Polymorphic Embedding of DSLs

Christian Hofer Klaus Ostermann
Tillmann Rendel
University of Aarhus, Denmark

Abstract

The influential *pure embedding* methodology of embedding domain-specific languages (DSLs) as libraries into a general-purpose host language forces the DSL designer to commit to a single semantics. This precludes the subsequent addition of compilation, optimization or domain-specific analyses. We propose *polymorphic embedding* of DSLs, where many different interpretations of a DSL can be provided as reusable components, and show how polymorphic embedding can be realized in the programming language Scala. With polymorphic embedding, the static type-safety, modularity, composability and rapid prototyping of pure embedding are reconciled with the flexibility attainable by external toolchains.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages, Specialized application languages; F.3.3 [*Logics and Meanings of*

Programs]: Studies of Program Constructs

General Terms Design, Languages

Keywords Algebraic semantics, compositionality, domain-specific languages, extensibility, pure embedding, scala

1. Introduction

Over a decade ago, Paul Hudak has shown how a domain-specific language (DSL) can be implemented as a library, instead of using a full-blown, stand-alone toolchain [12, 13]. Indeed, by embedding the DSL in a rich host language, the implementation effort is reduced dramatically. More concretely, the embedded DSL (EDSL) can reuse the syntax of the host language, its module system, existing libraries, its tool chain, and so on.

Clearly, the success of this symbiosis hinges on the power and the malleability of the host language. Hudak has shown that functional abstraction is a good mechanism to express the meaning of the embedded language. However, his methodology leads to tight coupling of the host language and the embedded one. More specifically, it restricts the EDSL to a single opaque interpretation, which is not amenable to analysis or optimizations, which are crucial for improving the performance of DSL programs. In general, the restriction to a single interpretation prevents flexible reuse and composition of both DSL programs and interpretations. To improve the performance of embedded DSLs, partial evalua-

GPCE '08 October 19-23, 2008, Nashville, Tennessee

Adriaan Moors

KU Leuven, Belgium

tion and multi-staged computation have been proposed [13, 27], but domain-specific optimizations or other kinds of analyses are still not possible. More importantly, these approaches do not address the problem of keeping the interpretation of the DSL open.

In this paper, we generalize Hudak's approach to support multiple interpretations by complementing the functional abstraction mechanism with an object-oriented one: virtual types. Thus, our architecture, which we call *polymorphic embedding*, introduces the main advantage of an external DSL, while maintaining the strengths of the embedded approach: compositionality and integration with the existing language. In this framework, optimizations and analyses are just special interpretations of the DSL program.

Given the need for functional as well as object-oriented abstraction mechanisms, we chose Scala [21] as our implementation lan-

guage, although the same ideas can be realized in other languages with a similar set of features. Most importantly, we rely on Scala's support for virtual types (abstract type members) and family polymorphism [22, 7, 6], higher-order genericity [18], and mixin composition [23], in addition to minor features, such as flexible and succinct syntax.

In summary, the main contributions of this paper are:

- We present a new approach to embedding DSLs that supports multiple interpretations.
- We show that domain-specific analyses and optimizations can be expressed in this framework as yet another interpretation of the DSL program.
- We demonstrate that polymorphic embedding enables the componentization of both DSL programs and their interpretations: Interpretations can be composed from simple, reusable interpretation components, and DSL programs in different DSLs can be composed quite flexibly.
- We leverage subtype (family) polymorphism and higher-kinded abstract type members [18] in embedding the DSL's type system, which suggests that these inherently object-oriented mechanisms may have a role similar to features that are primarily known from functional languages, such as generalized abstract datatypes [31] and various forms of parametric polymorphism.
- Our design is a showcase for the practical utility of many modern object-oriented language features that are not yet adopted in main-stream languages.

The remainder of this paper is structured as follows: The next section reviews existing approaches for embedding DSLs and identifies a number of desirable properties of such approaches. Section 3 presents polymorphic embedding as a solution to the problems of pure embedding and demonstrates its feasibility in Scala. Section 4 evaluates our implementation of polymorphic embedding and discusses its limitiations. Section 5 discusses related and future work. The final section concludes.

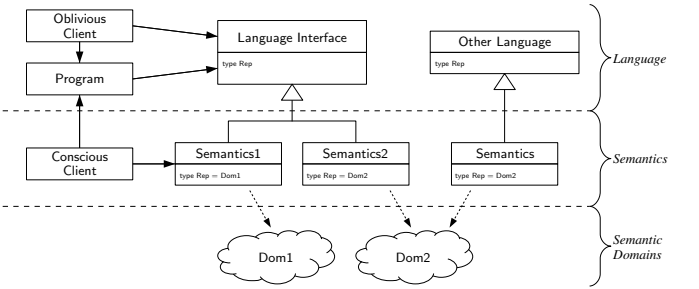

Figure 1. High-level overview of the architecture

The full code for all examples in the paper can be downloaded at: `http://www.daimi.au.dk/~rendel/dsl/`

2. Problem Statement

This section briefly reviews three different approaches to embedding a DSL into a general-purpose host language, and identifies a number of properties that are crucial to the applicability of such

an approach to a large class of DSL embedding scenarios. This article concentrates on embedding approaches, which implement a DSL in terms of the language elements of a host language, either by transforming mixed host language and DSL programs into host language programs, or by implementing DSL features directly in the host language. A broader discussion of DSL implementation techniques can be found in the comparative study of Mernik et al. [16], which introduces "Implementation patterns for Executable DSLs" in Table IX.

2.1 Code Generators

Traditionally, DSLs have been integrated into general-purpose host languages by preprocessors, which parse the mixed source files and convert the embedded DSL programs into program fragments of the host language, typically calls of library functions. The result of the preprocessing is a host-language-only source file, which can be processed by an ordinary host language compiler. This code-generator approach is quite flexible, because the preprocessor can process the DSL program in arbitrary complex ways, including domain-specific analysis and optimization. By using a different preprocessor, programmers can chose an alternative semantics for their DSL programs.

A well-known example of this approach is the yacc parser generator, which allows to mix domain-specific parsing constructs and ordinary C code. The mixed source files are translated by yacc into normal C files to be compiled by a stand-alone C compiler.

However, to adopt this approach, a DSL designer has to build a sophisticated infrastructure similar to a full-fledged compiler (scanning, parsing, etc.). The high costs of implementing a programming

language prevent the adoption of small specialized DSLs in many cases [13]. An additional problem with the code generation approach is that preprocessors are not easily composable, since they cannot parse each other's DSL syntax.

2.2 Embedded Interpreter

A DSL designer can significantly lower the costs of implementing an embedded DSL by reusing the host language parser and compiler. The idea is to write embedded DSL programs as host language expressions that create an abstract syntax tree (AST) representation of the embedded program. This AST is then processed by an interpreter implemented in the host language. This approach is described by Gamma et al. [8] as the Interpreter pattern. If the host language offers some syntactic flexibility, such as user-defined infix operators, nested calls of data constructors which construct the AST can be made to closely resemble a program in a regular DSL.

Recent research has shown that even complex type systems can be embedded into the type system of a host language with a sufficiently advanced type system [31]. The designer of an embedded DSL can reuse the syntax, semantics, standard library, and toolchain of the host language. However, this approach does not really overcome the composability problems of the code-generator approach.

If several DSLs are mixed within a single host-language expression, it is not obvious how the different semantics can be composed.

2.3 Pure Embedding

Hudak's pure embedding [13] is similar to the Interpreter pattern, except that it does not construct an explicit representation of the

DSL program. Instead, the domain types are directly implemented as host language types, and domain operations are modeled as host language functions on these types.

While this approach resembles the development of a traditional library, which also exports types and functions to its clients, it stresses the domain-specific concepts and operations during the design and implementation of the library.

A purely embedded DSL program can use a combination of several DSLs as long as they happen to agree on the implementation of the domain specific types, similarly to how different libraries can be integrated if they agree on a common set of types. Because the domain operations are defined in terms of the domain semantics, not the syntax of the DSL, this approach automatically yields compositional semantics with its well-known advantages, such as easier and modular reasoning about programs and improved composability. Nevertheless, purely embedded DSLs have to be designed for composability by using a common framework, since it is not possible to later change the representation of the domain-specific types.

Hudak argues that it is easy to prove that a DSL implementation in the pure embedded approach is consistent with the algebraic laws of the respective domain. However, while the semantics is in accordance with the algebraic laws of the domain, it cannot utilize them for optimization purposes.

It is generally recognized that languages that have been implemented by pure embedding can incur a severe performance penalty. Therefore, domain specific embedded compilers have been devised [15, 5]. For example, Elliot et al. [5] have implemented an embedded compiler that translates all the operations of their language Pan (a language for image synthesis and manipulation, and a superset of the region language that we use in this paper) to a simple first-order

target language. This required the adaption of the representation of the types as well as the operations, e.g., the representation of a point was adapted from a pair of floating-point values to a pair of abstract syntax trees representing floating-point values.

Others have worked on automating this manual approach: Hudak [13] experiments with partial evaluation, while Sheard [27] introduced explicit staging annotations into the interpretation. We conclude that optimization is a significant open problem of the pure embedding methodology.

Finally, support for multiple interpretations has many practical applications beyond optimization. Firstly, it is often useful to perform other (non-optimizing) analyses on a DSL program, such as more advanced correctness checks. Mernik et al. recommend not to use the embedding approach if "domain specific analysis, verification, optimization, parallelization or transformation is required" [16, p. 331]. Secondly, evolving an existing DSL also requires changing its interpretation, and having multiple interpretations allows the evolution to take place in a modular way. For example, the DSL can easily be adapted to be composable with another DSL by providing a corresponding interpretation into a common representation.

2.4 Desirable Properties

An optimal architecture for embedding DSLs should combine the advantages of these three different approaches to make it applicable to a large class of DSL embedding scenarios. We have identified the following properties:

- **Reuse of infrastructure**: DSL implementations can reuse syn-

tax, semantics, implementation and tools of the host language.

- **Pluggable semantics**: DSL programs can be interpreted in multiple ways rather than having *the* semantics of the DSL.
- **Static safety**: The wellformedness of DSL programs is statically checked against both syntactical (accordance to a context-free grammar) and contextual (scoping rules, type system) restrictions.
- **Compositionality**: The semantics of a DSL is specified by defining the meaning of each DSL expression exclusively in terms of the meaning of its direct subexpressions, without accessing the syntactic structure of the subexpression or information about the rest of the program.
- **Composability**: Different DSLs or slightly different semantics of the same DSL can be integrated with each other and the host language to be used interleaved, even if the DSLs were not originally designed for cooperation.
- **Modular semantics**: The semantics of a powerful DSL can be composed from more primitive semantics using a modular construction system.
- **Performance**: Implementations with reasonable performance in large-scale use are possible.

3. Architecture

Our architecture – like the pure embedding approach – enforces compositionality by restricting the DSL implementation to work directly on already interpreted semantical objects. Unlike the pure

embedding approach, which tightly couples a DSL to a specific interpretation, we achieve loose coupling by abstracting over the implementation of both the domain-specific operations, and the representation of the domain-specific types. This variation point allows the user of the DSL to plug different semantics, including analyses, as components into an embedded DSL program. Since each of the semantics and analyses is still defined compositionally, the advantages of the pure embedding approach are retained.

3.1 Participants

Fig. 1 illustrates the main elements of our architecture. The *language interface* is the central concept in our design. It contains abstract declarations of the domain-specific types and operations provided by the language. These declarations constitute the DSL's syntax and type system. Not every DSL needs to be typed, but one goal of our design is to enable typed DSLs whose type system is embedded in the host language's type system, i.e., DSL programs are automatically type-checked by the host language's type checker.

The language interface is implemented by concrete *semantics*. These map each DSL program into *semantic domains*, which encompass a set of types and the meaning of the operations on those types. Semantic domains in polymorphic embedding are similiar to the domains used in denotational semantics, but instead of mapping programs to purely mathematical objects, DSL programs are transformed into objects of the host language representing the intended meaning of the program. Each semantics is defined by giving a concrete representation to the domain-specific types and operations. On the one hand, the same program may be mapped to different semantic domains to achieve different results. For example, a DSL

implementer can define a domain-specific analysis by providing a semantics that maps into a semantic domain that represents the results of the analysis.

On the other hand, different language interfaces can be made interoperable by providing semantics which map into the same semantic domain. Since semantics can be added by users of a language interface, this is even possible if the language interfaces were not originally designed to be interoperable.

The openness of the concrete representation is indicated by the abstract type `Rep` in Fig. 1. In the general case, these abstract types may be parameterized, as illustrated by `Rep` in Fig. 5.

A *client* operates on a DSL program. We can distinguish two classes of clients: *Oblivious clients* know only about the syntax and type system of the DSL, and can hence only use the language interface of the DSL to extend the DSL program. Technically, there is no difference between an oblivious client and an extension of an existing DSL program.

On the other hand, a *conscious client* is restricted to specific semantics of the DSL program. In particular, it knows something about the representation of the domain types. This knowledge enables the client to compose a DSL program with programs written in other DSLs or with host language programs. The semantic domain serves as a common interface for this composition.

3.2 Modular Design

Our architecture opens new possibilities for modular language design and implementation, since a system of language interfaces and semantics can be considered as a component-based architecture. A component is a reusable and composable piece of software with a well-defined interface. The elements of the levels of our architec-

ture – languages, semantics and semantical domains – are all components. This new view on embedded DSLs enables programmers to apply their knowledge about design patterns and component-based design to the implementation of domain-specific languages.

We describe two important examples of component-enabled DSL implementation techniques that have proven useful in experiments with our architecture. The following sections elaborate on these techniques. However, we do not claim to have explored the whole design space yet.

On the level of languages, the language interface of a DSL may be defined using other lower-level DSLs without fixing their particular semantics. This *hierarchical definition* of DSLs enables code reuse by importing a previously implemented lower-level DSL, as well as introducing a variation point in the new DSL. It is possible to configure the low-level behavior of programs written in the higher-level DSL by choosing an appropriate semantics for the lower-level DSL.

On the level of semantics, many analyses and optimizations known from compiler design can be implemented as a *semantic transformer* that decorates an arbitrary semantics with additional processing. The additional processing could, for example, short-circuit the evaluation of subexpressions based on algebraic laws of the domain. Since semantic transformers are oblivious with respect to the decorated semantics, they can be chained by conscious clients to form the precise semantics needed.

3.3 The *Regions* Language

The remainder of this section demonstrates our architecture by

means of simple DSLs embedded in Scala.

Inspired by [13] and [4], we want to represent a language from the domain of image processing in our framework. For presentation purposes, our DSL is kept too simple to be very useful, but it would be straightforward to extend it with more types and operations, without changing anything in the basic architecture.

The interface of the *Regions* language consists of one domain specific type `Region`, some primitive constants to describe elementary regions, and some combinators to build more complex regions out of elementary ones.

A language interface is represented by a trait in Scala, as illustrated in Fig. 2. A trait is an abstract class that allows mixin composition. Abstract classes can contain (abstract) type and value members. A value member is either a method (**`def`**), an immutable field (**`val`**), or a mutable one (**`var`**). The `Regions` trait defines two type members (**`type`**). The first is a type synonym which declares a `Vector` to be identical to a pair of `doubles`. Pairs are directly supported by Scala with the notation `(first, second)`. The second type member declares the abstract domain type `Region` (the `Rep` type for this particular DSL). The value members declare a collection of operations to construct both elementary and complex regions. Abstract members (types and values) must be made concrete before the class that defines them can be instantiated.

The language interface shown here is restricted to five operations. The elementary regions are: `empty`, the region that does not contain any points, `univ` for the universal region containing every point, and `circle` for a unit circle around the origin of the coordinate space. The function `scale(v, x)` scales the whole region `x` by the vector `v`, and `union(r, s)` combines two regions `r` and `s`.

3.4 Programs and Oblivious Clients

Both DSL programs and oblivious clients depend only on the language interface, not on a particular semantics. They are implemented in Scala by accepting the actual semantics as a parameter, either by encoding the DSL program as a method or by providing the parameter in some enclosing scope. Since the result type of a program depends on the actual semantics used, the signature of such a method requires advanced type system features. In Scala,

```
// Language interface
trait Regions {
  // Ordinary type synonyms
  type Vector = (double, double)

  // Abstract domain types
  type Region

  // Abstract domain operations
  def univ : Region
  def empty : Region
  def circle : Region
  def scale(v : Vector, x : Region) : Region
  def union(x : Region, y : Region) : Region
}

// A simple program
def program(semantics : Regions)
                              : semantics.Region = {
  import semantics._
  val ellipse24 = scale((2, 4), circle)
  union(univ, ellipse24) // The returned expression
```

```
}
```

Figure 2. *Regions* Language interface in Scala

there are two options: either using path-dependent types, or using generics.

In the example program in Fig. 2, the semantics to be used is passed as a parameter to the program. The return type is a dependent type, which depends on the value `semantics`. Support for dependent method types is not yet part of standard Scala and must be activated explicitly with a compiler option (`-Xexperimental`). The second option is to use generics, namely by writing the `program` method as a generic class as follows:

```
class program[R](val
            semantics: Regions{type Region = R}) {
  def apply: R = { import semantics._
    val ellipse24: R = scale((2, 4), circle)
    union(univ, ellipse24)
  }
}
```

However, this workaround is more verbose and has significant disadvantages (see also Sec. 4.2 and 5.1), hence in the remainder of this paper we use dependent method types.

Instead of writing `semantics.union(semantics.univ, ellipse24)`, i. e., qualifying each name in the *Regions* program with the `semantics` scope qualifier, we use the **`import`** statement of Scala with its flexible scope control to make all members of `semantics` directly available in the class. The same can be done on the level of methods and even individual expressions.

3.5 Semantics

We can now implement different semantics for the *Regions* language. The first semantics is a direct embedding into the host language. It can be used to check whether a point is inside or outside the region described by a *Regions* program. The second semantics is a pretty printer which is useful for debugging and other activities.

A concrete semantics for the *Regions* language is given by extending the language interface trait and implementing the abstract type and all abstract operations. This works similarly to the overriding of abstract methods in other OO languages. Figure 3 shows the direct embedding semantics of the *Regions* language. In this semantics, regions are seen as sets of points, and represented as the characteristic function of those sets. The characteristic function maps points in the region to `true` and points outside the region to `false`.

```
trait Evaluation extends Regions {
  type Region = Vector ⇒ boolean

  def univ : Region = p ⇒ true
  def empty : Region = p ⇒ false
  def circle : Region =
    p ⇒ p._1 * p._1 + p._2 * p._2 < 1
  def scale(v : Vector, x : Region) : Region =
    p ⇒ x(p._1 / v._1, p._2 / v._2)
  def union(x : Region, y : Region) : Region =
    p ⇒ x(p) || y(p)
}

object Eval extends Evaluation

trait Printing extends Regions {
```

```
  type Region = String

  def univ : Region = "univ"
  def empty : Region = "empty"
  def circle : Region = "circle"
  def scale(v : Vector, x : Region) : Region
    = "scale(" + v + ",␣" + x + ")"
  def union(x : Region, y : Region) : Region
    = "union(" + x + ",␣" + y + ")"
}

object Print extends Printing

// prints "union(univ, scale((2.0,4.0), circle)"
println(program(Print))

// prints "true"
println(program(Eval)((1, 2)))
```

Figure 3. Concrete semantics

The trait `Evaluation` implements the *Regions* language interface by specifying that the abstract type `Region` should be the type of functions from `Vector` to `boolean`. This is expressed in Scala with the help of the ⇒ keyword, which is also used in the definitions of the domain operations to construct anonymous functions. For example, the universal region contains all points, so `univ` maps all arguments `p` to `true`. The definition for `circle` reads as: a point `p` is part of the unit circle if the squared sum of its components is less or equal than one. In the definition of the union operation, we can use the characteristic functions of the two regions we want to

combine to map all points to `true` which are mapped to `true` by any of the composed regions. It is the essence of the compositional approach that the definition of an operator like union works on already interpreted regions, instead of getting term representations as parameters and manually converting these to regions by recursively calling an interpreter.

The pretty printing semantics is implemented as another trait extending the *Regions* language interface. This time, the domain type is mapped to `String`, and each of the domain operations are implemented by composing strings to produce a text representation of the *Regions* program. We define both semantics as traits in order to make them reusable. As traits cannot be instantiated, we have to create classes `Eval` and `Print` from them before we can use them. We use the **`object`** keyword to declare singleton classes, i. e., classes with exactly one instance.

3.6 Modular Semantics

In this section, we discuss how to define a simple optimization of the *Regions* language as a reusable component that can be composed with every other *Regions* semantics. For example, it can be used together with the evaluation semantics from the previous sec-

```
trait Optimization extends Regions {
  val semantics : Regions

  type Region = (semantics.Region, boolean)

  def univ : Region = (semantics.univ, true)
  def empty : Region = (semantics.empty, false)
  def circle : Region = (semantics.circle, false)
```

```
  def scale(v : Vector, x : Region) : Region =
    if (x._2) (semantics.univ, true)
    else (semantics.scale(v, x._1), false)

  def union(x : Region, y : Region) : Region =
    if (x._2 || y._2) (semantics.univ, true)
    else (semantics.union(x._1, y._1), false)
}

// prints "union(univ, scale((2.0,4.0), circle)"
println(program(Print))

object OptimizePrint extends Optimization {
  val semantics = Print
}

// prints "(univ, true)"
println(program(OptimizePrint))
```

Figure 4. Optimization

tion to form an optimizing interpreter, or with the pretty printer to form a pretty printer that outputs the result of the optimization process. Since optimizations are expressed as semantics, they are not performed at host language compile time, but during the construction of the denotation of a DSL program. Nevertheless, we regard such optimizations as *static*, since an embedded DSL program can

be executed multiple times after its optimized denotation has been constructed.

Odersky and Zenger [23] give an in-depth account on how to build component-based systems using Scala. We adopt the hierarchical style of component composition described there by defining the `Optimization` component as a trait with an abstract value as shown in Fig. 4. Similar to abstract types and abstract methods, abstract values have to be defined elsewhere before instances can be created.

The optimization shown here tries to statically determine whether a region is the universal region and to simplify computations based on this knowledge. For example, the union of a region with `univ` is always the `univ` region. The key to make this possible is to keep track of additional information associated with every region: Optimized regions are represented as a pair of an unoptimized region and the additional information whether the unoptimized region is statically known to be universal. The components of a pair can be accessed with `pair._1` and `pair._2`.

The implementation of the domain operations can utilize the semantics to be optimized via the *Regions* language interface of the latter. The operations `univ`, `empty` and `circle` are implemented by adding the appropriate additional information to whatever the underlying semantics does. The actual optimization is done in the implementations of `scale` and `union`, where the additional information is inspected to shortcut some evaluation paths.

Since the `Optimization` component works on arbitrary *Regions* semantics, it can be combined with both the `Printing` and `Evaluation` semantics.

```
trait Functions {
  // Abstract domain types
```

```
  type Rep[X]

  // Abstract domain operations
  def fun[S, T](f : Rep[S] ⇒ Rep[T]) : Rep[S⇒T]
  def app[S, T](f : Rep[S⇒T], v : Rep[S]) : Rep[T]
}
```

Figure 5. The *Functions* language

3.7 The *Functions* Language

The *Regions* language we have discussed so far is very simple. In particular, it only contains a single type of domain objects: regions. In this section, we will demonstrate that more powerful languages with non-trivial type systems can be used in our architecture as well. We have chosen to define an embedding for the simply-typed lambda calculus. The language interface in Fig. 5 defines an abstract type operator (`Rep`) and the two operations for lambda abstraction (`fun`) and application (`app`).

Since the *Functions* language has a more elaborate type system compared with the *Regions* language discussed before, its language interface uses an abstract type operator and generic method declarations to handle higher-order functions. We use the idea of higher-order abstract syntax (HOAS) [25] to reuse Scala's binding mechanisms and scoping rules. Each semantics of the *Functions* language has to specify, by implementing `fun`, how functions are represented internally. The method `fun` takes an ordinary Scala function as parameter and returns the internal representation. If we ignore the `Rep` type operator for the moment, we can, for example, represent a host language function that adds `3` to a number, as a DSL function by

writing `fun((x : double) ⇒ x + 3)`.

However, each interpretation of the *Functions* language can require a specific representation of the types it operates on. For example, a pretty printer will represent all types as strings, while an evaluator can operate on arbitrary types as is. The choice of representation is defined by the type operator `Rep`. A type operator maps types to types. The Scala syntax for type operators requires to specify the type parameters within square brackets. In the example, the `Rep` type operator takes one type parameter `X`. As we will see below, the pretty printer implementation will map each type `X` to `String`, while the evaluator will map each type to itself.

3.8 Semantics of the *Functions* Language

We can now define a simple evaluator and a pretty printer of the *Functions* language, as shown in Fig. 6.

The evaluator operates on all kinds of terms as is, therefore its type operator `Rep` maps all types to themselves. It can use the type parameters to specify this type. Lambda abstraction and application are interpreted trivially as Scala's own lambda abstraction. The implementations of `fun` and `app` do not contain any special processing, because HOAS allows us to reuse Scala's abstraction mechanism.

The pretty printer represents each type as a string. This is reflected in the definition of `Rep`, which maps all types to `String`, and in the type of the `fun` operator, which takes a function from `String` to `String` as parameter. The generic type arguments `S` and `T` are not used in this simple semantics. A disadvantage of using the HOAS approach is that we cannot reify the variable name the user has chosen. We therefore have to generate a new variable

name, which is done by `variables.next` This generated name is used twice to compose the string representation of the lambda expression: Both as bound variable to the left of the "⇒" sign, and as argument to the function parameter to the right. The function pa-

```
trait FunEval extends Functions {
  type Rep[T] = T
  def fun[S,T](f : S ⇒ T) = f
  def app[S,T](f : S⇒T, v : S) : T = f(v)
}

trait FunPrinting extends Functions {
  type Rep[X] = String

  def fun[S,T](f : String ⇒ String) : String = {
    val v = variables.next
    "fun(" + v + " ⇒ " + f(v) + ")"
  }

  def app[S,T](f : String, v : String) : String =
    "app(" + f + ", " + v + ")"
}
```

Figure 6. A *Functions* evaluator

rameter will produce a string representation of the function body, with the generator variable name inserted where the programmer used the lambda-bound variable originally. The rest of the definition is straight-forward.

3.9 Hierarchical Language Composition

In Sec. 3.6 we have demonstrated how to compose different interpretations of one language. In this section, we will discuss how different languages can be composed. There are several ways in which languages can inter-operate. In the simplest case, one language is simply an extension of another language, adding new operations that operate on the same types. This case can straightforwardly be handled by using inheritance: the extended language interface inherits from the interface of the other language; accordingly, all extended language implementations inherit from the corresponding implementations.

This solution can also be used when the extension adds new types as well as new operations. To consider a simple case, let us turn back to the *Regions* language. So far, we have regarded vectors as being an integral part of it, having a fixed interpretation as a pair of `double` values. Instead, we could consider an independent language on vectors that is able to operate on polar as well as on Cartesian coordinates. Using inheritance for extending the vector language to a *Regions* language, however, can pose problems in relation to using modular semantics as in Sec. 3.6. In Fig. 4, we implicitly make use of the fact that all *Regions* language implementations build on top of the same representation for vectors. If we factor out an independent vector language, and make the `Regions` trait inherit from it, we would have to manually define the vector type and all vector operations in the `Optimization` trait to delegate to the corresponding definitions in the `semantics` subcomponent. This would not be modular with respect to changes in the vector language interface.

A better solution is to organize the two languages *hierarchi-*

cally (see also [23]), by making the vector language a field in the *Regions* language as shown in Fig. 7. Scala's **import** statement ensures that the operations on vectors are available in the *Regions* language when needed. The rest of the code remains the same. The `Optimization` trait then simply defines that it uses exactly the same vector language implementation as the `semantics` component. On the type level, this is expressed by declaring `vec` to be of the *singleton type* of the value `semantics.vec`.

A second interesting case of hierarchical composition is that a different (lower-level) DSL is used in the implementation of a specific semantics similiar to domain specific embedded compilers [15, 5]. From a compiler perspective one could say that the lower-

```
trait Regions {
  val vec : Vectors
  import vec._
  ...
}

trait Optimization extends Regions {
  val semantics : Regions
  val vec : semantics.vec.type = semantics.vec
  import vec._
  ...
}
```

Figure 7. A hierarchical composition

```
trait Base {
  // Domain types
```

```
  type BooleanR
  type FloatR
  ...
  def bool(b : Boolean) : BooleanR
  def or(a : BooleanR, b : BooleanR) : BooleanR
  ...
}

trait Evaluation extends Regions {
  val base: Base
  import base._
  type Region = ((FloatR, FloatR)) ⇒ BooleanR
  ...
  def union(x : Region, y : Region) : Region =
    p ⇒ or(x(p),y(p))
  ...
}
```

Figure 8. Translation to intermediate language

level DSL acts as a kind of intermediate language for the higher-level DSL. Figure 8 sketches how the `Evaluation` semantics from Fig. 3 can be turned into a translator to an intermediate language of arithmetic and boolean expressions. The advantage of this design is that it enables to plug in different semantics for the intermediate language, and in particular optimizations for it – similar to how optimizations in a stand-alone compiler are organized along various intermediate formats. Unlike traditional domain specific embedded compilers, polymorphic embedding allows a code-generating semantics in parallel with other interpretations.

3.10 Peer Language Composition

The arguably most challenging composition is the composition of languages which are completely independent, like the *Regions* and the *Functions* language. We can use the latter to create functions that map regions to regions, or even higher-order functions that map functions on regions to functions on regions. Note that there is no client-provider relationship between the two languages: the *Functions* language can embed terms of the *Regions* language and vice versa. As we cannot recognize a clear hierarchy between the two languages, we will integrate them as peers via *mixin composition*. This, however, is a design decision which has to be made carefully in practice, as using mixin composition has the same drawbacks concerning modularity as the single inheritance approach dismissed in Sec. 3.9.

Independently of the kind of composition we choose, we can only combine an implementation of the *Regions* language with an implementation of the *Functions* language if we can find a translation between the way regions are represented in the two languages. In the *Regions* language, they simply have the type `Region`, while

```
trait FunReg extends Regions with Functions {
  implicit def fromRegion(r: Region): Rep[Region]
  implicit def toRegion(r : Rep[Region]) : Region
}

def program(semantics : FunReg) :
                semantics.Rep[semantics.Region] = {
  import semantics._
  app(fun((x : Rep[Region]) ⇒
            scale((5,2),x)),empty)
```

```
}
```

Figure 9. Integrating functions and regions

```
object FunRegEval extends FunReg with Evaluation
     with FunEval {
  implicit def fromRegion(r:Region):Rep[Region] = r
  implicit def toRegion(r:Rep[Region]):Region = r
}

object FunRegPrinting extends FunReg with Printing
     with FunPrinting {
  implicit def fromRegion(r:Region):Rep[Region] = r
  implicit def toRegion(r:Rep[Region]):Region = r
}

// prints "app(fun(x1 ⇒ scale((5,2), x1)), empty)"
println(program(FunRegPrinting))

// prints "false"
println(program(FunRegEval)(2,3))
```

Figure 10. Integrating the semantics

the *Functions* language expects them to be represented by the type `Rep[Region]`. We express the existence of this translation in the interface of the integrated language, as shown in Fig. 9.

We use **`implicit`** definitions for these translations. Scala's *implicits* [18] make the compiler insert these method calls automat-

ically if they reconstitute type correctness. If we had used ordinary method definitions, we would have to encapsulate every region term in the DSL program by a call to `fromRegion` to make it a term of the *Functions* sub-language, and vice versa use `toRegion` for the other direction. In this case, the use of the variable `x` within the `program` method is automatically translated to type `Region`, while the constructor `empty` is translated to type `Rep[Region]`. Since Scala has only local type inference, we have to specify the type of the introduced variable `x` explicitly.

We can now compose the evaluation and the pretty printing semantics of the two languages accordingly, as shown in Fig. 10.

In both cases, the translation between the representations is trivial. In the first case, **`type`** `Rep[T] = T` is explicitly specified by the `FunEval` trait, which also holds for regions. In the second case, both interpretations map regions to strings. In general, however, the translation is not trivial. For example, one of the pretty printers could represent expressions using a structured `Text` type rather than plain strings, and then a conversion would be necessary.

It would be natural to combine the *Functions* language with one of the optimizers of the *Regions* language. However, if the `Evaluation` trait in Fig. 10 is just replaced with an optimized version, the optimization is delayed within function bodies until the actual execution of the embedded DSL program. Instead, optimizations should be applied during the construction of the denotation of a DSL program as explained in Sec. 3.6. To propagate the optimization into the function bodies, it seems to be necessary to resort to an AST representation of the function body, or to restrict the combined language so that only functions over regions (but not arbitrary functions) can be constructed. That would allow branching based on the type of the function (e.g. `Rep[Region⇒Region]`)

and enable optimizations to handle functions by pattern matching on the type of the parameter and constructing adequate arguments. Unfortunately, the type language of Scala seems to be too weak to express this. Further analysis and possible solutions of this problem are part of our future work, and we position this problem as a challenge to type system designers.

4. Evaluation

In this section, we want to assess how well the properties that we have stated can be achieved with our architecture, in particular with respect to its implementation in Scala.

4.1 Reuse of Infrastructure

As we are using a pure embedding approach, we are able to reuse the infrastructure provided by the host language. We use Scala method calls as the DSL syntax. We do not have to extend the semantics of Scala; the meaning of the different DSL implementations is given by the semantics of the actual Scala code. We can directly reuse built-in or library-provided Scala functionality, e. g., floating-point arithmetics for defining the evaluation of the `circle` method in Fig. 3.

However, reuse of the infrastructure still has some limitations. First, the Scala language tools are not aware of the DSL abstractions. For example, a DSL user who is running the debugger will have to operate on the implementation level of the language. However, compared with a stand-alone DSL implementation, at least *some* debugging support comes for free.

Second, Scala does not offer a possibility to get a grip on

Scala expressions that are not defined via a language interface. For example, if we use Scala arithmetics, like multiplication, we cannot later redefine what multiplication is. It is always possible, though, to create a language interface for the respective parts of the Scala language or its libraries, as sketched in Fig. 8, to circumvent this problem in Scala's support for polymorphic embedding.

The degree to which the desired syntax of the DSL can be encoded in the host language's syntax depends on the flexibility of the host language's syntax. Scala's flexible import statements and *implicits* help a lot in keeping the DSL programs succinct. There are some limitations with respect to infix syntax, though. We have used a simple prefix notation in the code examples in this paper, but an infix notation would have been more suitable for some of the domain-specific operators. For example, instead of `union(univ, circle)`, it would be more natural to write `univ || circle` and overload the `||` operator to stand for `union`. While Scala allows to define user-defined infix operators, this flexibility is tightly coupled to the use of objects, because `x + y` is always interpreted as `x.+(y)`. Hence, to use infix notation, these operations would have to be methods of the abstract types such as `Region` rather than of the enclosing class. We have experimented with different styles of how to enable infix notation via encodings of virtual classes [7] in Scala, but since these styles require more infrastructural code we decided to stay with prefix notation in this paper.

4.2 Pluggable Semantics

We have shown in Sec. 3.5 how different semantics can be plugged into a language interface. These can be completely different interpretations, or they may be analyses or optimizations that are related

to one semantics.

From a usage perspective, we can represent DSL programs as first-class functions mapping each semantics to a domain value for this semantics. When using the dependent method type compiler extension, Scala's type inference is powerful enough to handle the type dependency of the domain value without requiring to pass a type parameter around. Without dependent method types (i.e., using generics), the type parameter would have to be declared on all methods which directly or indirectly refer to a DSL program without deciding its semantics. Furthermore, the original call sites would have to know the exact type of the argument, which excludes the usage of subtyping and other forms of existential typing, as required, e.g., for heterogenous collections [6, Sec. 3.2].

4.3 Static Safety

As we are using a pure embedding, the syntactic well-formedness is guaranteed by the Scala syntax checker. Furthermore, we reuse Scala's type checker for ensuring type correctness of DSL use and implementation. We have also shown that it is possible to embed the type system of the simply-typed lambda calculus using higher-kinded generics [18]. However, one can certainly fancy more complex DSL type systems that cannot (in an obvious way) be embedded in Scala's type system. The question which kinds of type systems are desirable for practical DSLs and what kind of host language type system is ideal for their embedding is an area of future work.

4.4 Compositionality

The way the language interface for a DSL is defined in our archi-

tecture enforces the compositionality property, as the interpretation of each expression cannot pass hidden information to the interpretation of its sub-expressions and vice versa. Mathematically, this can be recognized as an algebraic approach, where each implementation is ensured to be a structure-preserving map (homomorphism) from the initial algebra to the interpretation.

Naturally, we cannot strictly enforce compositionality in a host language which allows for side-effects. This can also be regarded as an advantage, as it is easier to provide debugging information in this way, knowing that compositionality is not an issue in that regard. While it is the responsibility of the DSL implementer to use the host language appropriately, the approach certainly focuses on compositionality as the right default.

4.5 Composability

In the traditional pure embedding approach, two languages are composable if they share a common representation for the types that are relevant in the composition. Polymorphic embedding generalizes this property, in that it is sufficient to give an interpretation for both languages that results in a common representation of those types. Composability is a direct consequence of compositionality. We can be sure that an interpreter does not use implicit non-local knowledge that cannot be guaranteed to be available, when languages are combined.

We have demonstrated several dimensions of composability. Conscious clients can compose DSL programs in different languages (or DSL programs with host language programs) via common domains. A language interface can be defined on top of one or more lower-level DSLs, as illustrated in Fig. 7. A high-level DSL

semantics can be "compiled" into an intermediate language DSL for optimization (Fig. 8). Independent DSLs can be composed, as illustrated in Fig. 9. One can also imagine cases where languages with similar semantics have to be merged. For example, if we have a language of colored shapes and a language of regions, we can intuitively regard regions as black-and-white shapes. We can therefore find a common representation for shapes and regions, e. g., as functions from points to colors, and define appropriate interpretations for both languages. There is also potential for sharing optimizations. The "known-to-be-universal" analysis can be generalized to a "known-to-be-single-colored" analysis.

However, composition of independent languages has its own challenges, as we have discussed in Sec. 3.10. It is not necessarily straightforward to unify different representations. Furthermore, the composition of two language implementations does not necessarily bring forth the intended result. In the example discussed, the combination of the *Functions* evaluator with the *Regions* optimization would delay the optimization until the function is applied. One might expect, instead, to have the optimization also operating on the function bodies.

Finally, there is still a design choice to make with respect to deciding for hierarchical versus peer composition. Only the former operates well with the modular semantics property, while the way in which two languages should be ordered in the hierarchy is not always obvious.

4.6 Modular Semantics

In Sec. 3.6 we have shown how different semantics of a language can be regarded as reusable components that can be put together

to construct new interpretations. We defined an optimization component that could be combined with another interpretation component to construct an optimized version of the latter component. We have furthermore indicated in Sec. 3.9, that this style of component integration coordinates well with the hierarchical composition of languages. We believe that it is even possible to create libraries of useful semantics combinators to compose interpretations in combinator style, but this is part of our future work (see Sec. 5.6).

4.7 Performance

We have indicated how optimizations can be performed within the architecture in Sec. 3.6 on the level of using algebraic laws. The optimizations that we have shown here are very simple, but it is also possible to define quite sophisticated analyses in this style. In an imperative sample DSL (code available on the paper's website) we have implemented a full-fledged data-flow analysis in the style shown in this paper. We have also shown that it is possible to "compile" DSLs to intermediate DSLs, which can then have their own optimizations and analyses. In the case of typed DSLs (such as our *Functions* language), the embedding of the type system in the host language further improves the performance of the interpretation, since no dynamic type checks have to be performed.

There is one significant catch, though: all optimizations and analyses have to be defined compositionally. This is not the way many analyses are usually described. For example, a data-flow analysis is usually described as a non-compositional fixed-point iteration over the control flow graph. In our example, we made the analysis compositional by choosing first-class functions (the

transfer function of the respective statement) as the domain of the interpretation. However, this style of defining the analysis is unusual and it is not trivial to convert an analysis into it. Hence, it is currently not clear how well this approach to optimization will work in practice for sophisticated optimizations.

5. Related and Future Work

5.1 Domain-specific Embedded Languages

Traditional approaches to the embedding of EDSLs have already been discussed in detail in Sec. 2. To get more practical experience with our architecture, we are currently working on implementing a non-trivial, practical parser DSL in the style of [14] in our architecture.

The approach by Carette et al. [1] was an important influence and is closely related to this work. Carette et al. implement a family of interpretations in ML and Haskell – encompassing an interpreter, a compiler and a partial evaluator – that operate on the same term representation of a simply-typed lambda-calculus DSL. This means, that the same purely embedded DSL program can be interpreted in different ways within the execution of the host language program. Their work can hence be seen as a realization of polymorphic embedding in Haskell and ML. However, the goal of their work is quite different: It concentrates on finding ways to implement tagless staged interpretations for typed higher-order functional languages, and as such does not elaborate on the ramifications of polymorphic embedding for general DSL development, which is the main focus of this work. There are also significant technical differences. First, their Haskell and ML solutions depend

on forms of parametric polymorphism, which – in contrast to the family polymorphism used in our Scala solution – hinders subtyping and other forms of existential typing, see also Sec. 4.2. Second, the optimizations described in the paper are – while being more sophisticated than the ones presented here – coupled to specific interpretations and not composable with other interpretations as in our architecture.

5.2 Programming Language Semantics

Our approach to give semantics to DSLs is similar to both denotational semantics (DS) [19] and initial algebra semantics (IAS) [10]. Our language interfaces such as the `Regions` trait in Fig. 2 can be viewed as abstract syntax definitions from the perspective of DS and as an algebraic signature or inital algebra for that signature from the perspective of IAS. A concrete semantics as in Fig. 3 can be seen as a compositional semantic function (DS) or as homomorphisms from the initial to the target algebra (IAS). The main difference to these approaches is that the denotations in our architecture are not mathematical objects but Scala objects.

Modularity in PL semantics is a well-known topic [17, 20] but these approaches usually focus on adding and defining particular language features such as side effects in a modular way, but not on the composition and decomposition of different interpretations or analyses. Depending on the kind of DSLs, these techniques (such as monads) would also be useful in our approach to structure the semantic functions.

5.3 Generic Programming

Our design can also be seen as a powerful form of generic pro-

gramming. In GP, a generic *algorithm* describes its syntactic and semantic requirements on one or more (abstract data) types in the form of a *concept*. Types that meet the requirements of a concept are said to *model* the concept [9]. GP is related to DSLs through the following set of equations:

Algorithm	=	program in DSL
Concept	=	syntax and type system of DSL
Model	=	DSL semantics

5.4 Attribute Grammars

Attribute grammars are a seasoned declarative formalism for specifying the semantics of programming languages. The methodology of defining attributes for nodes in the syntax tree has turned out to be particularly useful for generating compilers, but has also been applied for other kinds of language processors [24]. Recently, two systems based on the attribute grammar methodology have been developed independently which allow for extending the Java language with domain-specific extensions [3, 30]. Both supply a definition of the Java language based on attribute grammars, into which language extensions can plug their own attribute definitions. As the lexical environment can be modelled via attributes, the extensions can even introduce their own binding constructs at the correct scope.

Composing several language extensions can, on the one hand, lead to lexical/syntactic ambiguities, on the other hand, they can lead to circular attribute specifications. The former problem can be avoided by restricting the class of grammars, while the latter can only be checked at extension composition time [30]. Compositionality furthermore depends on the language extensions to be restricted to local transformations.

Systems based on attribute grammar in essence provide language developers with more flexibility, for the price of not guaranteeing compositionality. In addition, developers have to use an external framework based on attribute grammars in order to specify their extensions. For example, in ableJ [30], where scanner and parser generators are part of the Silver framework, defining the language extensions still requires defining a grammar and a two-phase process creating an abstract syntax tree from a concrete one. In contrast, we build on top of a pure embedding, implying more restrictions on how a DSL can be defined, but in exchange admitting the programmer to stay within the host language. And although polymorphic embedding does not solve all of the issues resulting from the composition of independent languages (see Sec. 4.5) either, it is, because of its more restrictive approach, able to provide guarantees on compositionality.

5.5 Metaprogramming

The problems of embedding DSLs have also been tackled by various approaches of metaprogramming (see the discussion in [2]). These approaches can be regarded as *code generator* approaches, as they convert an AST representation of the embedded DSL program at compile time into host language code. However, they do not use a separate preprocessor, but rely on the metaprogramming capabilities of the host language to express this transformation. The multi-stage programming approach [29] can even interpret a program in several different phases, similar to the optimization we discussed in Sec. 3.6, which operates at run time, but at a different phase than the actual processing of a region.

An interesting case study of using metaprogramming for DSL

implementation is given by Seefried et al. [26]. The authors use *Template Haskell* [28] to provide *PanTHeon*, an implementation for the image synthesis and manipulation language *Pan* [4].

Using this methodology, language implementation can be regarded as a two-step process. On the one hand, the implementer provides a specific implementation of the language, which in this case is very similar to the evaluator semantics of the *Regions* language. However, this implementation is not used as an actual Haskell library, but is reified as a Haskell AST. On the other hand, the implementer provides syntax transformations that operate on the AST to optimize the resulting Haskell program, which is then fed to the Haskell compiler. The user can program in *PanTHeon* exactly as in the embedded language *Pan*, with the only difference that the complete DSL program has to be enclosed in brackets, and is therefore represented as a Haskell AST as well. In contrast to polymorphic embedding, this approach depends on specific language-mechanisms for metaprogramming. Furthermore it does not allow for defining different implementations of a language on the DSL level, but only allows for applications of analyses and optimizations on the produced Haskell code. However, it should be possible to compose several DSLs which make use of the same representation of shared types. It would be interesting to see how the different analyses could be made interoperable.

5.6 Semantics Combinators

An important area of future work are *semantics combinators*, whose purpose is to combine different semantics of a DSL. A simple example is building the product of two semantics, i. e., combining both interpretations of a language term as a pair. Unfortunately,

if we try to express this combinator within the architecture, we have to redefine all language operations, as shown in Fig. 11. In princi-

```
trait RegionsProduct extends Regions {
  val r1 : Regions
  val r2 : Regions
  type Region =  (r1.Region, r2.Region)
  def univ : Region = (r1.univ, r2.univ)
  def union(x : Region, y : Region) : Region =
    (r1.union(x._1, y._1), r2.union(x._2, y._2))
  // ... similarly for all methods of Regions
}
```

Figure 11. Sample Semantics Combinator

ple this code could be generated via macros or advanced techniques such as morphing [11].

We could also think of making the product combinator an instance of a *ZipWith* combinator that takes two semantics and builds a combined semantics operatorwise. We intend to use such combinators to decompose a semantics into very simple, fine-grained, and reusable analyses and optimization steps that are composed in a combinator-style fashion. We also intend to investigate how to express these combinators in a category-theoretic framework.

6. Conclusions

We have shown that, given appropriate host language abstractions, pure embedding can be made to support multiple interpretations,

yielding polymorphic embedding. We have also demonstrated that polymorphic embedding addresses many of the problems of classical embedded DSLs, such as domain-specific analyses and optimizations. Scala turned out to be an excellent language to realize polymorphic embedding, but we also identified several shortcomings with regard to its syntactic flexibility and type system, which calls for more research in this area. Finally, we drew several interesting connections to other fields: The "languages as components" point of view yields a connection to generic programming. The "semantic domains as components" perspective yields a connection to denotational semantics, applicative functors, monads and so on. The "semantics as components" point of view yields interesting future work on semantics combinators and their algebraic or category-theoretic interpretations.

Acknowledgments

The authors would like to thank the reviewers for their insightful suggestions that helped improve the paper. Christian Hofer and Tillmann Rendel are supported by the ScalPL project of the European Research Commission. Adriaan Moors is supported by a grant from the Flemish IWT.

References

[1] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated. In *Asian Symposium on Programming Languages and Systems (APLAS'07)*, pages 222–238. Springer LNCS 4807, 2007.

[2] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In

Domain-Specific Program Generation, volume 3016 of *LNCS*, pages 51–72. Springer, 2003.

[3] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA '07: Proceedings of the 22th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–18. ACM, 2007.

[4] C. Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.

[5] C. Elliott, S. Finne, and O. D. Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[6] E. Ernst. Family polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326, London, UK, 2001. Springer-Verlag.

[7] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *33rd ACM Symposium on Principles of Programming Languages (POPL'06)*. ACM SIGPLAN-SIGACT, 2006.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, Boston, MA, 1995.

[9] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, 2007.

[10] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, 1977.

[11] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings ECOOP'07*, pages 399–424. Springer LNCS, 2007.

[12] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4), 1996.

[13] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[14] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[15] S. Kamin. Research on domain-specific embedded languages and program generators, 1998.

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

[17] E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science, Springer LNCS 530*, pages 138–139, 1991.

[18] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of OOPSLA '08 (to appear)*, 2008.

[19] P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 575–631. 1990.

[20] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[21] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[22] M. Odersky, V. Cremet, C. Rckl, and M. Zenger. A nominal theory of

objects with dependent types. In *Proceedings ECOOP '03*. Springer LNCS, 2003.

[23] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05*, pages 41–57, New York, NY, USA, 2005. ACM.

[24] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[25] F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

[26] S. Seefried, M. M. T. Chakravarty, and G. Keller. Optimising embedded DSLs using Template Haskell. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 2004.

[27] T. Sheard. Languages of the future. In *OOPSLA '04 Companion*, pages 116–119, New York, NY, USA, 2004. ACM.

[28] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[29] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation, Springer LNCS 3016*, pages 30–50, 2003.

[30] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *ECOOP'07*, LNCS. Springer Verlag, July 2007.

[31] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, NY, USA, 2003. ACM.