. Hughes. Tag Elimination and Jones-Optimality. *Proceedings of the Second Symposium on Programs as Data Objects*, pages 257–275, 2001.

[21] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.