Modular Domain-Specific Language Components in Scala

Christian Hofer
Aarhus University, Denmark
chmh@cs.au.dk

Klaus Ostermann
University of Marburg, Germany
kos@informatik.uni-marburg.de

Abstract

Programs in domain-specific embedded languages (DSELs) can be represented in the host language in different ways, for instance implicitly as libraries, or explicitly in the form of abstract syntax trees. Each of these representations has its own strengths and weaknesses. The implicit approach has good composability properties, whereas the explicit approach allows more freedom in making syntactic program transformations.

Traditional designs for DSELs fix the form of representation, which means that it is not possible to choose the best representation for a particular interpretation or transformation. We propose a new design for implementing DSELs in Scala which makes it easy to use different program representations at the same time. It enables the DSL implementor to define modular language components and to compose transformations and interpretations for them.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable Libraries; D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages, Specialized Application Languages

General Terms Languages, Design

Keywords Embedded Languages, Domain-Specific Languages, Term Representation, Visitor Pattern, Scala

1. Introduction

The methodology of domain-specific embedded languages, where a domain-specific language (DSL) is embedded as a library into a typed host language, instead of creating a stand-alone DSL, is nowadays well-known. It goes back to Reynolds [26] and has been systematically described by Hudak [12]. It is ideal for prototyping a language for two reasons. Firstly, simple interpreters can be quickly derived and implemented from the denotational semantics of the DSL. Secondly, many parts of the host language can be directly reused: not only its syntax, but also some semantic features like its libraries and even its type system. Furthermore, it is easy to extend the DSL or compose and integrate several DSLs into the same host language, since DSL composition is the same as library composition.

Assume for example that we have three different DSLs: a language of regions, one of vectors, and a lambda calculus. We can simply compose these languages, if the concrete representations of their types in the host language match. Assuming Scala as our host language, we can then write a term like:

```
app(lam((x: Region) ⇒union(vr(x),univ)),
    scale(circle(3),
          add(vec(1,2),vec(3,4))))
```

This term applies a function which maps a region to its union with the universal region to a circle that is scaled by a vector.

However, the main advantage of this method is also its main disadvantage. It restricts the implementation to a fixed interpretation which has to be *compositional*, i. e., the meaning of an expression may depend only on the meaning of its sub-expressions, not on their syntactic form or some context. In the above example, we could perform two optimizations. First, the union of any region with the universal region is itself the universal region, allowing us to replace the body of the lambda abstraction with `univ`. And second, we see that the parameter `x` is not used (or without the first optimization: only used once) in the body, allowing us to inline the application [1], reducing the whole term to `univ`. Optimizations like these are hard to implement using compositional interpretations [2, 11].

In order to perform analyses on the code or write a more efficient implementation (or even a compiler) of the language, the DSL backend can be replaced by an explicit representation of the abstract syntax tree (AST) [8]. Then, arbitrary traversals over the AST can be implemented for analysis and interpretation. However, this flexibility comes at a price: extending the DSL with new operations, or even composing it with other languages, requires adaptation of both the data structure and the tree traversals.

Recently, variants of the DSEL approach have been proposed that introduce a separation between language interface and implementation [2, 6, 11]. In this way, it is possible to define several (compositional) interpretations for the same language, e. g., partial evaluators or transformations to continuation-passing style [6]. Furthermore, the languages can still be extended and even composed with other languages in the sense of extending and composing the *interpretations* [11].

However, two challenges remain. Firstly, can we express program transformations as independent modular components? This requires a program *representation* that is itself extensible and composable and can serve as the target domain of the transformation. Secondly, can we express *non-compositional* interpretations like the optimization mentioned above?

We believe that a scalable approach to embedding DSLs should address these problems. We propose a design that extends techniques developed in recent studies of the visitor pattern [5, 21] and the expression problem [34], and builds on our own work on polymorphic embedding [11]. More specifically, the design goals set forth in this paper are:

- The design should enable the composition of independently developed languages and their representations.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.
Copyright © 2010 ACM 978-1-4503-0154-1/10/10...$10.00

- Embedded languages are statically typed (uni-typed in the simple case such as the region DSL). A composed language should preserve the types of the individual languages.
- It should be possible to apply different (kinds of) interpretations on the same language representation.
  - The target domain of the interpretation should be allowed to be the term representation (program transformation). It should be possible to compose several program transformations in this way before interpreting to another domain.
  - It should be possible to define compositional as well as non-compositional interpretations.
- The language representations and their interpretations should be independent. That is, it should be possible to add new interpretations without having to change the language representation.

We choose Scala as the implementation language, as its combination of language features both allows for solving the expression problem and makes DSL embedding smooth [11, 34]. In particular, we make use of Scala's support for nested traits and mixin composition, abstract type members, higher-kinded types, self-type annotations, implicit conversions, type inference, and the flexible `import` statement. We will only show incomplete code examples for space reasons. Furthermore, we will not discuss infix operations in the paper. The complete source code with further examples can be downloaded at: `http://www.cs.au.dk/~chmh/mdslcs/`.

The central contributions of this paper are:

1. We show how to integrate extensible term representations into a DSEL approach in Scala. These term representations can be used as the target domain of program transformations on DSL terms.
2. We show that those term representations are composable in the same way as the languages that they represent. In particular, this composition also reflects the types of the DSL expressions.
3. Our representation accommodates for three kinds of interpretations: compositional interpretations, interpretations based on explicit AST traversal, and interpretations based on AST inspection. We motivate and compare these three options for writing interpretations.
4. We discuss name-binding on DSELs by means of composing with a lambda calculus language. In particular, we present an extensible term representation for the simply-typed lambda calculus using higher-order abstract syntax. This representation allows all three kinds of interpretations.
5. We present DSEL development as an application domain for advanced visitor techniques.

We will introduce the core of the representation for a single, uni-typed language in Sec. 2. In Sec. 3 we show how different languages (and their type representations) can be composed. In Sec. 4 we present language composition for a more challenging type system: the simply-typed lambda calculus. At the same time, we consider the issue of name-binding and its representation. In Sec. 5 we discuss the design goals and the different kinds of interpretations that our representation allows. Related work is discussed in Sec. 6. Section 7 concludes.

2. Presentation of the Core Design

In this section, we present the core design for a simple, uni-typed embedded language. We will first show how to define the language interface and compositional interpretations. Then we will introduce the term representation and an interpretation to create it. We demonstrate how the representations can be used to apply both compositional and non-compositional interpretations. We will use a language of regions [12] as the running example.

```
trait RegionLI {
  type Region
  def univ : Region
  def circle(radius : Double) : Region
  def union(reg1 : Region, reg2 : Region) : Region
}
trait Example {
  val regionInterpretation: RegionLI
  import regionInterpretation._
  ... union(circle(2.0), univ) ...
}
object ExampleInstance extends Module {
  val regionInterpretation = new EvalRegion {}
}
```

Figure 1. The region language interface and its usage

2.1 Defining the Language Interface

Each language specifies the language interface as the signature of an algebra: abstract type members declare the sorts (domains) of the algebra, methods its constants and operations [11]. The language interface of the regions language is shown in the trait `RegionLI` in Fig. 1. `Region` is the only sort in this algebra. `univ` is the universal region, `circle` is a circle around the origin and `union` is a binary operation to construct the union of two regions. More operations could be defined in the same way. We could also add them later by defining an extended language interface that inherits from `RegionLI`: The language interface is extensible.

To create a term of a language, we need an object that implements the language interface. However, it is easy to abstract over the actual interpretation such that the same DSL program can be interpreted in multiple ways. One way to do that is shown in the trait `Example`. Here, we specify a dependency of the example on some interpretation of the region language. The object `ExampleInstance` then fixes a specific interpretation. Note that the `import` construct in Scala can appear anywhere in the code and can refer to arbitrary values. We use it here to import the operations of the interpretation so we can use them without prefixing them by the name `regionInterpretation`.

2.2 Defining Compositional Interpretations

Each interpretation is an algebra of the corresponding signature. It is implemented by defining the domains and the operations that are declared in the language interface [11]. As a consequence of the algebraic method, each interpretation is guaranteed – under the assumption that we do not use side-effects – to be compositional in the following sense: The interpretation of an expression is only dependent on the interpretation of its sub-expressions, not on their syntactic structure or some context. This can be seen in the declaration of the `union` method: its parameters are of type `Region`, which is the domain of the algebra, and not of some expression type that represents region expressions. If we look at it as a traversal of the AST, each interpretation is a primitive recursion (*fold*) over the tree, applying the interpretation recursively on all sub-expressions. Compositionality of interpretations is the selling point of denotational semantics. It enables compositional reasoning about programs and it eases language extension.

To give an example of how an interpretation looks like, we define an evaluating interpretation of the region language in Fig. 2. The domain of regions is represented by a predicate on points in the coordinate space. The universal region is the region that contains all points, the union of a region is calculated by evaluating whether a point is contained in one of the united regions, etc.

```
trait EvalRegion extends RegionLI {
  type Region = (Double,Double)⇒Boolean
  def univ = (_,_) ⇒true
  def circle(rad: Double) = (x,y) ⇒x∗x + y∗y <= rad ∗ rad
  def union(r1: Region, r2: Region) = (x,y) ⇒r1(x,y) || r2(x,y)
}
```

Figure 2. A region evaluator written as a compositional interpretation

```
trait RegionAST {
  trait RExp {
    def acceptI[R](v : IVisitor[R]) : R
    def acceptE[R](v : EVisitor[R]) : R
  }
  case class Univ() extends RExp {
    def acceptI[R](v : IVisitor[R]) : R = v.univ
    def acceptE[R](v : EVisitor[R]) : R = v.univ
  }
  case class Circle(radius : Double) extends RExp {
    def acceptI[R](v : IVisitor[R]) : R = v.circle(radius)
    def acceptE[R](v : EVisitor[R]) : R = v.circle(radius)
  }
  case class Union(reg1: RExp, reg2: RExp) extends RExp {
    def acceptI[R](v : IVisitor[R]) : R =
      v.union(reg1.acceptI(v), reg2.acceptI(v))
    def acceptE[R](v : EVisitor[R]) : R = v.union(reg1, reg2)
  }
  type IVisitor[R] <:RegionLI { type Region = R }
  type EVisitor[R] <:RegionEVisitor[RExp,R]
}
trait RegionEVisitor[RExp,Region] {
  def univ : Region
  def circle(radius : Double) : Region
  def union(reg1 : RExp, reg2 : RExp) : Region
}
object RegionASTSealed extends RegionAST {
  type IVisitor[R] = RegionLI { type Region = R }
  type EVisitor[R] = RegionEVisitor[RExp, R]
}
```

Figure 3. A term representation for the region language

2.3 The Term Representation

In the next step, we define the explicit term representation. Its basic design is adapted with some minor modifications from the functional decomposition approach described in Zenger / Odersky [34], but implementing both the internal and the external visitor pattern [5, 19]. The representation of region terms is shown in Fig. 3 in the trait `RegionAST`.

For each sort in the signature of the algebra, we define an abstract syntax tree (AST) representation. In the example, the only sort is `Region`. For each operation that maps to a value of that domain, we implement an AST node as a *case class*[1] inheriting from a common super-node (`RExp`). That super-node declares the `acceptI` and the `acceptE` methods of the internal and external visitor pattern, respectively.

We declare higher-kinded abstract type members [16] for both the internal and the external visitor interface (`IVisitor[R]` and `EVisitor[R]`). Each sort of the algebra is a type parameter of this type member. To be able to extend the language with new operators, we do not fix these types [34]. Only in the object `RegionASTSealed` we create a concrete instance of the AST representation, where the visitor interface types are fixed. In `RegionAST` we only constrain them: The visitor interface of the internal visitor has to extend the language interface of the region language (see Fig. 1), with the type `Region` corresponding to the type parameter of the visitor. The visitor interface for the external visitor is shown in `RegionEVisitor`. There are two differences to the internal visitor interface: Firstly, we define the sorts (`Region`) as type parameters and not any more as an abstract type members, as we do not want to use the external visitor as a language interface. Secondly, and more importantly, the visitor takes another type parameter (or set of type parameters) that represents the expression types (`RExp`). This type parameter is used in the operations that take elements of the domains as parameters (here: `union`). In the external visitor interface, those operations take these expressions as parameters and not the domain elements.

This reflects the difference between internal and external visitor pattern. The internal visitor is applied to the sub-expressions before they are passed to the visitor of the expression (see the method `acceptI` in the class `Union`). This enforces compositional interpretations. The `acceptE` method of the external visitor, in contrast, does not perform a recursive call on the sub-expressions, but passes them directly to the visitor. In that way, the visitor has access to the syntactic structure of the sub-expressions. This makes non-compositional interpretations possible.

2.4 Program Transformations with Internal Visitors

Having defined the term representation, we can now define program transformations, i. e., interpretations that map to this term representation. These interpretations can then be composed with other interpretations by applying the `accept` method to the latter. For example, we can write an optimization interpretation (a program transformation) and compose it with the evaluator by supplying the latter as a visitor to the result of the former.

A trivial program transformation is the reification of the program. It takes a term and maps it to its representation. It is the identity element with respect to the composition of interpretations. The reification for the region language is shown in the trait `ReifyRegion` in Fig. 4. Being a compositional interpretation it implements the region language interface and can be used as an internal visitor. The trait is parametrized by a value `regAST` that references the exact instantiation of the AST representation. This parametrization is needed to accommodate for extensions of the region language with further operators. If we instantiate this value with an extended AST representation, reification operates as an injection into this richer structure. The type `Region`, which specifies the domain of the interpretation, is defined as the expression supertype in the chosen representation, making the interpretation a mapping into the term representation. The operations simply construct the corresponding AST nodes.

A more interesting program transformation is optimization. In our case, we define a simple optimization that makes use of an algebraic law on regions: that the union of some region with the universal region is equivalent to the universal region. The interpretation is shown in the trait `OptimizeRegion` and again is just a compositional interpretation (i. e. inheriting from the language interface). Again, the term representation it produces is parametrized by the value `regAST`. The interesting case is the implementation of union. Here, we use pattern-matching on the sub-expressions, i. e., we inspect the explicit AST representation. Note that the optimization has already been applied recursively to the parameters `reg1` and `reg2`. In that way, the optimization is propagated through the AST.

2.5 Program Transformations with External Visitors

We can define an alternative optimization using an external visitor, shown in Fig. 5. All interpretations using an external visitor depend on the language module, i. e., they have to be nested in another trait. Here, the trait `Optimize` is nested in `OptimizeRegionExternal`. The reason is that `acceptE` has to be called recursively on the optimization visitor. To be able to call `acceptE`, we have to make sure

[1] Scala case classes are basically classes suitable for pattern matching.

```
trait ReifyRegion extends RegionLI {
  val regAST : RegionAST
  import regAST._
  type Region = RExp
  def univ = Univ()
  ...
  def union(reg1 : Region, reg2 : Region) = Union(reg1, reg2)
}
trait OptimizeRegion extends RegionLI {
  val regAST: RegionAST
  import regAST._
  type Region = RExp
  def univ: Region = Univ()
  ...
  def union(reg1: Region,reg2: Region) : Region =
    (reg1, reg2) match {
      case (Univ(), _) ⇒Univ()
      case (_, Univ()) ⇒Univ()
      case _ ⇒Union(reg1, reg2)
    }
}
```

Figure 4. Two compositional program transformations

```
trait OptimizeRegionExternal {
  val regAST : RegionAST
  import regAST._
  trait Optimize extends RegionEVisitor[RExp, RExp] {
    this: EVisitor[RExp] ⇒
      type Region = RExp
      def univ : Region = Univ()
      ...
      def union(reg1: RExp, reg2: RExp): Region = {
        val r1 = reg1.acceptE(this)
        r1 match {
          case Univ() ⇒Univ()
          case _ ⇒{
            val r2 = reg2.acceptE(this)
            r2 match {
              case Univ() ⇒Univ()
              case _ ⇒Union(r1, r2)
            }}}}}}
```

Figure 5. An optimizer as an external visitor

that `Optimize` is in fact a valid visitor of type `EVisitor[RExp]`. We cannot guarantee this at this point, as the value of `regAST` and therefore the visitor interface is not yet fixed. But we can make the compiler ensure that each concrete instance of it has to be a valid visitor. This is done by declaring the type of `this` to be `EVisitor[RExp]` using Scala's self-type annotations in the first line of the body of `Optimize`. The visitor is defined in `regAST`. If we had declared this value inside of the `Optimize` trait instead, we would not be able to refer to it in the self-type annotation.

In the `union` case, we get the two sub-expressions as unevaluated expressions of type `RExp` as parameters. In that way, we can implement a more efficient version of the optimizer than before: We at first only optimize the first sub-expression recursively by calling `acceptE` on it. Only if this is not the universal region, we optimize the other sub-expression, too. We will discuss the respective advantages of compositional interpretations and explicit tree traversals in Sec. 5.

3. Composing Domain-Specific Languages

In this section, we will discuss how to compose the representations of the embedded languages. If we only ever want to compose several languages that share the same sort (i. e., in an untyped setting), this is similar to extending a language with new operations.

```
trait VectorLI {
  type Vector
  def vec(x : Double, y : Double) : Vector
  def add(v1 : Vector, v2 : Vector) : Vector
}
trait ExtRegionLI extends RegionLI {
  type Vector
  def scale(reg : Region, vec : Vector) : Region
}
```

Figure 6. Language interface for vector and extended region languages

As the term representations we are using have been written with this extensibility in mind [34], this is easy.

On the other hand, if we only want to introduce new sorts within a single language, we could make the different expression types share the same visitor, with the `accept` methods taking several type parameters (to reflect the different sorts) instead of one. The difficulty arises when we want to compose independent languages that each define their own sorts, as the `accept` methods cannot be extended by type parameters through inheritance.

In the following, we will discuss how the term representation can be made to work with language and sort composition. At the same time, this will show how individual languages can be extended with new operators. As a running example we will compose the region language with a simple language of vectors. Its language interface is defined in Fig. 6. We restrict ourselves to two operators: `vec` constructs a two-dimensional vector, `add` is the common vector addition. Again, we can implement different compositional interpretations for this language interface, e. g., an interpreter or a pretty-printer. Furthermore, we assume to have a term representation of the vector language defined.

The need to integrate region and vector language arises if we want to extend our region language with a new operator: `scale`, that scales a region by a vector. An example term of the composed language could be:

```
scale(circle(2.0), add(vec(1,2), vec(0,.5)))
```

We extend the language interface of the region language by inheriting from `RegionLI`, as shown in trait `ExtRegionLI`. Besides declaring the method `scale`, we declare an abstract type member `Vector` in addition to the inherited type member `Region`. We have shown how to compose language interfaces and their interpretations in this setting in earlier work [11]. Here, we focus on the corresponding composition of term representations.

3.1 Composing Term Representations

To compose the different term representations in a modular way, we create an interface between them. The term representation for the vector language is not modified. The representation for the extended region language is shown in the trait `ExtRegionAST` in Fig. 7. It abstracts over the vector representation `VectorRep`: We do not necessarily have to compose with an explicit term representation of vectors, but can alternatively use a direct representation. For example, we could choose to represent vectors as pairs of numbers, which could be the result of an evaluating interpretation using the `VectorLI` language interface.

The external visitor interface for the extended region language is shown in trait `ExtRegionEVisitor`. It takes the representation of vectors as an additional type parameter. Each non-compositional interpretation has the responsibility to deal with the respective representation of the other domain (here: vectors), be it an explicit term representation or a direct representation. The `acceptE` method in the class `Scale` is straightforward: It forwards the region expression and the vector representation to the visitor.

```
trait ExtRegionAST extends RegionAST {
  type VectorRep
  case class Scale(region : RExp, vector : VectorRep) extends RExp {
    def acceptE[R](v : EVisitor[R]) : R = v.scale(region, vector)
    def acceptI[R](v : IVisitor[R]) : R =
      v.scale(region.acceptI(v), v.interpretVector(vector))
  }
  type EVisitor[R] <:ExtRegionEVisitor[RExp, VectorRep, R]
  type IVisitor[R] <:ExtRegionIVisitor {
    type VRep = VectorRep
    type Region = R
  }
}
trait ExtRegionEVisitor[RExp, VRep, Region]
                                  extends RegionEVisitor[RExp, Region] {
  def scale(reg: RExp, vec: VRep): Region
}
trait ExtRegionIVisitor extends ExtRegionLI {
  type VRep
  def interpretVector(vec : VRep) : Vector
}
```

Figure 7. Extended language representation with visitor interfaces

For the internal visitor interface, we could take the same strategy and simply require the interpretations to directly deal with the term representations of the other domains. However, we believe that this is in conflict with the spirit of compositional interpretations. Interpretations should not operate on term representations, but only on the results of their interpretation. On the other hand, each interpretation has different constraints about the other domain. For example, an evaluator of the region language might expect to find vectors represented as pairs of `Doubles`, while a pretty printer might expect strings. It is therefore not possible to define the translation from a vector term representation to a vector domain representation outside of the visitor itself. That implies that the visitor has to supply a method `interpretVector` that does this interpretation. Otherwise, the internal visitor interface just extends the language interface. The code is shown in the trait `ExtRegionIVisitor`.

Now, the `acceptI` method in `Scale` is responsible for translating between the vector term representation and the vector domain. It delegates this to the extended visitor, and supplies the result to the `scale` operation (and potentially all the other operations that use the vector domain).

3.2 Defining Interpretations

In the following, we present how to implement interpretations using internal and external visitors. We will show how to implement the extension of the optimizer in both variants. We will also show how to implement an evaluator on the extended language using an internal visitor in order to demonstrate how distinct representations of the different languages can be combined.

We will start with the external visitors, as their implementation is more straightforward. The extended optimizer is shown in trait `OptimizeExtRegionExternal` in Fig. 8. It is independent of the representation of vectors and just reuses the one specified in the target domain `regAST`. The same holds also for the internal visitor which is shown in trait `OptimizeExtRegion`. It does not touch the representation of the vector language and thus defines `interpretVector` to be the identity function on the vector representation.

Finally, it is worth looking at the implementation of an evaluator for the extended region language, to see how `interpretVector` operates on different representations, and thus: how different representations of languages can be mixed. We present two variants of an evaluator in Fig. 9. Both make use of the same base evaluator defined for the extended region language `EvalExtRegion` that expects vectors to be represented as pairs of `Doubles`.

```
trait OptimizeExtRegionExternal extends OptimizeRegionExternal {
  val regAST: ExtRegionAST
  import regAST._
  trait Optimize extends super.Optimize with
        ExtRegionEVisitor[RExp, VectorRep, RExp] {
    this : EVisitor[RExp] ⇒
    def scale(reg: RExp, vec: VectorRep) = {
      val r = reg.acceptE(this)
      r match {
        case Univ() ⇒Univ()
        case _ ⇒Scale(r, vec)
      }}}}

trait OptimizeExtRegion extends OptimizeRegion with
      ExtRegionIVisitor {
  val regAST: ExtRegionAST
  import regAST._
  type VRep = VectorRep
  type Vector = VRep
  def interpretVector(v: VRep): Vector = v

  def scale(reg: Region, vec: Vector): Region =
    reg match {
      case Univ() ⇒Univ()
      case _ ⇒Scale(reg, vec)
    }
}
```

Figure 8. Two extensions of the optimizer as internal and external visitors, respectively

In the first version (`EvalRegionWithVector`), we do not use a term representation of vectors. Accordingly, `interpretVector` is implemented as the identity function on the vector domain. In the second version (`EvalRegionWithVecAST`), vector terms are represented. We can still reuse `EvalExtRegion`. The method `interpretVector` will apply a corresponding visitor for the vector language to get a value in the right domain. As the evaluator is now dependent on the specific term representation for vectors, the corresponding module `vecAST` is a dependency on the evaluation of regions. The object `EvalRegionWithVecASTSealed` is an example instantiation.

To conclude, we note that while the composition of explicit term representations increases the dependencies on the side of the interpretations, the main task with respect to language composition is the extension of the individual representations (here: the region representation) to accommodate for the new language constructs that bring together the two languages. As the chosen representations are extensible, language extension itself is straightforward. We will discuss a more advanced example of language composition in the next section.

4. Embedding the Lambda Calculus

Both the region and vector languages are uni-typed, so combining them resulted in a language of two types. However, the design also scales to more complex types in the embedded language. A prominent language with a more demanding type system is the simply-typed lambda calculus with its inductive construction of arrow types. We will therefore briefly sketch how the lambda calculus can be represented.[2] Introducing the lambda calculus serves also another purpose: as a showcase on how to handle name-binding in the embedded language.

In this section we will first introduce a language interface for the typed lambda calculus using higher-order abstract syntax (HOAS)

[2] The full code is in the accompanying code of the paper.

```
trait EvalExtRegion extends EvalRegion with ExtRegionLI {
  type Vector <:(Double,Double)
  def scale(r : Region, v : Vector) : Region =
    (x,y) ⇒r(x/v._1, y/v._2)
}
object EvalRegionWithVector extends EvalExtRegion with
    ExtRegionIVisitor {
  type Vector = (Double, Double)
  type VRep = (Double, Double)
  def interpretVector(v: VRep) = v
}
trait EvalRegionWithVecAST {
  val vecAST : VectorAST
  trait Eval extends EvalExtRegion with ExtRegionIVisitor {
    type Vector = (Double, Double)
    type VRep = vecAST.VectorExp
    def interpretVector(v : VRep) = v.acceptI(evalVector)
  }
  def evalVector: vecAST.IVisitor[(Double,Double)]
}
object EvalRegionWithVecASTSealed extends EvalRegionWithVecAST
    {
  val vecAST = VectorASTSealed
  object Eval extends super.Eval
  val evalVector = new EvalVector {}
}
```

Figure 9. Two evaluators based on the internal visitor pattern

```
trait THoasLI {
  type Rep[_]
  type VRep[_]
  def vr[T](x: VRep[T]): Rep[T]
  def lam[S,T](f: VRep[S]⇒Rep[T]): Rep[S⇒T]
  def app[S,T](fun: Rep[S⇒T], param: Rep[S]): Rep[T]
}
```

Figure 10. Language interface for the lambda calculus in higher-order abstract syntax

[23]. We will then show how to integrate it with the region language interface. Next, we will present an explicit term representation based on HOAS. Discussing the short-comings of this representation, we will motivate a De Bruijn index representation for the untyped lambda calculus, which we will briefly present.

The language interface is shown in Fig. 10. We use the type constructor `Rep[T]` to represent lambda calculus expressions of type `T`, and `VRep[T]` for variables of type `T`. Lambda calculus terms are either variables (constructed with `vr`), lambda abstractions (`lam`) or applications (`app`). Lambda abstractions make use of HOAS, i. e., we use function literals in Scala to represent lambda abstraction. An example term is: `lam((x: VRep[Int]) => vr(x))`, which represents the identity function on integers. The Scala type checker is not able to infer the type of the parameter `x`. Therefore, we have to specify it explicitly.

Some related works have proposed another representation that omits the `vr` constructor and the separate representation for variable types [2, 6]. That representation, however, does not give rise to a term representation [24]. Our representation can be regarded as a generalization of [33] for a typed representation.

4.1 A Term Representation for the Lambda Calculus

A term representation for the lambda calculus is shown in Fig. 11. Only the constructor for lambda abstractions is shown, the others are straightforward. The main point to note is that we need a different representation for each type of variable representation. Therefore, the latter has to be supplied as a type parameter to `THoasExp`. The second type parameter `T` is a type index for the corresponding lambda calculus expression. The represented domain type for

```
trait THoasAST {
  trait THoasExp[VR[_],T] {
    def acceptI[R[_]](v: IVisitor[R, VR]): R[T]
  }
  case class Lam[VR[_],S,T](f: VR[S]⇒THoasExp[VR,T])
                                        extends THoasExp[VR,S⇒T]
                                                  {
    def acceptI[R[_]](v: IVisitor[R, VR]): R[S⇒T] =
      v.lam(x ⇒f(x).acceptI(v))
    def acceptE[R[_]](v: EVisitor[R, VR]): R[S⇒T] = v.lam(x ⇒f(x))
  }
  ...
  type IVisitor[R[_], VR[_]] <:THoas {
    type Rep[T] = R[T]; type VRep[T] = VR[T]
  }
  type EVisitor[R[_], VR[_]] <:THoasEVisitor[THoasExp, R, VR]
}
trait THoasEVisitor[THExp[VR[_],_], R[_], VR[_]] {
  def vr[T](x: VR[T]): R[T]
  def lam[S,T](f: VR[S]⇒THExp[VR,T]): R[S⇒T]
  def app[S,T](fun: THExp[VR,S⇒T], param: THExp[VR,S]): R[T]
}
trait ReifyToTHoas[VR[_]] extends THoasLI {
  val hoasRep: THoasAST
  import hoasRep._
  type Rep[T] = THoasExp[VR,T]
  type VRep[T] = VR[T]
  ...
}
```

Figure 11. A term representation for the typed lambda calculus

an interpretation is `R[T]`. That means that `R[_]` is the type operator that describes the interpretation of a type and is therefore the higher-kinded type parameter of the `accept` methods. Note that reification (see trait `ReifyToTHoas`) is also always bound to a specific representation type for variables.

4.2 Integrating the Lambda Calculus with Other Languages

We can compose the lambda calculus with the region language and get regions as a base type in the lambda calculus and, on the other hand, the capability to use name binding in the region language. To this end, we extend both the region and the lambda calculus language interface, as shown in Fig. 12. We define implicit conversions (i. e., type conversions that will be inserted by the type-checker of Scala automatically) `toRegion` and `fromRegion` to translate between the different representations of region language and lambda calculus. The extended interface of the region language needs to know how a lambda calculus type is represented (`FunRep[T]`). In the same way, the lambda calculus interface needs knowledge about the representation of regions. The main restriction compared to the integration with the vector language is that we need an index type (i. e., a type parameter to `Rep`) to refer to the atomic region type in the lambda calculus representation. This cannot be `Region`, as the representation of functions has to be independent of a concrete interpretation domain of regions. That, however, requires that a region representation is not touched when it is transformed to a HOAS term: the parameter in `fromRegion` is not of type `Region` and we do not extend the visitor interface with a method `interpretRegion`. For symmetry, we also left out `interpretFunction`, making the interpretations themselves responsible for the interpretation step in the other domain. The corresponding extension of the term representation is straightforward and presented in the accompanying source code.

4.3 A Term Representation Based on De Bruijn Indices

Unfortunately, HOAS is not a good choice for programming interpretations that need to interpret recursively the body of a lambda

```
trait RegionForFunLI extends RegionLI {
  type FunRep[_]
  type RegionRep
  implicit def toRegion(r: FunRep[RegionRep]): Region
}
trait THoasForRegionLI extends THoasLI {
  type RegionRep
  implicit def fromRegion(r: RegionRep): Rep[RegionRep]
}
```

Figure 12. Composing lambda calculus and region language

```
trait LCLI {
  type Rep
  def vr(m: Int): Rep
  def lam(body: Rep): Rep
  def app(fun: Rep, param: Rep): Rep
}
trait LCEVisitor[LCExp, Rep] {
  def vr(m: Int): Rep
  def lam(body: LCExp): Rep
  def app(fun: LCExp, param: LCExp): Rep
}
trait LCAST {
  trait LCExp {
    def acceptI[T](v: IVisitor[T]): T
    def acceptE[T](v: EVisitor[T]): T
  }
  case class Vr(m: Int) ...
  case class Lam(body: LCExp) extends LCExp {
    def acceptI[T](v: IVisitor[T]): T = v.lam(body.acceptI(v))
    def acceptE[T](v: EVisitor[T]): T = v.lam(body)
  }
  case class App(fun: LCExp, param: LCExp) ...
  type IVisitor[T] <:LCLI { type Rep = T }
  type EVisitor[T] <:LCEVisitor[LCExp, T]
}
```

Figure 13. An untyped De Bruijn representation

abstraction, which is the case for many program transformations [29]. For this case, we propose a representation based on De Bruijn indices [7]. In this representation, variables do not have names, but are represented by an index that specifies in which binding it was defined. For example, the lambda expression $\lambda x.\lambda y.y$ is written as $\lambda(\lambda 0)$, with the variable index 0 referring to the variable of the innermost binding (y). On the other hand, $\lambda x.\lambda y.x$ is written as $\lambda(\lambda 1)$, where the 1 refers to the variable of the next-innermost binding (x).

Unfortunately, a typed De Bruijn index representation still has limitations. Atkey et al. [2] argue that the translation from HOAS to De Bruijn indices cannot be done fully type-safe, if the type-system of the language does not incorporate parametricity principles. What is worse: It is not obvious how to express relevant interpretations, e. g., substitutions, in a type-safe way. We will therefore represent only an untyped lambda calculus in the De Bruijn index representation, and it is up to the implementor to ensure type-safety of translations and interpretations. Still, the type-safety of embedded lambda calculus *terms* is preserved, if the `THoasLI` language interface is used.

The De Bruijn interface `LCLI` and its term representation are shown in Fig. 13. The reification of HOAS terms to De Bruijn index terms can be derived from [2] and is defined in the accompanying source code. For a demonstration that this representation allows for interesting program transformations we refer the reader to [2], where a shrinking reduction [1] is defined by an explicit traversal of the syntax tree. It is an interesting question, how much of it can be implemented using a compositional interpretation and we will come back to this in Sec. 5.

To conclude, it is possible to represent a typed lambda calculus using HOAS and compose it with other languages. However, for many program transformations it is not obvious how to implement them in this representation. Therefore, we follow [2] in performing these operations on an untyped representation based on De Bruijn indices.

5. Discussion

In this section, we will first review that the presented design indeed meets our design goals. In the second part, we will compare the different representations that are part of our encodings. Finally, we will briefly discuss alternative encodings for the visitor pattern.

5.1 Reviewing the Design Goals

First of all, the design allows for the composition of independently developed languages and their representations. We have demonstrated, how the representations of several languages can be composed in Sec. 3 and Sec. 4. We have furthermore demonstrated, how the interpretations compose even for distinct representations of the different languages in the evaluator example of Fig. 9.

We have seen that the composed language preserves the types of the individual languages. For example, the `scale` operation requires a region in the first parameter and a vector in the second. The implicit conversions between regions and lambda expressions ensure that only representations of regions can be converted.

Furthermore, we have demonstrated that different interpretations can be applied on the same language representation. We have shown, how we can define program transformations like the region optimization that transform to the same representation of the terms and can be composed with other interpretations. The representation can be used for defining compositional interpretations (using the internal visitor pattern) and non-compositional interpretations (using the external visitor pattern).

Finally, we have kept language representations and interpretations independent. This is a major difference to the Zenger/Odersky design [34]. We can seal a language representation as demonstrated, e. g., in `RegionASTSealed` in Fig. 3, and define interpretations like those in Fig. 4 independently from it, using dependency injection in the interpretations.

5.2 Comparing the Representations

So far, we have focussed the discussion on two different representations, namely external versus internal visitors. However, we are in fact dealing with four different representations.

1. The implicit term representation defined by the language interface.
2. The Church encoding expressed by the `acceptI` method of the internal visitor
3. The Scott encoding expressed by the `acceptE` method of the external visitor
4. The explicit AST representation that is part of both the external and the internal visitor pattern

In the following, we discuss each of these representations.

5.2.1 The Implicit Term Representation

The implicit term representation is defined by the language interface: Each operator of the DSL is represented by a method declaration in the language interface and each term is at some point mapped to a concrete interpretation to a target domain. However, this representation cannot be used as a target domain of an interpretation by itself. If we want to define a program transformation, we immediately have to compose it with an interpretation to another

```
object UsesOf extends LCLI {
  type Rep = Int ⇒Int
  def vr(m: Int): Rep = n ⇒if (n == m) 1 else 0
  def lam(body: Int⇒Int): Int⇒Int = n ⇒body(n+1)
  def app(fun: Int⇒Int, param: Int⇒Int): Int⇒Int =
    n ⇒fun(n)+param(n)
}
```

Figure 14. Counting the occurrences of a lambda expression

target domain. As a consequence, the transformation of an expression has no access to the transformation of its sub-expressions, but only to the results of their final interpretation. To overcome this, we can define the target domain to be a pair, where the first component is the intended interpretation and the second component is some information that we need for performing the program transformation. We have demonstrated this for the optimization of regions in [11], where the second component was a Boolean flag that informed us, if a region was the universal region. We then could shortcut the intended interpretation in the first component, whenever the Boolean flag was `true`.

5.2.2 The Church Encoding

It is known that the internal visitor pattern corresponds to a Church encoding [5]. The Church encoding, as well as the standard visitor pattern, are not by themselves extensible. An extensible solution has to allow for adding domains and operations to the language interface. The presented design does exactly that.

Defining an interpretation using the Church encoding makes it compositional. While compositionality is certainly beneficial for reasoning about a DSL term, not every interpretation can directly be encoded in this style. However, for many non-compositional interpretations to a domain we can find a compositional interpretation to a computation of that domain. This is the core idea behind using monads to define modular denotational semantics [15].

For example, the optimization interpretation in Fig. 4 is not optimal: in the `union` case, if the first region is the universal region, then we could short-circuit the interpretation of the second region, as the result will be the universal region. However, as we defined the parameters call-by-value, the interpretation of the second region has already taken place. To avoid this, we could redefine the language interface to take the parameters of `union` as call-by-need parameters. However, if we do not want to change the language interface, we could redefine the domain of the optimization interpretation to be a function from the unit type to the AST representation. In that way, we can manually control the triggering of the optimization in the sub-expressions.

Another example would be a language of arithmetics, where we cannot implement a compositional evaluator to a domain of numbers that handles division-by-zero, but we could implement a compositional evaluator to a domain of computations that can fail (described by the error monad).

And finally, there are many interpretations that depend on a context. One example is the interpretation that counts the occurrences of a free variable in a De Bruijn index representation of a lambda expression. Atkey et al. [2] define this interpretation as a recursion on an explicit AST representation. But we can also express it as a fold, as shown in Fig. 14. We represent the domain as a function that maps a De Bruijn index to the number of occurrences of the variable with this index. The De Bruijn index is the context that is passed through the interpretation of the sub-expressions and is increased inside a lambda body.

Another limitation of the Church encoding is that it is hard to define accessor functions to the sub-expressions of an expression. We encounter this problem, if we translate the shrinking reduction implementation from [2] to one based on internal visitors, as shown

```
object Shrink extends LCLI {
  import LCIASTSealed._
  type Rep = LCExp
  def vr(n: Int): Rep = Vr(n)
  def lam(body: Rep): Rep = Lam(body)
  def app(fun: Rep, param: Rep): Rep = fun match {
    case Lam(u) ⇒if (u.accept(UsesOf)(0) <= 1)
                       u.accept(Subst)(0, param).accept(this)
                     else App(fun,param)
    case _ ⇒App(fun, param)
  }
}
```

Figure 15. Shrinking reduction

in Fig. 15. The shrinking reduction [1] is an inlining operation that performs a beta-reduction in cases where a bound variable is used at most once.

For simplicity, this interpretation is hard-wired to some sealed version of the AST representation for internal visitors.[3] Furthermore, it assumes a substitution interpretation (`Subst`). It also does not claim to be an efficient implementation.

The interesting part is the interpretation of `app`: It does a pattern-matching on the interpretation of the first parameter. If it is a lambda expression, it might perform a substitution to inline the application. If we wanted to avoid using pattern-matching, we would need an accessor to the body of the lambda abstraction.

The solution to this problem was discovered by Kleene, who defined the predecessor function on Church numerals by a triple construction [13] together with a projection to the first component of the triple. This trick can be generalized to arbitrary accessors on inductive data structures. However, using this method, the sub-expression has to be fully reconstructed from bottom up, making accessors a linear-time instead of a constant-time operation. It is also not obvious how to adapt Kleene's trick to access the body of a lambda expression in a higher-order abstract syntax representation.

5.2.3 The Scott Encoding

Like the internal visitor pattern corresponds to a Church encoding, Oliveira et al. [21] have pointed out that the external visitor pattern corresponds to a Parigot encoding [22], which is a typed version of the Scott encoding. Like the Church encoding, the Scott encoding itself is not extensible.

The Scott encoding makes it easy to define an accessor operation on inductive data structures. In effect, that means that we can implement everything with the Scott encoding that we can implement by pattern matching. The interpretations are not guaranteed to be compositional. Using the presented version of the external visitor pattern has one core advantage over directly accessing the AST representation via pattern-matching: The interpretation stays extensible. If we extend a language by a new operation, we simply have to define the interpretation for this operation. If we had used pattern-matching instead, the interpretation for the extended language version would have to override the original interpretation in order to take the extended cases into consideration.

5.2.4 The Explicit AST Representation

After this discussion, there seems to be no place where the explicit AST representation is really needed. We used it in many examples, nevertheless, however not as an alternative encoding, but inside the internal and external visitors. The representation is useful, when we want to analyze the structure of the sub-expressions, typically after applying a code transformation on them. Writing another interpretation that does this analysis is in many cases cumbersome,

[3] This could of course include the conversion from region language terms.

when a pattern matching is so much easier. However, it should be kept in mind that this could conflict with the extensibility of the interpretation which has to rely on correct defaults (see also [35]).

5.3 Alternative Encodings for the Visitor Pattern

While the visitor pattern in its basic version does not accommodate well for extending data types, there are several approaches to make the visitor pattern extensible and in that way give a solution to the expression problem [19, 30, 34]. We have adapted the solution presented in [34] to get an extensible visitor pattern as the basis of our design. We have modified it to separate representation from visitors and to make use of higher-kinded type members [16]. Furthermore, we have implemented an internal visitor pattern variant for it.

In Sec. 5.5 of [18], Oliveira presents a very similar design for an extensible visitor pattern in the context of data type generic programming. Instead of defining two different `accept` methods for external and internal visitors, this design merges both by using an additional type parameter and an additional implicit parameter. In that sense, it trades simplicity for genericity, but that choice is orthogonal to our design goals.

Oliveira's design is most similar to our representation of the lambda calculus in that it represents types uniformly by applying an abstract type constructor to them as we apply `Rep` to the types in the lambda calculus language. As a consequence, each interpretation instantiates this type constructor which is then uniformly applied to all represented types. This uniform representation, however, prevents mapping different domains like regions and vectors to different representation types.

Common to both designs is the problem that combining several extensions of a language requires combining the visitors explicitly [34]. An interesting alternative encoding of visitors [19] overcomes this limitation and allows constructing internal and external visitors in a very customizable way. This gives the user fine-grained control over which language operators to include. A core advantage of this approach is that it renders the dependency injections that inform each interpretation about the exact language used (see, e. g., value `regAST` in Fig. 4) unnecessary. Instead, an interpretation of an extended language can always be used as an interpretation for a more restricted language.

However, if we want to avoid dependency injection even when composing languages, the visitors and the constructors of the overarching language constructs have to take more type parameters. To integrate these visitors with the original language components, we had to curry the type parameters of the visitors by using Scala's encoding of anonymous type functions [16]. As a result, the type parameters got very cluttered and Scala's type checker was clearly pushed to its limits. However, if anonymous type functions get direct support in Scala, this might be the preferable approach.

Finally, it would be interesting to see if an analogous design can be expressed in a functional programming language. Oliveira et al. [20] describe a solution to the expression problem in the field of generic programming which may be a promising starting point for a design using Haskell.

6. Related Work

Espinosa [9] presented an (untyped) design for denotational semantics where the language interface is decoupled from the (compositional) interpretations. The interpretations Espinosa uses are implemented using monads and monad transformers, making them extensible with respect to different kinds of computational capabilities. Term representations as the domain of an interpretation are not considered.

Carette et al. [6] have been using a Church-like encoding for the lambda calculus based on a typed version of [14]. They mainly focus on a MetaOCaml implementation. In this implementation, the terms themselves are not written in an explicit Church encoding in the sense of lambda abstracting over the interpretation of the `lam` and `app` terms, but instead they are encapsulated in functors that provide this abstraction. This prevents using a Church encoding as the target domain of an interpretation, although in their type-class-based Haskell implementation this would be possible. However, the encoding allows applying different interpretations with different target domains. For defining program transformations like partial evaluation, they use quoted MetaOCaml terms. In Haskell, they use an explicit AST representation. They do not discuss extension or composition of languages, but restrict themselves to the presentation of a lambda calculus with a fixed set of arithmetics and Boolean operations.

In our own previous work [11], we have used an approach similar to [6] to representing terms in Scala that allows for easy composition of languages and interpretations. The definition of language interfaces as traits that declare the signature of an algebra was developed there. However, we only used the implicit term representation and did not consider the possibility to use a Church encoding of the target domain. As a consequence, program transformations like the optimization of regions could only be expressed by coupling them with an interpretation to another target domain, as has been discussed in Sec. 5.

Atkey et al. [2] adapt the type-class based representation of [6] written in Haskell and present a typed and an untyped variant of it. They show that this representation is extensible and composable. On the other hand, they argue that an *unembedding* to an explicit data structure representation of ASTs using De Bruijn indices is necessary for some interpretations. This AST representation, however, is – in contrast to our representation – neither extensible nor composable.

There is plenty of literature on using higher-order abstract syntax to represent the lambda calculus in a host language beyond the one already mentioned (recent articles are [10, 17, 25, 27, 29, 33]). Many of them use HOAS on an explicit data structure representation and discuss issues like adequacy of the representation that are beyond of the scope of this paper. Our encoding of HOAS has been inspired by the untyped variant discussed in [33].

Stump [29] introduces a new meta-programming language, Archon, based on the untyped lambda calculus, but extended with direct support for structural reflection, using HOAS to represent lambda abstraction. In contrast to approaches built on top of explicit encodings of the object language, Archon introduces explicit language constructs for opening of lambda expressions along with other language constructs for working on variables and overcomes in this way the restrictions of HOAS representations.

Buchlovsky / Thieleke [5] have analyzed the type theory of the visitor pattern. They observed the difference between the internal visitor pattern and the external visitor pattern and elaborated the correspondence of the former to the Böhm-Berarducci encoding [3]. Oliveira et al. [21] observed that the external visitors correspond to the Parigot encoding [22]. These correspondences make the visitor pattern an ideal candidate to define compositional and non-compositional interpretations on an AST representation in object-oriented languages.

Keeping language representations and their interpretations as extensible components has been the eponymous example for the discussion of the expression problem [32], i.e., the problem of extending data types and operations on them independently. We have adapted and extended the design from [34]. Our work has a different focus, though. The expression problem is about incrementally extending individual languages, not about composing independently developed languages and their representations. More importantly, while the expression problem has been described for untyped expressions, our design had to accommodate for a typed setting.

Finally, there are other approaches to implement embedded languages that use external tools to integrate the embedded language into the host language. Examples are the attribute grammar based approach of ableJ [31] and the term rewriting approach of MetaBorg using Stratego/XT [4]. On the other hand, Kiama [28] is a project to integrate those language processing tools as Scala DSELs for code generation from external DSLs. It may be worthwhile inquiring whether the ideas of Kiama can be merged with our design in order to get language tools as modular DSEL components to process DSELs.

7. Conclusion

We have presented a design for integrating extensible term representations into a typed DSEL approach. We showed how to use these term representations as target domains for program transformations on DSL terms and as starting points for writing non-compositional interpretations. Furthermore, we demonstrated how several DSELs can be composed in a type-preserving way. We discussed name-binding by introducing the lambda calculus as a DSEL, together with two representations: a typed HOAS-based and an untyped De-Bruijn-index-based representation. Finally we have discussed, how the presented design accommodates for three kinds of interpretations: compositional interpretations, interpretations based on explicit AST traversal and interpretations based on AST inspection. We have compared the advantages and disadvantages of these three styles of interpretation. In the future, we want to further investigate typed lambda calculus representations and the limits of representability in a DSEL approach.

Acknowledgments

The authors would like to thank Adriaan Moors, Michael Achenbach, and the anonymous reviewers for their insightful comments and suggestions that helped improve the presentation of the paper.

References

[1] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[2] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, pages 37–48, 2009. ACM.

[3] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[4] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04*, pages 365–383, 2004. ACM.

[5] P. Buchlovsky and H. Thielecke. A type-theoretic reconstruction of the visitor pattern. In *21st Annual Conference on Mathematical Foundations of Programming Semantics*, ENTCS 155, pages 309–329, 2006.

[6] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[7] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[8] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[9] D. A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, New York, NY, 1995.

[10] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *POPL '96*, pages 284–294, 1996. ACM.

[11] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE '08*, pages 137–148, 2008. ACM.

[12] P. Hudak. Modular domain specific languages and tools. In *ICSR '98*, pages 134–142. IEEE Computer Society, 1998.

[13] S. C. Kleene. A theory of positive integers in formal logic. part i. *American Journal of Mathematics*, 57(1):153–173, 1935.

[14] T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 39–44, 1995. ACM.

[15] E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science '91*, LNCS 530, pages 138–139, 1991. Springer.

[16] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *OOPSLA '08*, pages 423–438, 2008. ACM.

[17] A. Nanevski. Meta-programming with names and necessity. In *ICFP '02*, pages 206–217, 2002. ACM.

[18] B. C. d. S. Oliveira. *Genericity, extensibility and type-safety in the Visitor pattern*. PhD thesis, Oxford University Computing Laboratory, Oxford, England, 2007.

[19] B. C. d. S. Oliveira. Modular visitor components. In *ECOOP '09*, LNCS 5653, pages 269–293, 2009. Springer.

[20] B. C. d. S. Oliveira, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming, Volume 7*, pages 199–216, 2006. Intellect Books.

[21] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08*, pages 439–456, 2008. ACM.

[22] M. Parigot. Recursive programming with proofs. *Theor. Comput. Sci.*, 94(2):335–336, 1992.

[23] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI '88*, pages 199–208, 1988. ACM.

[24] F. Pfenning and P. Lee. Metacircularity in the polymorphic lambda-calculus. *Theor. Comput. Sci.*, 89(1):137–159, 1991.

[25] T. Rendel, K. Ostermann, and C. Hofer. Typed self-representation. In *PLDI '09*, pages 293–303, 2009. ACM.

[26] J. C. Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages, IFIP Working Group 2.1 on Algol*. INRIA, 1975.

[27] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2):1–57, 2001.

[28] A. M. Sloane. Experiences with domain-specific language embedding in scala. In *2nd International Workshop on Domain-Specific Program Development*, Oct. 2008.

[29] A. Stump. Directly reflective meta-programming. *Higher-Order and Symbolic Computation*, 2010. To appear.

[30] M. Torgersen. The expression problem revisited. In *ECOOP '04*, LNCS 3086, pages 123–143, 2004. Springer.

[31] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *ECOOP '07*, LNCS 4609, pages 575–599, 2007. Springer.

[32] P. Wadler. Expression problem, Java Genericity Mailing List, 12 November 1998. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[33] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(1):87–140, 2008.

[34] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *12th International Workshop on Foundations of Object-Oriented Languages*, 2005.

[35] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *ICFP'01*, pages 241–252, 2001. ACM.