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
Copyright © 2006 ACM [to be supplied]. . . $5.00

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 understand 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.