Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Scala-Virtualized: Linguistic Reuse for Deep Embeddings

Tiark Rompf · Nada Amin · Adriaan Moors · Philipp Haller · Martin Odersky

Received: date / Accepted: date

Abstract Scala-Virtualized extends the Scala language to better support hosting embedded DSLs. Scala is an expressive language that provides a flexible syntax, type-level computation using implicits, and other features that facilitate the development of embedded DSLs. However, many of these features work well only for shallow embeddings, i.e. DSLs which are implemented as plain libraries. Shallow embeddings automatically profit from features of the host language through linguistic reuse: any DSL expression is just as a regular Scala expression. But in many cases, directly executing DSL programs within the host language is not enough and deep embeddings are needed, which reify DSL programs into a data structure representation that can be analyzed, optimized, or further translated. For deep embeddings, linguistic reuse is no longer automatic.

Scala-Virtualized defines many of the language's built-in constructs as method calls, which enables DSLs to redefine the built-in semantics using familiar language mechanisms like overloading and overriding. This in turn enables an easier progression from shallow to deep embeddings, as core language constructs such as conditionals or pattern matching can be redefined to build a reified representation of the operation itself.

While this facility brings shallow, syntactic, reuse to deep embeddings, we also present examples of what we call deep linguistic reuse: combining shallow and deep components in a single DSL in such a way that certain features are fully implemented in the shallow embedding part and do not need to be reified at the deep embedding level.

Keywords Code generation · Domain-specific languages · Linguistic reuse · Language virtualization

This research was sponsored by the European Research Council (ERC) under grant 587327 "DOPPLER"

École Polytechnique Fédérale de Lausanne (EPFL)
EPFL IC IFF LAMP
Station 14
1015 Lausanne
E-mail: first.last@epfl.ch

1 Introduction

The necessity of general purpose languages to serve as meta languages for hosting embedded languages or domain-specific extensions is widely recognized [27, 51, 21, 22]. However, when it comes to assessing the meta language capabilities of existing languages, the consensus seems to be that "Our current languages are OK but not great" [48]. Are there some simple tweaks we can apply to our current languages to improve their DSL capabilities?

Embedded language developers need to balance diverse criteria such as expressiveness, performance, safety and implementation effort. The main difficulty in building embedded languages is to select suitable host language mappings, as different implementation strategies come with very different trade-offs. For example, a shallow embedding that implements the embedded language as a host language library profits greatly from linguistic reuse [26]: it is easy to implement and blends well with the host language ecosystem. The downside is that it will be limited in the achievable performance. By contrast, a deep embedding that reifies all expressions as host-language data structures is amenable to symbolic analysis, optimization and code generation, but also significantly harder to build and quite likely less pleasant to use and debug because of syntactic and semantic gaps between the host language and the embedded language.

The concept of language virtualization [9] is an attempt to capture the goal of reducing this trade-off in an analogy to the hardware world: Like in a data center, where one wants to virtualize costly "big iron" server resources and run many logical machines on top of them, it is desirable to leverage the engineering effort that went into a general-purpose language to support many small embedded languages, each of which should behave more or less like a real language, ideally according to all the different criteria given above.

Hardware virtualization solutions that are correct, safe, and performant, can be realized by intercepting privileged processor instructions and direct communication with hardware resources, overriding the default implementation with one that redirects to a virtualized resource manager [37]. Can we draw an analogy for programming languages again? We can think of "virtualizing" a host language feature by adding a facility to intercept this construct and reinterpret it in the context of a DSL. In that sense, many modern languages support virtualization of a certain set of features. For example, we could say that C++ has virtualized infix operators because classes are free to add their own implementations for the given set of operators. Similarly, we could say that Haskell has virtualized statement composition because its 'do' notation can be used with any monad.

This ability to customize otherwise built-in language features is very powerful. In C++, operator overloading is a key enabler of expression templates [61]; in Haskell, all kinds of powerful abstractions are implemented using monads and 'do'-notation. However, most programming languages (maybe with the exception of Lisp and Scheme) come with a certain, limited set of features they allow programmers to customize. Only rarely is it possible to redefine the meaning of core constructs like conditionals and pattern matching. If this mechanism is so powerful, then the question bears asking: Why stop half way through?

In this paper, we present Scala-Virtualized, the result of our efforts to further virtualize the Scala language and extend its DSL hosting capabilities. Scala is a mature language that is seeing widespread adoption in industry. Scala-Virtualized is a suite

of binary-compatible minimal extensions, based on the same codebase and undergoing the same rigorous testing as the main Scala distribution. Scala-Virtualized tries to take the idea of redefining language features as method calls all the way. Hence, there is an immediate correspondence between virtualized language features and their implementation using virtual (overloadable and overridable) methods in the object-oriented sense.

A key use for virtualization in Scala is to achieve reification of program expressions with a shallow embedding: On the surface, a DSL program consists just of method calls, but behind the scenes, this shallow embedding may create a data type representation that can be further analyzed, optimized and translated. We thus make the key benefit of shallow embeddings, linguistic reuse of host language constructs, in a systematic way available for deep embeddings.

Moreover, the shallow embedding can perform non-trivial computation while building up the data structures that constitute the deep embedding, a setup reminiscent of multi-stage programming [56]. The main benefit of such a staged embedding is to keep the deep embedding part simple. For example, many embedded languages rely on closures, higher-order functions or generic types, but often, it is possible to resolve these abstractions within the shallowly embedded parts. In this case, we speak of *deep* linguistic reuse: An embedded language with a deep embedding component reuses a host language feature without implementing dedicated support on the deep embedding level.

A number of DSL projects are using Scala-Virtualized and associated techniques such as Lightweight Modular Staging (LMS) [43], resulting in embedded DSL programs that outperform handwritten C code [52, 45], that compile to GPUs [6, 28], that query databases using generated SQL [62], or that generate JavaScript for running in web browsers [25].

The purpose of this paper is twofold. The first aim is to serve as a reference for DSL developers in Scala. The second aim is to describe the techniques we applied to Scala in such a way as to make them accessible to other language implementors.

1.1 Organization

The rest of this paper is structured as follows.

Section 2 provides an introduction to DSLs in Scala. Section 2.1 introduces a small example DSL for probabilistic programming. This section walks through an initial shallow embedding, a shallow monadic embedding, and finally a deep embedding, discussing the different aspects of each; for example the ability to symbolically reason about probability distribution in the deep embedding. As we go along, we highlight the required changes to DSL programs in plain Scala and features of Scala-Virtualized which facilitates a smoother progression from shallow to deep embeddings—without significant changes to the original DSL program.

Section 3 presents the overall methodology of defining language constructs as method calls and discusses the virtualization of control structures (Section 3.1) and method calls (Section 3.2).

Section 4 describes virtualized pattern matching. It discusses both reification of pattern match expressions (Section 4.1) and replacing the default option monad with a probabilistic monad, yielding an interesting embedding of a rule-based logic programming DSL in Scala (Section 4.2).

Section 5 introduces virtualized record types, as well as record construction and access. The example here is an embedding of SQL-like query expressions (Section 5.1).

Section 6 is about integrative concerns. Section 6.1 discusses statements and bindings. Section 6.4 presents DSL scopes, a syntactically lightweight mechanism to interface DSL programs with the DSL definition. Examples comprise an embedding of JavaScript (Section 6.2) and staged array computations (Section 6.5).

Section 7 is about DSL debugging support. It discusses lifting static information about types and source locations (Section 7.2), and producing DSL specific error messages (Section 7.3).

Section 8 discusses related work and Section 9 concludes.

An earlier version of this paper appeared as a "tool demo" short paper at PEPM'12 [31]. The present version adds much additional material, for example on virtualized pattern matching (Section 4) and DSL scopes (Section 6.4), as well as many more examples, including on probabilistic (Section 2.1) and logic programming (Section 4.2). The accompanying source code is available online[1].

2 Scala and Embedded DSLs

Scala [33] is a general-purpose language that originated in academia, but is seeing widespread adoption in industry and open-source projects. One of Scala's success factors is its tight interoperability with the Java and JVM ecosystem. A second factor is that Scala combines features from the object-oriented and functional programming paradigms in new and interesting ways.

Since its early days, Scala has been used successfully as an environment for pure library-based DSLs, such as parser combinators [30], actors [19] and testing frameworks. DSLs typically leverage the following three axes of flexibility that Scala provides:

- Exploiting flexible syntax. Parser combinators mimic BNF syntax, actor DSLs provide concise message passing syntax similar to Erlang [2], and testing frameworks such as Specs [59] express executable test cases similar to human-readable specifications:

  ```
  // generate 500 different mail addresses
  mailAddresses must pass { address =>
    address must be matching(companyPattern)
  }
  ```

  Although Scala does not provide real, arbitrarily extensible syntax (such as Racket [58], for example), Scala can still be seen as an extensible language. In many cases libraries can be made to look like built-ins. Conceptually, every value is an object in Scala, and every operation a method call. Method call syntax can either follow the traditional `receiver.method(argument)` pattern, or use the infix invocation syntax `receiver method argument` that is used pervasively in the snippet above.
- Redefining the run-time semantics of for-comprehensions. An expression such as

  ```
  for (i <- foo) yield 2 * i
  ```

  is desugared to the following method call:

  ```
  foo.map(i => 2 * i)
  ```

[1] https://github.com/tiarkrompf/scala-virtualized

The class of `foo` defines the implementation and type signature of the methods (such as `map` and `flatMap`) that define the semantics of a for-comprehension. For comprehensions fit very well with monadic encodings and can be seen as the Scala analogue of Haskell's 'do'-notation.
- Customizing the type system and reifying types at run time. Domain-specific typing rules can be implemented using implicit resolution, which provides a limited form of logic programming at the type level [11], comparable to type classes in Haskell [64]. Phantom types and other classic tricks are also commonly used. Finally, manifests [13] can be used to reify types at run time, e.g., to aid type-directed code generation.

For DSLs as pure libraries, these features are often all that is needed. But when non-standard semantics are needed, when better performance is required, when additional program properties must be verified for safety, or when code is to be generated for different platforms, then the situation is more complicated. In many cases, a deep embedding, i.e., an accessible representation of embedded programs that can be analyzed and transformed, is called for. Section 2.1 highlights the issues that arise when moving from a shallow to a deep embedding in Scala by way of an example.

To better support DSLs that need to go beyond simple shallow embeddings, Scala-Virtualized extends the ideas from regular Scala as follows:

- Not only for-comprehensions are expressed in terms of methods calls, but many other control structures as well: conditionals, while loops and variable assignment (Section 3.1), pattern matching (Section 4), certain kinds of object construction (Section 5).
- Infix functions provide additional syntactic freedom, and enable overriding existing method implementations in a lexically-scoped way (Section 3.2).
- DSL scopes enable access to customized DSL method implementations within a lexical block (Section 6.4).
- Implicit source contexts supply static source information at run time (Section 7.2). Static source information can be used to improve DSL debugging and runtime error messages.

We present a walk-through of a small example DSL implementation in Section 2.1 and continue with a more in-depth description of particular features in Section 3 and beyond.

2.1 Example: A Probabilistic Programming DSL

We discuss embedding a small language within Scala, contrasting three implementations, from very shallow to deep. With Scala-Virtualized, the deeper embeddings can maintain the illusion of a very shallow embedding which is easy to use and well-integrated in the host language.

Our running example is a DSL for probabilistic programming that sports a probabilistic choice operator. Probabilistic choice can be used, for example, to simulate rolling of a die:

```
val die = choice(1 -> 1/6, 2 -> 1/6, 3 -> 1/6, 4 -> 1/6, 5 -> 1/6, 6 -> 1/6)
```

Using this basic choice operator, it is straightforward to define a number of higher-level operators:

```
def uniform[A](xs: A*)   = choice(xs.map((_,1.0)):_*)
def flip(p: Prob)        = choice(true -> p, false -> (1-p))
def always[A](x: A)      = choice(x -> 1.0)
def never                = choice()
def rand(a: Int, b: Int) = uniform((a to b):_*)
```

where `Prob` is just a type alias for `Double`. Rolling a die or throwing a fair coin can now be expressed as:

```
val die  = rand(1,6)
val coin = flip(0.5)
```

2.1.1 A Pure, Shallow Embedding

In the simplest and shallowest possible embedding, the probabilistic choice operator returns a random sample, computed according to the distribution. A probabilistic model, such as the traffic example presented in Figure 1, is run many times to obtain an approximate posterior distribution. This approximate probabilistic inference can be achieved by a small piece of driver code, which takes the model argument as a by-name parameter (indicated by the arrow `=>` in the "`model: => A`" notation):

```
type Dist[A] = List[(A,Prob)]
def evaluate[A](model: => A): Dist[A] = {
  // run model 5000 times, group by result and sum up probabilities
}
```

Despite its obvious drawbacks, a key benefit of this very shallow embedding is linguistic reuse! The result of a choice over an integer distribution is just an integer, so no lifting is necessary. As we move to deeper embeddings, we would like to keep the intuitive and lightweight feel of this very shallow embedding.

2.1.2 A Shallow Monadic Embedding

What if we want to perform exact inference? One approach, exemplified by the language Hansei [24], embedded in OCaml, uses continuations to explore more than one path: the embedding remains very shallow (e.g. a roll of a die still has type `Int`), but the choice operator now has a control effect. Here, we explicitly model the paths by introducing a type constructor `Rand` so that rolling a die has type `Rand[Int]`.

We first define some data types to model search trees:

```
abstract class Path[+A] { def x: A; def append[B](that: Path[B]): Path[B] }
case object Root extends Path[Nothing]
case class Choice[+A](id: Int, x: A, p: Prob, parent: Path[Any]) extends Path[A]
type Model[+A] = List[Path[A]]
```

A model is a list of paths, and a path a list of choices. This is not a very efficient representation but it will do for our purposes.

The `Rand` monad just wraps a model:

```
case class Rand[+A](model: Model[A]) {
  def map[B](f: A => B): Rand[B]              = flatMap(x => always(f(x)))
  def orElse[B >: A](that: Rand[B]): Rand[B]  = Rand(model ++ that.model)
  def flatMap[B](f: A => Rand[B]): Rand[B]    =
    Rand(model.flatMap(path => f(path.x).model.map(post => path.append(post))))
```

```
// 1. Model traffic light
abstract class Light
case object Red extends Light
case object Yellow extends Light
case object Green extends Light
val trafficLight = choice(
  Red -> 0.5, Yellow -> 0.1, Green -> 0.4)

// 3. Model crash
abstract class Result
case object Crash extends Result
case object NoCrash extends Result

def otherLight(light: Light) = light match {
  case Red => Green
  case Yellow => Red
  case Green => Red
}
```

```
// 2. Model drivers
abstract class Action
case object Stop extends Action
case object Drive extends Action
type Driver = Light => Action

def cautiousDriver(light: Light) = light match {
  case Red    => always(Stop)
  case Yellow => choice(Stop -> 0.9, Drive -> 0.1)
  case Green  => always(Drive)
}
def aggressiveDriver(light: Light) = light match {
  case Red    => choice(Stop -> 0.9, Drive -> 0.1)
  case Yellow => choice(Stop -> 0.1, Drive -> 0.9)
  case Green  => always(Drive)
}
```

```
def crash(driver1: Driver, driver2: Driver, light: Light): Result = {
 if (driver1(light) == Drive && driver2(otherLight(light)) == Drive)
   choice(Crash -> 0.9, NoCrash -> 0.1)
 else
   NoCrash
}

// 4. Model different scenarios
val trafficModel1 = crash(cautiousDriver, aggressiveDriver, trafficLight)
val trafficModel2 = crash(aggressiveDriver, aggressiveDriver, trafficLight)
```

Fig. 1 Traffic Modelling (Source: [38] via [54]).

}

The `flatMap` operator (monadic bind) concatenates the paths that constitute the underlying models.

The choice operator creates a new monadic value and assigns a fresh identifier to keep track of the choice:

```
def choice[A](xs: (A,Prob)*): Rand[A] = {
  val id = freshChoiceId()
  Rand[A](xs.toList.map { case (x,p) => Choice(id,x,p, Root) } )
}
```

It is crucial to note that our probability monad models search trees, not just collapsed distributions (like most other probability monads in the literature). To preserve the by-value semantics of the choice operator from the shallow embedding, it is necessary to distinguish individual random choices. Otherwise, dependencies between random variables would get lost. Paths on which a single random variable is assumed to take on different values need to be identified and assigned probability zero.

From this model-as-paths representation, we can easily compute the exact posterior distribution. Here is the sketch of an (straightforward but inefficient) evaluation algorithm:

```
def evaluate[A](r: Rand[A]): Dist[A] = {
  // multiply probabilities along each path (with incompatible choices yielding p=0)
```

```
  // group paths by their bottom item and sum up probabilities
}
```

With this monadic embedding, we get exact inference on the traffic modelling example:

```
trafficModel1
NoCrash : 0.9631
Crash   : 0.0369
trafficModel2
NoCrash : 0.9109
Crash   : 0.0891
```

However, without further improvements to our embedding, we must adapt the modelling code to fit this monadic style:

```
type Driver = Light => Random[Action]
def crash(driver1D: Driver, driver2D: Driver, lightD: Random[Light]) =
  lightD.flatMap(light =>
    driver1D(light).flatMap(driver1 =>
      driver2D(otherLight(light)).flatMap(driver2 =>
        (driver1, driver2) match {
          case (Drive,Drive) => choice(Crash -> 0.9, NoCrash -> 0.1)
          case _             => always(NoCrash)
        })))
```

Explicit monadic style everywhere is cumbersome. Scala's for-comprehension, like Haskell's 'do'-notation, provides only some relief:

```
def crash(driver1D: Driver, driver2D: Driver, lightD: Random[Light]) =
  for {
    light   <- lightD
    driver1 <- driver1D(light)
    driver2 <- driver2D(otherLight(light))
  } yield (driver1,driver2) match {
    case (Drive,Drive) => choice(Crash -> 0.9, NoCrash -> 0.1)
    case _             => always(NoCrash)
  }
```

We still have to be very explicit and extract values from the monadic domain before using them. To regain some readability and convenience, we would like to program on a higher level, by lifting more operations to the `Rand[T]` domain. For example, we would like to be able to add two `Random[Int]` values.

We first define a generic lifting function for binary operations, which can lift any function with signature `(A,B) => C` to the domain of random values `(Rand[A],Rand[B]) => Rand[C]`:

```
def liftOp2[A,B,C](x: Rand[A], y: Rand[B])(f: (A,B) => C): Rand[C] =
  for (a <- x; b <- y) yield f(a,b)
```

We then use Scala's implicit classes to add arithmetic to `Rand[Int]` as follows:

```
implicit class RandArithOps(x: Rand[Int]) {
  def +(y: Rand[Int]) = liftOp2(x,y)(_ + _)
  def *(y: Rand[Int]) = liftOp2(x,y)(_ * _)  ...
}
```

We are now able to express, e.g., the sum of two dice rolls without explicit monadic operations:

```
val sumOfDice = rand(1,6) + rand(1,6)
```

It is also handy to define an implicit conversion from `T` to `Rand[T]`, which enables us to treat all values as random values (e.g. to express `rand(1,6) * 10`):

```
implicit def liftVal[T](x: T) = always(s)
```

Similarly, we may want to define an operator `a pair b` that takes expressions `a:Rand[T]` and `b:Rand[U]` to type `Rand[(T,U)]`. We use an implicit class again:

```
implicit class RandUtilsOps[T](x: Rand[T]) {
  def pair(y: Rand[Int]) = liftOp2(x,y)((_,_))
}
```

We can already see that adding individual methods to `Rand[T]` values using implicit classes incurs a fair bit of syntactic boilerplate. Since this situation is so common, Scala-Virtualized adds another facility, infix methods. In Scala-Virtualized, we can write:

```
def infix_pair[A,B](x: Rand[A], y: Rand[B]) = liftOp2(x,y)((_,_))
```

to extend `Rand[A]` with a method `pair`. The semantics are slightly different from implicit classes; a detailed description is given in Section 3.2.

We turn our attention back to the traffic example. Drivers now take a `Random[Light]` argument and perform the pattern matching inside a call to `flatMap`:

```
type Driver = Random[Light] => Random[Action]
def cautiousDriver(light: Random[Light])   = light flatMap { /* pattern match */ }
def aggressiveDriver(light: Random[Light]) = light flatMap { /* pattern match */ }
```

Now we can write the crash model like this (using the tuple lifting operation `pair`):

```
def crash(driver1: Driver, driver2: Driver, light: Rand[Light]) =
  (driver1(light) pair driver2(otherLight(light))) flatMap {
    case (Drive,Drive) => choice(Crash -> 0.9, NoCrash -> 0.1)
    case _             => NoCrash
  }
```

If we prefer, we can also write it like this (we need to introduce `===` and `&&` methods):

```
def crash(driver1: Driver, driver2: Driver, light: Rand[Light]) = {
  (driver1(light) === Drive && (driver2(otherLight(light)) === Drive)) flatMap {
    case true => choice(Crash -> 0.9, NoCrash -> 0.1)
    case _    => NoCrash
  }
}
```

In standard Scala, we cannot use if-then-else or `==`. Here we are hitting the limits of Scala's DSL capabilities: if-then-else is a built-in construct, and implicits can only define *new* methods, not redefine pre-existing methods like `==`. In Scala-Virtualized, we can recover the exact syntax of the very shallow embedding by overloading if-then-else and the equality operator `==` to operate on `Rand[Boolean]` (Section 3.1).

This monadic embedding is still arguably shallow, because we are just lifting regular operations into the monad.

2.1.3 A Deep Embedding

A deep embedding reifies the DSL program into a data structure, enabling analyses and optimizations. As a motivating example, suppose we would like to evaluate the following DSL program, which computes the sums of a series of coin tosses:

```
val coins = for (i <- 0 until 10) yield flip(0.5)
val sum = coins.map(c => if (c) always(1) else always(0)).sum
val fiveHeads = sum == always(5)
```

Depending on the strength of our probabilistic intuition, we may or may not realize that we could have written the same program as follows:

```
val sum = binomial(0.5, 10)
val fiveHeads = sum == always(5)
```

The second implementation enables much cheaper inference: Instead of evaluating all 2^{10} paths defined by the 10 coin tosses, we can compute probabilities for the outcome of the binomial distribution directly using the well-known formula

$$\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

In the following, we show how to recognize a sum of coin tosses as a binomial distribution automatically and transform the program accordingly. In other words, we automatically rewrite the first snippet into the second, optimized, one.

We define the actual probabilistic operations as data types:

```
abstract class Exp[T]
case class Flip(id: Int, p: Prob) extends Exp[Boolean]
case class Binomial(id: Int, p: Prob, n: Int) extends Exp[Int]
case class Always[A](e: A) extends Exp[A]

case class Plus(x: Exp[Int], y: Exp[Int]) extends Exp[Int]
case class Equals[A](x: Exp[A], y: Exp[A]) extends Exp[Boolean]
case class IfThenElse[T](x: Exp[Boolean], y: Exp[T], z: Exp[T]) extends Exp[T]
```

We express the operations from the very shallow embedding in terms of these data types, so that the clients of the DSL just use the same intuitive syntax.

```
def flip(p: Prob)             = Flip(freshId(), p)
def binomial(p: Prob, n: Int) = Binomial(freshId(), p, n)
def always[A](e: A)           = Always(e)

def infix_+(x: Exp[Int], y: Exp[Int]): Exp[Int]                 = Plus(x, y)
def __equals[A](x: Exp[A], y: Exp[A]): Exp[Boolean]             = Equals(x, y)
def __ifThenElse[T](x: Exp[Boolean], y: => Exp[T], z: => Exp[T]) = IfThenElse(x,y,z)
```

The last two methods, `__equals` and `__ifThenElse`, implement the `==` and `if (x) y else z` constructs in Scala-Virtualized. We will introduce more details about this mechanism in Section 3.

We define transformations that capture that the two `fiveHeads` snippets are equivalent. For soundness, these rewritings are subject to a linearity condition: All choice ids on the left hand side of a rewrite rule must be unique and not appear anywhere else in the program. An expression like `val r = binomial(p, 5); r+r` should *not* be rewritten to `binomial(p, 10)`.

```
IfThenElse(Flip(id, p), Always(1), Always(0))      ⟶  Binomial(id, p, 1)
Plus(Binomial(id1, p, n1), Binomial(id2, p, n2))   ⟶  Binomial(freshId(), p, n1 + n2)
Plus(Always(0), r)                                 ⟶  r
```

After transforming our model, we can either interpret it or generate code. The evaluation procedure is free to make probabilistic choices in any order, and use any kind of inference algorithm that respects the model semantics.

Finally, notice that the deep embedding does not model lists or functions, but our example readily used a list of coin flips. This is a case of deep linguistic reuse: We are able to keep the deep embedding simple, because the feature of lists is translated away in the shallow embedding part of the language. Of course the limitation is that we are only working with lists of random values, not with actual random distributions over lists.

3 Everything is a Method Call

The overarching idea of embedded languages is that user-defined abstractions should have the same rights and privileges as built-in abstractions. Scala-Virtualized redefines many built-in abstractions as method calls, so that the corresponding method definitions may be redefined by a DSL, just like any other method.

The essential difference between Scala-Virtualized and regular Scala is that more Scala language constructs are expressed in terms of method calls. The aim is to cover the full expression sub-language, removing any special forms so that all expressions are either constants, references or method calls. This fits nicely with the "finally tagless" [8] or polymorphic DSL embedding [20] approach, which goes back to an old idea of Reynolds [39], namely representing object programs using method calls rather than data constructors. By overriding or overloading the default implementations appropriately, the embedding can be configured to generate an explicit program representation. Note that we do not in general redefine "non-executable" terms like types, class definitions, etc. (but see Section 5 for our treatment of record types).

3.1 Virtualizing Control Structures

In Scala-Virtualized, an expression such as `if (c) a else b` is defined as the method call `__ifThenElse(c,a,b)`. By providing its own implementation of a method with this name, a DSL is free to define the meaning of conditional expressions within the DSL. Among other options, the DSL can have the method generate an AST for this part of the domain program, which can then be further analyzed and optimized by the DSL implementation. When no alternative implementation is provided, the if-then-else has the usual semantics.

This approach fits well with the overall Scala philosophy: for-comprehensions and parser combinators were implemented like this from the beginning, and the Scala language already has similar definitions for certain types of expressions, for example:

```
matrix(i,j)       // defined as matrix.apply(i,j)
matrix(i,j) = x   // defined as matrix.update(i,j,x)
```

Unlike approaches that lift host language expression trees 1:1 using a fixed set of data types, the DSL implementor has tight control over which language constructs are lifted and which are not.

To return to the probability DSL from Section 2.1, we can define versions of **if** for both the monadic embedding

```
def __ifThenElse[T](cond: Rand[Boolean], thenp: => Rand[T], elsep: => Rand[T]): Rand[T] =
  cond.flatMap(c => if (c) thenp else elsep)
```

and also for the deep embedding:

```
def __ifThenElse[T](cond: Exp[Boolean], thenp: => Exp[T], elsep: => Exp[T]): Exp[T] =
  IfThenElse(cond, thenp, elsep)
```

These definitions enable us to use conditionals with `Rand[Boolean]` and `Exp[Boolean]` values as if they were regular `Booleans`:

```
val bias = flip(0.3)
val coin = if (bias) flip(0.6) else flip(0.5)
```

In addition to **if**, the following control structures and built-ins (left column) are virtualized into method calls (right column):

`if (c) a else b`	`__ifThenElse(c, a, b)`
`while(c) b`	`__whileDo(c, b)`
`do b while(c)`	`__doWhile(b, c)`
`var x = i`	`val x = __newVar(i)`
`x = a`	`__assign(x, a)`
`return a`	`__return(a)`
`a == b`	`__equal(a, b)`

While it should be evident that DSLs can implement any of these methods to their choosing, there also needs to be a default implementation that is used by regular, non-DSL Scala code. This default implementation is defined in a trait[2] `EmbeddedControls`, in the top-level `scala` package:

```
package scala
trait EmbeddedControls {
  def __ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T): T
  def __whileDo(cond: => Boolean, body: => Unit): Unit
  def __doWhile(body: => Unit, cond: => Boolean): Unit
  def __newVar[T](init: T): T
  def __assign[T](lhs: T, rhs: T): Unit
  def __return(expr: Any): Nothing
  def __equal(expr1: Any, expr2: Any): Boolean
  // (other definitions elided)
}
```

Trait `EmbeddedControls` is mixed into `scala.Predef`, which is implicitly imported in every compilation unit. Thus, the default virtualization hooks are available at the top-level in any Scala program and the methods are visible everywhere. If the Scala-Virtualized compiler resolves a particular invocation of `__ifThenElse` to the method defined in `EmbeddedControls`, this means that there was no DSL-defined implementation that took precedence. Thus, the compiler will perform the usual non-virtualized translation of conditionals that the regular Scala compiler would do.

[2] In Scala, traits are similar to classes, but they can take part in mixin-composition, a restricted form of multiple inheritance [34].

Embedded languages can use the standard language mechanisms of overloading, overriding or shadowing to control how DSL definitions interact with the default implementations. For example, programmers can shadow the default implementation by importing a different one into a lexical scope:

```
// import scala.Predef._  (implicit)
object DSL {
  def __ifThenElse[T](cond: Exp[Boolean], thenp: => Exp[T], elsep: => Exp[T]): Exp[T] = ...
}
import DSL._
if (c) a else b  // always resolves to the DSL version
```

If this is not the desired behavior, a DSL implementation can inherit from `EmbeddedControls` to put its own method definitions into an object-oriented overloading relation with the default ones:

```
// import scala.Predef._  (implicit)
object DSL extends EmbeddedControls {
  // may override the default implementation
  def __ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T): T = ...
  // may overload method with a DSL specific signature
  def __ifThenElse[T](cond: Exp[Boolean], thenp: => Exp[T], elsep: => Exp[T]): Exp[T] = ...
}
import DSL._
if (c) a else b  // resolved according to overloading resolution
```

The conditional resolves to the most specific method in object `DSL`. If `c,a,b` are plain Scala types, this may now be the default implementation from `EmbeddedControls`, if it is not overridden by the DSL. In summary, all the usual ways of structuring object oriented programs apply. It is worth pointing out that in general, these mechanisms are hygienic in the sense that it is not possible to remotely introduce new bindings into a distant lexical scope. In some cases more relaxed conditions are desirable, which is why Scala-Virtualized introduces the notion of DSL scopes (see Section 6.4).

Some readers may wonder why variable definitions (`var x = ..`) are virtualized, but not value bindings (`val x = ..`). The short answer is that we can reuse the binding structure of the host language for ordinary value bindings (see Section 6.1).

3.2 Virtualizing Method Calls

Ordinarily, there are two ways to customize the meaning of an expression such as `x a y`, which is short for `x.a(y)`. Obviously, if we control the type of `x`, we can simply introduce the appropriate method in its class. Otherwise an implicit class can be used — if (and only if) `x`'s type does not already provide an (appropriately typed) member `a`. While very useful, this technique requires a fair bit of boilerplate code. Most importantly, it cannot be used to override existing methods, such as the ubiquitous method `toString`, which is defined at the top of the type hierarchy.

Overriding existing behavior is necessary in a number of cases for embedded DSLs. Let us assume that we are working with a deeply embedded DSL and want to write a program that iteratively computes some values. We would like to write those values into an `ArrayBuffer` that is defined outside the DSL world, i.e. it is a constant in the deep embedding representation:

```
val buffer = new ArrayBuffer[String]
import DSL._
OptimizeAndRun {
  while (!done()) {
    val res = computeResult()
    buffer += "----------"
    buffer += res
  }
}
```

We take `OptimizeAndRun` to be a DSL scope (see Section 6.4) that enables virtualization for the enclosed code, compiles and executes it. The issue is now that class `ArrayBuffer` already has a `+=` method but we need to override the behavior within the DSL to correctly represent all buffer writes in the deep embedding of the program. If we do not, the line of dashes will be added to the buffer while the deep embedding is constructed, but it will not become part of the reified program representation, as would be expected. This is due to the fact that both the buffer and the constant string have plain Scala types, not types corresponding to the deep embedding.

Scala-Virtualized introduces infix functions that are able to selectively and externally introduce new methods on existing types, as well as override existing ones, without any run-time overhead. The idea is simple: We redefine `x.a(y)` as `infix_a(x,y)`. If the type of `x` has any members with name `a`, we insert corresponding sentinel methods of the form `def infix_a(x,y)` into `EmbeddedControls`. If overloading resolution picks one of the sentinels, the regular invocation `x.a(y)` is chosen. Otherwise a user-defined method takes precedence.

In the buffer example above, our DSL can define a method `infix_+=` that will take precedence over the method defined in class `ArrayBuffer`. Thus, all buffer writes are correctly reified into the deep embedding.

3.2.1 Example: DSLs With a Restricted Grammar

Even in cases where implicit classes can be used, infix functions can sometimes greatly reduce boilerplate. We consider a simple case of a DSL that requires a fixed sentence structure.

Complementary to Scala's syntactic flexibility, Scala's type system also enables enforcing certain restrictions. For example, it may be desirable to restrict DSL expressions to a given grammar. Here is an example of how adherence of DSL expressions to a context-free grammar ($a^n b^n$) can be enforced using phantom types and infix functions:

```
object Grammar {
  type ::[A,B] = (A,B)
  class WantAB[Stack] extends WantB[Stack]
  class WantB[Stack]
  class Done
  def start() = new WantAB[Unit]
  def infix_a[Stack](s: WantAB[Stack]) = new WantAB[Unit::Stack]
  def infix_b[Rest](s: WantB[Unit::Rest]) = new WantB[Rest]
  def infix_end(s: WantB[Unit]) = new Done
  def phrase(x: => Done): String = "parsed"
}
```

```
import Grammar._
phrase { start () a () a () b () b () end () }      // "parsed"
phrase { start () a () a () b () b () b () end () } // error
phrase { start () a () a () b () end () }           // error
```

The same behavior can be encoded in vanilla Scala using implicit classes but in a more verbose way:

```
object Grammar {
  type ::[A,B] = (A,B)
  class WantB[Stack]
  class Done
  class WantAB[Stack] extends WantB[Stack] {
    def a() = new WantAB[Unit::Stack]
  }
  implicit class W2[Rest](s: WantB[Unit::Rest]) {
    def b() = new WantB[Rest]
  }
  implicit class W3[Rest](s: WantB[Unit]) {
    def end() = new Done
  }
  def start() = new WantAB[Unit]
  def phrase(x: => Done): String = "parsed"
}
```

4 Virtualizing Pattern Matching

We explain how Scala's standard pattern matching can be redefined as operations on a zero-plus monad. We then present custom pattern matchers: a reifier of pattern matching expressions for a deep embedding (Section 4.1), and a DSL for probabilistic logic programming (Section 4.2).

Scala supports pattern matching in a way similar to ML and Haskell but with *case classes* playing the role of data constructors. A case class is a convenient way of defining a class whose instances can be deconstructed using pattern matching. Scala provides extractors to decouple the internal data representation of an object and how it is deconstructed [14]. Patterns are to constructors as extractors are to factory methods: they provide an abstraction layer for pattern matching just like factory methods allow customizing what it means to construct a new object. If a factory method for a data type `U` is a function `(T1, ..., TN) => U`, the corresponding most general extractor is a function `Any => Option[(T1, ..., TN)]` where `Any` is the top of the subtype lattice and `Option` is Scala's `Maybe` monad.

Consider the following example of a case class `P` with two integer fields, `a` and `b`:

```
case class P(a: Int, b: Int)
```

The Scala compiler expands this class definition as follows:

```
object P {
  def apply(a: Int, b: Int): P = new P(a, b)
  def unapply(p: P): Some[(Int, Int)] = Some((p.a, p.b))
}
class P(val a: Int, val b: Int)
```

Note that the signature of `P`'s `unapply` method indicates that it is only applicable to a `P` and that it always succeeds (since `Some[T]` is the subtype of `Option[T]` that indicates success). Concretely, the stricter type `P` for the unapply's argument gives rise to a type test that ensures that the extractor can be called. Similarly, its return type tells the pattern matching analyses that the extractor is irrefutable. A Scala programmer may define these `apply` and `unapply` methods explicitly, thus performing arbitrary validation during construction and transparent rewriting of the internal representation during deconstruction. Programmers can also implement extractor objects that are not tied to particular case classes at all. For instance, we can define an extractor on integers that succeeds only if a number is even, returning its half if so:

```
object Twice {
  def unapply(x: Int): Option[Int] = if (x%2 == 0) Some(x/2) else None
}
```

To see how an `unapply` method customizes pattern matching, we formulate the following match expression in terms of extractor calls:

```
P(1, 2) match { case P(x, y) => x + y }
```

Conceptually, this match expression corresponds to:

```
val px = P.unapply(P.apply(1, 2))
if (px.isEmpty) throw new MatchError
else px.get._1 + px.get._2
```

A pattern match is virtualized by interpreting it as a computation in a (user-defined) zero-plus monad. Extractors need to be adapted accordingly: A single pattern corresponds to a computation that fails (it returns the monad's `zero`) when the pattern does not match the input, and that succeeds with a tuple of the values to be matched by its sub-patterns. Pattern nesting gives rise to sequencing of these computations (usually called the `bind` or `flatMap` operator). Cases (and alternative patterns) are combined into a match using the alternative combinator (the `plus` or `orElse` operator). Finally, a guard preserves a computation if the conditional succeeds and fails otherwise. The body of a case is a computation that always succeeds.

Consider virtualizing the match `x match { case P(a, b) => a + b }` in Scala's standard zero-plus monad, `Option`, where computations are sequenced using `flatMap`, and alternatives combined using `orElse`. A successful computation's result is stored in a `Some` and failure yields a `None`. In this monad the above snippet is virtualized as `P.unapply(x).flatMap{ o => Some(o._1 + o._2) }`. The successful result is passed to the function that is bound to (or "flatMapped over") the result of `P`'s `unapply` method.

A nested match, such as

```
x match { case Q(R(x), S(y)) => x + y }
```

is virtualized as follows:

```
Q.unapply(x).flatMap(x1 =>
  R.unapply(x1._1).flatMap(x2 =>
    S.unapply(x1._2).flatMap(x3 =>
      Some(x2 + x3))))
```

In general, a case is flattened using a depth-first traversal of the translations of its patterns, with the individual computations combined using `flatMap`.

A match with alternatives, such as

```
x match { case P(x, y) => x; case Q(x, y) => y }
```

is virtualized as follows:

```
P.unapply(x).flatMap(x1 => Some(x1._1)).orElse(
  Q.unapply(x).flatMap(x2 => Some(x2._2)))
```

Besides data constructor patterns that give rise to `unapply` extractor calls, Scala's pattern language includes literals, types and sequences. The translation of these patterns does not introduce any new insights.

Finally, match virtualization is triggered by introducing the appropriate `__match` object into scope. It must define four methods – `zero`, `one`, `guard`, `runOrElse` – that specify the semantics of a match. The signature of the `one` method determines the type constructor of the monad.

The above examples assumed the following `__match` object:

```
object __match {
  def zero = None
  def one[T](x: T): Option[T] = Some(x)
  def guard[T](cond: Boolean, then: => T): Option[T] =
    if (cond) Some(then) else None
  def runOrElse[T, U](in: T)(matcher: T => Option[U]): U  =
    matcher(in).getOrElse(throw new MatchError(in))
}
```

The full expansion of `x match { case P(a, b) => a + b }` is:

```
__match.runOrElse(x)(x1 =>
  P.unapply(x1).flatMap(x2 =>
    __match.one(x2._1 + x2._2)))
```

Internally, the Scala-Virtualized compiler will apply all the usual optimizations if the (non-virtualized) default case is detected, such as translating match expressions to conditionals and generating jumps instead of method calls. Since the virtualization of a match and, say, an if-then-else, is fundamentally the same, we can use the techniques for obtaining a deep embedding of match expressions with the same techniques used for other control structures. Furthermore, other interesting monads can be plugged in to allow matches to back track, or express parsers and automata as pattern matches (see Section 4.2 for an example).

4.1 Example: Reifying Match Expressions

Figure 2 sketches a deep embedding of pattern matching. First, we define a deep embedding of the option monad. To distinguish it from the actual Scala `Option` type we introduce a phantom type `Maybe`. Values of the lifted monad have types `M[T] = Exp[Maybe[T]]`. Most operations directly create a data structure representation using the corresponding case classes. The `flatMap` operation unfolds its argument function with a fresh symbol as argument to obtain a flat algebraic representation.

The pattern matching operations are then defined using the lifted monad. Thus, match expressions will be reified into an intermediate representation. For example, the expression

```
7 match { case 5 => "foo"; case _ => "bar" }
```

is desugared into

```
7 match { case x if x == 5 => "foo"; case _ => "bar" }
```

```
abstract class Maybe[+T]
type M[+T] = Exp[Maybe[T]]

case object Zero extends M[Nothing]
case class One[T](x: Exp[T]) extends M[T]
case class OrElse[T](x: M[T], y: M[T]) extends M[T]
case class FlatMap[T,U](x: Exp[T], y: Exp[T], z: M[U]) extends M[U]
case class ResultOrMatchError[T](x: M[T]) extends Exp[T]
case class IfThenElse[T](x: Exp[Boolean], y: M[T], z: M[T]) extends M[T]

implicit class MaybeOps[A](self: M[A]) {
 def flatMap[B](f: Exp[A] => M[B]): M[B]   = { val x = freshSym[A]; FlatMap(self, x, f(x)) }
 def map[B](f: Exp[A] => Exp[B])           = flatMap(x => One(f(x)))
 def orElse[B >: A](alt: => M[B]): M[B]    = OrElse(self, alt)
}

object __match {
 def zero: M[Nothing]                                          = Zero
 def one[T](x: Exp[T]): M[T]                                   = One(x)
 def guard[T](cond: Exp[Boolean], then: => Exp[T]): M[T]       = IfThenElse(cond, one(then), zero)
 def runOrElse[T, U](in: Exp[T])(matcher: Exp[T] => M[U]): Exp[U]  = ResultOrMatchError(matcher(in))
}

println(7 match { case _ => "bar" })
// ResultOrMatchError(One(bar))

println(7 match { case 5 => "foo"; case _ => "bar" })
// ResultOrMatchError(OrElse(FlatMap(IfThenElse(7==5, One(7), Zero), x, One(foo)), One(bar)))
```

Fig. 2 Reifying pattern match expressions

which expands to

```
__match.runOrElse(7)(x =>
  __match.guard(x == 5, "foo").orElse(__match.one("bar")))
```

and evaluates to

```
ResultOrMatchError(
  OrElse(
    FlatMap(IfThenElse(7 == 5, One(7), Zero), x, One("foo")),
    One("bar")))
```

assuming proper lifting of constants and ==.

Currently, case class extractors are not lifted automatically, because the option monad is hard-coded in their specification. Though we are planning improvements to generalize case class extractors, custom extractors readily provide a work-around. For example, we can reify pattern-matching against reified lists by defining lifted extractors manually:

```
case class Unapply[T, U](kind: String, x: Exp[T]) extends M[U]
object Nil {
  def unapply(x: Exp[List[_]]): M[Unit] = Unapply("Nil", x)
}
object Cons {
```

```
  def unapply[T](x: Exp[List[T]]): M[(T, List[T])] = Unapply("Cons", x)
}
```

Then, the match expression

```
List(1, 2, 3) match { case Nil() => "default"; case Cons(hd, tl) => hd }
```

evaluates to

```
ResultOrMatchError(
  OrElse(
    FlatMap(Unapply(Nil,List(1, 2, 3)),x,One(default)),
    FlatMap(Unapply(Cons,List(1, 2, 3)),x,One(x._1))))
```

assuming proper lifting of tuples and the type test methods `isInstanceOf` and `asInstanceOf`.

4.2 Example: Logic programming

Since alternatives in pattern matching are combined using `orElse`, we can create a non-deterministic pattern matcher by swapping the default `Option` monad for a more sophisticated one that explores more than one alternative. As an example, we customize the pattern matcher with the underlying `Rand` monad described in Section 2.1.2. The definition of the match logic as well as examples that define rules and exercise the logic programming capabilities are shown in Figure 3.

In this probabilistic pattern matcher, an extractor's `unapply` method returns a `Rand[T]` instead of an `Option[T]`. We define an implicit class `Rule` to lift any function from `String => Rand[String]` into an extractor:

```
implicit class Rule(f: String => Rand[String]) {
  def unapply(x: String): Rand[String] = f(x)
}
```

Et voilà! We have embedded a simple rule-based logic programming into the pattern matcher.

The first thing to notice is that pattern matching no longer follows a first-match-wins policy but may explore multiple branches. In the definition of rule `Friend`, for example, the pattern "A" occurs three times on the left-hand side. Probabilities are implicit, and a certain case can be given more weight by repeating it. This is why the double occurrence of the pair "A","C" will count with twice its weight, denoting a strong friendship. Rule `Knows` is the reflexive transitive closure of `Friend` and illustrates recursive rules.

Rules expressed as extractors compose intuitively. The definition of `ShouldGrabCoffee` can be almost read out loud: if x likes coffee, and x knows someone else, y, who also likes coffee, then they should grab coffee together:

```
lazy val ShouldGrabCoffee: Rule = { x: String => x match {
  case Likes("Coffee") && Knows(y @ Likes("Coffee")) if x != y =>
    x + " and " + y + " should grab coffee"
}}
```

Since probabilities are always carried along with the computation, the final results not only produces a list of coffee matchings, but assigns a weight to each of them, which can be interpreted in an application-specific way.

```
object __match {
  def zero                                                  = never
  def one[T](x: T): Rand[T]                                 = always(x)
  def guard[T](cond: Boolean, result: => T): Rand[T]        = if (cond) one(result) else zero
  def runOrElse[T, U](in: T)(matcher: T => Rand[U]): Rand[U] = matcher(in)
}

implicit class Rule(f: String => Rand[String]) {
  def unapply(x: String): Rand[String] = f(x)
}
val && : Rule = { x: String => x match {
  case x => (x,x)
}}
val Likes: Rule = { x: String => x match {
  case "A" => "Coffee"
  case "B" => "Coffee"
  case "D" => "Coffee"
  case "D" => "Coffee" // likes coffee very much!
  case "E" => "Coffee"
}}
val Friend: Rule = { x: String => x match {
  case "A" => "C"
  case "A" => "C" // are really good friends!
  case "C" => "D"
  case "B" => "D"
  case "A" => "E"
}}
val Knows: Rule = { x: String => x match {
  case Friend(Knows(y)) => y
  case x => x
}}
val ShouldGrabCoffee: Rule = { x: String => x match {
  case Likes("Coffee") && Knows(y @ Likes("Coffee")) if x != y =>
    x + " and " + y + " should grab coffee"
}}


val coffeeModel1 = uniform("A","B","C","D","E") flatMap { case ShouldGrabCoffee(y) => y }
```

Result:

```
A and D should grab coffee : 0.5714285714285714
B and D should grab coffee : 0.2857142857142857
A and E should grab coffee : 0.14285714285714285
```

Fig. 3 Logic programming: using virtualized pattern matching with the probability monad

5 Virtualizing Type Information and Object Construction

We have seen how we can virtualize control structures and method calls. How can we virtualize working with objects, classes and types? The general idea is to make object construction and field accesses overridable in a way that preserves the type structure. But let us go step by step.

For any given class, we readily can build up a lifting manually, defining virtualized factory and accessor methods. For example, given a class such as `Item`:

```
class Item(val itemName: String, val customerName: String)
```

```
val a = new Item("undisclosed_item", "John Doe")
println(a.customerName)
```

we can define a lifted interface:

```
def NewItem(itemName: Exp[String], customerName: Exp[String]): Exp[Item]
def infix_itemName(x: Exp[Item]): Exp[String]
def infix_customerName(x: Exp[Item]): Exp[String]
```

and use it in (almost) the same way:

```
val a = NewItem("undisclosed_item", "John Doe")
println(a.customerName)
```

However, defining an embedding for each class separately is tedious and it only covers the case of a fixed universe of classes. This is not a very realistic assumption. We want to enable DSL users to define new types in their programs, too. As an example, we present a deep embedding of a small statically-typed query language that can generate SQL statements to be run in a relational database.

5.1 Example: Embedding SQL

Our small query DSL understands table definitions, such as

```
type Item = Record {
  val itemName: String
  val customerName: String
}
val items = Table[Item]("items")
```

and queries, such as

```
items Select (e => new Record {
  val name = e.customerName
})
```

from which SQL can be generated:

```
SELECT customerName AS name FROM items
```

Our DSL is statically typed, so that it would be an error to refer to a non-existing field.

Our DSL is deeply embedded, with the following data-type representation:

```
trait Exp[T]
case class Const[T](x: T) extends Exp[T]
implicit def liftString(x: String): Exp[String] = Const(x)
case class ResultRecord[R](fields: Map[String, Exp[_]]) extends Exp[R]
case class Select[T, U](tgt: Exp[U], field: String) extends Exp[T]
case class Table[R <: Record](name: String) extends Exp[List[R]]
case class ListSelect[R <: Record, T](t: Exp[List[R]], f: Exp[R] => Exp[T]) extends Exp[List[R]] {
  def projectedNames: Iterable[String] = f(null) match {
    case ResultRecord(fields) => fields.keys
  }
}
...
```

The `Const` class defines a constant that is lifted into the DSL representation. The implicit `liftString` automatically represents string literals, such as `"hello"`, as expression trees, such as `Const("hello")`.

An expression object of type `ResultRecord` describes the shape of results returned by an SQL `Select` clause. It contains a mapping from column names to expression trees that describes how each column value of the result is computed. The `Select` case class represents expressions that select a field on a target record, e.g., `item.customerName`. To represent database tables we use the `Table` case class. It takes a type parameter `R` that abstracts over the type of records stored in the table. `R` extends the `Record` class, introduced below. Essentially, `Record` is used to create result records in queries. The `Table` class extends `Exp[List[R]]`, i.e., it represents an expression returning a list of `R` elements. This enables us to treat a table literal (which only contains the name of the corresponding database table) as a list of records, for which the standard `Select` clauses are defined.

To represent `Select` clauses we use the `ListSelect` case class. Like `Table`, it extends `Exp[List[R]]` where `R` is a type parameter for the type of the returned records. A `ListSelect` expression node points to a list expression of type `Exp[List[R]]`, which is the list that we are selecting elements from, and a selector function of type `Exp[R] => Exp[T]`, which is used for determining (a) how to select elements from the list, and (b) how to transform a selected element to a result record. For example, the following DSL expression is represented as a `ListSelect` node:

```
items Select (e => new Record {
  val customerName = e.customerName
})
```

In this case, `items` is lifted to an expression tree of type `Exp[List[R]]`, and the function literal is lifted to an expression tree constructor function of type `Exp[R] => Exp[T]`. Section 5.2 explains how the selector function works: How can it use all of the fields defined in `R` type to select columns to be included in the result? How is the result lifted to an expression tree?

Finally, we add an implicit conversion that lifts generic lists into expression trees:

```
implicit def liftList[T](x: List[T]): Exp[List[T]] = Const(x)
```

Let us now define the methods that are concerned with the actual embedding of query expressions in Scala. The first method we will add provides `Select` clauses on lists of records:

```
implicit def listSelectOps[R <: Record](l: Exp[List[R]]) = new {
  def Select[T](f: Exp[R] => Exp[T]): Exp[List[R]] = ListSelect(l, f)
}
```

This method relies on Scala's standard implicit conversion mechanism: since an expression of type `Exp[List[R]]` does not provide a method called `Select`, the compiler will turn an expression such as

```
table Select { f => ... }
```

where `table : Exp[List[R]]`, into

```
listSelectOps(table).Select{ f => ... }
```

The `Select` method call will in turn create a representation of the DSL's `Select` expression, an expression node of type `Exp[List[R]]`.

5.2 Virtualizing Record Types

As we have seen in the previous example, result records are created using:

```
new Record { val column_i = ... }
```

To represent such an expression in the AST of our DSL, we therefore need to lift instance creation using **new** (see Section 5.3). We use the `Struct` type constructor, which is part of the Scala-Virtualized library, as a marker to indicate when an instance creation **new** `T` should be virtualized. This reification will take place whenever `T <: Struct[R]`, for some type constructor `R`. For convenience, we declare the following class:

```
class Record extends Struct
```

The actual lifting relies on the `__new` method, which is defined as follows for our embedding:

```
def __new[T](args: (String, Boolean, Exp[T] => Exp[_])*): Exp[T] =
  new ResultRecord(args map { case (n, _, rhs) => (n, rhs(null)) } toMap)
```

The `__new` method takes a variable number of triples as arguments. Each triple contains the name of a field in our record type, whether it was declared as mutable or immutable (i.e. **var** or **val** in Scala), and a function which creates an expression tree for the field initializer in terms of an expression representing the "self" instance. Since SQL does not support self-referential rows, we simply pass **null** as the representation of the self reference. A more robust implementation could inject an `ErrorExpresion` so that code generation can emit a suitable error message. In any case, we simply create an instance of `ResultRecord`, using the arguments to fill its map.

To support projections in queries, we need to be able to select fields of records. Therefore, we need to have a way to create expression trees for field selections. In Scala-Virtualized we can lift such selections by defining the special `selectDynamic` method for the types of objects on which we would like to select fields. We can provide this method through the following implicit conversion:

```
implicit def selectOps(self: Exp[_ <: Record]) = new {
  def selectDynamic[T](n: String): Exp[T] = Select(self, n)
}
```

When a field selection does not type check according to the normal typing rules, as would be the case for, e.g., the selection `e.customerName` since `e`'s type `Exp[R]` does not define a field `customerName`, the Scala-Virtualized compiler will generate an invocation of the `selectDynamic` method on `e` – because `R` relates to the special marker `Struct` as explained in Section 5.3. Since it knows from the record type that the selection is supposed to produce an expression that evaluates to a `String`, it will pass this information along to `selectDynamic` as a type argument.

Due to the above implicit conversion, objects of type `Exp[_ <: Record]`, i.e., expression trees whose result type is a subtype of `Record`, have a `selectDynamic` method that creates a `Select` node that contains the expression tree of the target of the selection (`self`), the name of the selected field (`n`), and that statically specifies the result type of the expression.

5.3 Translation in Detail

The Scala-Virtualized compiler can turn an expression `new C{val/var x_i: T_i = v_i}` into a method call `__new( ("x_i", false/true, (self_i: R) => v'_i) )`. Virtualization relies on a marker trait `Struct` defined in `EmbeddedControls`:

```
trait Struct
```

Virtualization is not performed, unless `C` is a subtype of `Struct`.

Furthermore, for all `i`,

- there must be some `T'_i` so that `T_i = Rep[T'_i]` – or, if that previous equality is not unifiable, `T_i = T'_i`
- `v'_i` results from retyping `v_i` with expected type `Rep[T'_i]`, after replacing **this** by a fresh variable `self_i` (with type `Rep[C{ val x_i: T'_i }]`, abbreviated as `R`)
- `mut_i` is `true` if `x_i` is mutable (declared with a **var**) and false if it's immutable (declared wtih a **val**)

Finally, the call `__new(("x_i", mut_i, (self_i: R) => v'_i))` must type check with expected type `R`. If this is the case, the **new** expression is replaced by this method call. This assumes a method in scope whose definition conforms to:
`def __new[T](args: (String, Boolean, Rep[T] => Rep[_])*): Rep[T]`.

In addition to virtualizing object creation, Scala-Virtualized provides a facility for type-safe access of record fields. When `e` refers to a representation of a record, `e.x_i` is turned into `e.selectDynamic[T_i]("x_i")` as follows. When a selection `e.x_i` does not type check according to the normal typing rules, and `e` has type `Rep[C{ val x_i: T_i }]` (for some `Rep` and where `C` and the refinement meet the criteria outlined above), `e.x_i` is turned into `e.selectDynamic[T_i]("x_i")`. Note the `T_i` type argument: by defining `selectDynamic` appropriately, the DSL can provide type safe selection on records.

5.4 Example: Probabilistic Profiles

The virtualization of `__new` provisions for self-references in the field definitions. We now illustrate a general technique to reify new structures with self-references by extending our deep embedding of the probabilistic programming DSL with "profiles" to bundle related random values. For example:

```
val person = new Profile {
  val happy   = good + healthy
  val good    = binomial(0.5, 3)
  val healthy = binomial(0.5, 3)
}
```

Our goal is for `person.happy` to reify to:

```
Select(ProfileDef(List(("happy",   Plus(Binomial(id1,0.5,3),Binomial(id2,0.5,3))),
                       ("good",    Binomial(id1,0.5,3)),
                       ("healthy", Binomial(id2,0.5,3)))
       "happy")
```

while `person.happi` should result in a compile-time error. The first argument to a `Binomial` node is an integer id that uniquely identifies the random variable. This is to express that `happy` is dependent on the choice of both `good` and `healthy`.

First, we create our marker trait `Profile`, and extend our intermediate representation with `ProfileDef`, which represents the reification of a profile definition, `new Profile { ... }`, and `Select`, which represents a field selection from a profile, such as `person.happy`.

```
trait Profile extends Struct
case class ProfileDef[R <: Profile](fields: List[(String, Exp[_])]) extends Exp[R]
case class Select[T, U](tgt: Exp[U], field: String) extends Exp[T]
```

Now, we want `__new` to evaluate each right-hand side exactly once. For this, we use a `Self` helper instance, which takes a map from each field name to its evaluation function, and caches all requests to evaluate a right-hand side.

```
def __new[T](args: (String, Boolean, Exp[T] => Exp[_])*): Exp[T] = {
  val self = new Self[T](args map{case (n, _, e) => (n -> e)} toMap)
  ProfileDef(args map {case (n, _, _) => (n -> self(n))} toList)
}
class Self[T](members: Map[String, Exp[T] => Exp[_]]) extends Exp[T] {
  private val cache = scala.collection.mutable.Map.empty[String, Exp[_]]
  def apply(name: String): Exp[_] = cache.getOrElseUpdate(name, members(name)(this))
}
```

Finally, we use an implicit class to intercept the `selectDynamic` calls generated by the Scala-Virtualized compiler. For those on a self instance (e.g. the `good` on the right-hand side of the `happy` definition of `person`), we call the self evaluator, and for those on a reified instance (e.g. the `happy` of `person.happy`), we reify the field selection.

```
implicit class ProfileOps[U <: Profile](receiver: Exp[U]) {
  def selectDynamic[T](field: String): Exp[T] = receiver match {
    case self: Self[_] => self(field).asInstanceOf[Exp[T]]
    case _             => Select(receiver, field)
  }
}
```

6 Putting it All Together

In the preceding sections, we have seen virtualization of control structures, pattern matching and user-defined types. For all of these features, we followed the "everything is a method call" pattern more or less closely. In this section, we will first discuss bindings and statement sequencing, which are not virtualized explicitly (Sections 6.1,6.2). We will continue by looking at ways to structure DSL implementations effectively (Section 6.4) and introduce the concept of DSL scopes (Section 6.4). Finally we will review an end-to-end example of combining shallow and deep embedding components for performance optmization (Section 6.5).

6.1 Deep Reuse of Bindings and Statement Order

In Section 3.1, we virtualized variable definitions `var x = y` by defining them as method calls `__newVar(y)`, but we did not treat value bindings like `val x = y` in any special way. If we have an expression like `val x = compute(); x + x` and use a simple expression-oriented deep embedding, this may lead to unexpected results because the expression will be represented as `Plus(Compute(),Compute())`. In other words, we seem to have lost the sharing

information inherent in the original expression and we can no longer distinguish it from `compute() + compute()`, which, when evaluated, may yield a very different result if the embedded language has side effects. Even if the language is pure, evaluation will likely be less efficient.

When modeling the probability DSL embedding in Section 2.1, we resorted to a trick to distinguish `val x = flip(0.5); x && x` from `flip(0.5) && flip(0.5)`: Each probabilistic choice was assigned a unique id when the corresponding expression was created.

```
def flip(p: Prob) = Flip(freshId(), p)
```

This simple mechanism illustrates an important principle: Since Scala is an impure language, we can easily observe the evaluation order of method calls that constitute a shallow DSL embedding. If these methods create a deep embedding representation and we design the corresponding data structures in such a way that the order of creation of the individual nodes is preserved, we can play back the computation in the original order without any further work. In essence, we achieve deep linguistic reuse of the host language evaluation order.

A simple way to force a certain evaluation order in the deep embedding is to create explicit `val` bindings for intermediate results (let-insertion). If bindings are inserted systematically, we obtain a representation where all intermediate results are named. Having bindings for all intermediate results implies that we also have bindings for everything that had a binding in the original program. Therefore, overapproximation enables us to reuse the binding structure of the host language.

Performing let-insertion manually is of course tedious for the DSL developer, so we would like to automate and better encapsulate this process. One avenue would be to use a let-insertion monad [53], but imposing a monadic style might be overly restrictive. Fortunately, we can map the let-insertion monad into direct style using Filinski's monadic reflection [17,18]. Implementing the corresponding `reflect` and `reify` operators would be straightforward using Scala's support for delimited continuations [41]. But it turns out we do not even need delimited control, since we can model the desired behavior directly using mutable state and a by-name parameter for `reify` (see Section 6.2).

To substantiate this intuition, the key idea is to treat DSL program fragments as context-sensitive statements, not context-free expressions. Statements need to be explicitly performed (reflected), inserting a `val` binding and generating a fresh identifier to refer to the result of the statement. The counterpart to performing a statement is accumulating (reifying) the statements performed by a block expression. We can give the operators `reflect` and `reify` the following context-sensitive semantics:

```
reify { E[ reflect(stm) ] }  ⟶  val fresh = stm; reify { E[ fresh ] }
reify { x }                  ⟶  x
```

Here, we assume `E` to be a `reify`-free evaluation context and `fresh` a fresh identifier. An implementation is shown below in Section 6.2.

With respect to extensional equality (≡) of the represented code, `reify` is a left inverse of `reflect`:

```
reify { reflect(stm) }  ⟶*  val fresh = stm; fresh  ≡  stm
```

If intensional (structural) equality is desired, a simple special case can be added to the above definition to directly translate `reify(reflect(stm))` to `stm`.

Within a suitable context, `reflect` is also a left inverse of `reify`: Reflecting a set of accumulated statements together is the same as just reflecting the statements individually.

In the following, we will present an implementation of an effectful embedded language that uses this mechanism to represent statements. We consider an embedding of JavaScript, which is a stripped-down variant of the one presented in [25].

6.2 Example: Embedding JavaScript

Consider the following JavaScript program:

```
var kim = { "name" : "kim", "age" : 20 }
kim.age = 21
if (kim.age >= 21) {
  var allowedDrink = "beer"
} else {
  var allowedDrink = "milk"
}
```

This section will show how to set up the DSL so that we can embed this program in Scala-Virtualized as follows:

```
var kim = new JSObj { val name = "kim"; val age = 20 }
kim.age = 21
var allowedDrink = if (kim.age >= 21) {
  "beer"
} else {
  "milk"
}
```

As mentioned above, Scala-Virtualized does not provide explicit support for reifying (or virtualizing) the sequencing of statements but instead relies on a shallow embedding using mutable state and the native run-time semantics of Scala statements to capture the sequencing of statements in embedded domain programs. To see how this works, let us consider the following fragment of our running example.

```
var kim = ...
kim.age = 21
```

The Scala-Virtualized compiler rewrites this to:

```
val kim = __newVar(...)
__assign(selectOps(kim).selectDynamic("age"), liftInt(21))
```

Since `kim` is a struct type (see Section 5), the field selection `kim.age` will be rewritten as a method call `kim.selectDynamic("age")`. No method `selectDynamic` exists on `kim`, though, so the receiver will be wrapped in an instance of the implicit class `selectOps`, which provides a suitable implementation.

This is the definition of `Select` and `selectOps`:

```
case class Select[T, U](tgt: Exp[U], field: String) extends Exp[T] {
implicit class selectOps(self: Exp[_ <: JSObj]) {
  def selectDynamic[T](n: String): Exp[T] = Select(self, n)
}
```

The variable definition and assignment are rewritten as `__newVar` and `__assign`, respectively. These methods are defined as follows:

```
def __newVar[T](x: Exp[T]): Exp[T]                    = VarInit(x)
def __assign[T](lhs: Exp[T], rhs: Exp[T]): Exp[Unit] = VarAssign(lhs, rhs)
```

In contrast to previous examples, `VarInit` and `VarAssign` are not expressions (`Exp[T]`); they are statements (`Def[T]`):

```
case class VarInit[T](x: Exp[T]) extends Def[T]
case class VarAssign[T](v: Exp[T], x: Exp[T]) extends Def[Unit]
```

The infrastructure for dealing with statements provides the crucial missing ingredient, namely the `reflect` operator above, which we implement as an implicit conversion from `Def[T]` to `Exp[T]`. Let us first make the conversion explicit:

```
def __newVar[T](x: Exp[T]): Exp[T] =                  = reflect(VarInit(x))
def __assign[T](lhs: Exp[T], rhs: Exp[T]): Exp[Unit] = reflect(VarAssign(lhs, rhs))
```

The crucial insight is that the order in which the `reflect` calls are executed corresponds to the order in which the statements occur in the embedded JavaScript program above. This tells us all we need to know about sequencing of statements in the embedded program.

To drive the point home, inlining `__newVar` and `__assign` peels off the last layer of syntactic sugar and indirection from our initial fragment:

```
val kim = reflect(VarInit(...))
reflect(VarAssign(selectOps(kim).selectDynamic("age"), liftInt(21)))
```

Why does this work, and why is `selectDynamic` not translated as a separate statement? The answer is in the types: `Select` extends `Exp`, whereas `VarInit` and `VarAssign` extend `Def`.

Running this Scala program creates an accurate representation of the embedded JavaScript program, as the DSL implementation keeps track of the current scope of the domain program, and `reflect` populates this scope in the order in which it is called. On each invocation, `reflect` creates a fresh symbol and enters it into the current scope. The symbol links the new entry in the current scope to the original expression.

The bookkeeping for DSL scopes uses lists of `Scope`s, themselves lists of `ScopeEntry`s to correlate a definition and the unique symbol, an expression, that is used to refer to it:

```
case class ScopeEntry[T](sym: Sym[T], rhs: Def[T])
type Scope = List[ScopeEntry[_]]
var scopeDefs: List[Scope] = Nil
```

We are now ready to present the implementation of the `reflect` and `reify` operator pair.

The method `reifyBlock` creates a block by accumulating the definitions that are entered in scope during the evaluation of the argument "`e`". This operation increases the nesting level by creating a new nested scope; it should be called when entering a block in the DSL program.

The method `reflect` performs a statement by appending a new definition to the current scope. This reifies the sequencing of definitions.

```
case class Block[T](stms: Scope, e: Exp[T])
implicit def reifyBlock[T](e: => Exp[T]): Block[T] = {
  // push a new nested scope onto the stack
  scopeDefs = Nil::scopeDefs
  // evaluate e after going to a new nesting level
  val r = e
  // reflect calls will now populate the current scope
  val stms = scopeDefs.head // save the populated scope
```

```
  scopeDefs = scopeDefs.tail // pop it
  Block(stms, r) // wrap it up in a block
}
implicit def reflect[T](d: Def[T]): Exp[T] = {
  // make a fresh symbol (: Exp[T]) to refer to the def
  val sym = freshSym[T]()
  // append it to the current scope
  scopeDefs = (scopeDefs.head :+ ScopeEntry(sym, d)) :: scopeDefs.tail
  sym // the expression-representation of the definition
}
```

With this machinery in place, we can explain the implementation of if-then-else statements for imperative DSLs, such as our JavaScript example. The representation is defined as follows, and the virtualization hook looks simple enough: it simply creates an `IfThenElse` node.

```
case class IfThenElse[T](c: Exp[Boolean], a: Block[T], b: Block[T]) extends Def[T]
def __ifThenElse[T](cond: Exp[Boolean], thenp: => Exp[T], => elsep: Exp[T]): Exp[T] =
  IfThenElse(cond, thenp, elsep)
```

However, looks can be deceiving. Behind the scenes, implicit conversions are taking care of reifying the sequencing. Making the implicit conversions explicit, we see what is really going on:

```
reflect(IfThenElse(cond, reifyBlock(thenp), reifyBlock(elsep)))
```

The details of code generation for JavaScript do not provide any further insights. For completeness, the generated code looks as follows:

```
var x1 = {"name" : "kim","age" : 20}
var x2 = (x1.age = 21)
if (x1.age >= 21) {
  var x3 = "beer"
} else {
  var x3 = "milk"
}
```

It is easy to complete the example by generating an HTML page that can be used to run the code in a web browser. A more sophisticated JavaScript embedding with additional features was studied by Kossakowski et al. [25].

6.3 Structuring DSL Implementations

Since embedded languages are just libraries, we can use all Scala modularity facilities (classes, traits, objects) to structure DSLs as components. Often it makes sense to encapsulate implementation details of a DSL embedding. For example, we may want to keep functionality like `reflect/reify` internal to the DSL implementation and invisible from actual DSL programs. Moreover, we may want to keep the deep embedding representation inaccessible from the DSL programs, too: If a program can observe its own representation, on-the-fly optimizations while building up the deep embedding may no longer be sound because the program could base its computation on a particular term representation [55].

Thus, a common approach for structuring DSL embeddings (as popularized by the LMS [42] and Delite frameworks [45], and inspired by earlier work on *finally tagless*

[8] or *polymorphic* embedding of DSLs [20]) is to separate the DSL *interface* from its *implementation*.

We will consider our probabilistic DSL from Section 2.1 as example. We first define the DSL interface as trait `ProbDSL` that extends a trait `Base`, which we may take to contain some infrastructure common to multiple embedded DSLs:

```
trait Base extends EmbeddedControls {
  type Rep[T]
}
trait ProbDSL extends Base {
  type Prob = Double
  def flip(p: Prob): Rep[Boolean]
  def binomial(p: Prob, n: Rep[Int]): Rep[Int]
  def always[A](e: Rep[A]): Rep[A]
}
```

Trait `ProbDSL` contains only abstract methods, and instead of fixing a concrete representation for DSL expressions, we use the abstract type constructor `Rep`, which is defined in trait `Base`. Trait `Base` also inherits from `EmbeddedControls` to make the usual virtualization hooks available to its subclasses.

We go on to define a trait `BaseExp`, which will contain the core support for our deep embedding:

```
trait BaseExp extends Base {
  type Rep[T] = Exp[T]
  abstract class Exp[T]
  abstract class Def[T]
  abstract class Block[T]
  def reflect[T](d: Def[T]): Exp[T] = ...
  def reifyBlock[T](e: => Exp[T]): Block[T] = ...
}
```

With `BaseExp` at hand, we are ready to define `ProbDSLExp`, which contains the deep embedding of our probabilistic DSL expressions:

```
trait ProbDSLExp extends BaseExp with ProbDSL  {
  case class Flip(p: Prob) extends Exp[Boolean]
  case class Binomial(p: Prob, n: Exp[Int]) extends Exp[Int]
  case class Always[A](e: Exp[A]) extends Exp[A]
  def flip(p: Prob): Exp[Boolean]               = reflect(Flip(p))
  def binomial(p: Prob, n: Exp[Int]): Exp[Int]  = reflect(Binomial(p, n))
  def always[A](e: Exp[A])                      = reflect(Always(e))
  def apply: Exp[Any]
}
```

Within the implementation hierarchy, `Rep[T]` is fixed to `Exp[T]`, but this fact is not observable from DSL programs that have no knowledge about `ProbDSLExp`.

To write and run an actual DSL programs, we first extend the DSL interface `ProbDSL`

```
trait MyProgram extends ProbDSL {
  def apply = {
    val bias = flip(0.3)
    if (bias) flip(0.6) else flip(0.5)
  }
```

```
}
```

and then create an object that mixes in the DSL program with the DSL implementation `ProbDSLExp`:

```
new MyProgram with ProbDSLExp
```

This way, all bindings from `ProbDSL`, but not those from `ProbDSLExp`, are available to the program inside `apply`.

Finally, on this object we can invoke a DSL specific method (e.g. `result`) that will run the `apply` method defined on the DSL program to obtain a deep embedding, and perform DSL specific interpretation or compilation of the deep representation.

6.4 DSL Scopes: Reducing Boilerplate by Relaxing Hygiene

The approach of structuring DSLs into interface and implementation components comes with a number of benefits but also with a drawback: it imposes some boilerplate not only on the DSL developer but also on the DSL user. Extending traits and creating objects in a certain way just to define a little DSL program may be asking too much.

Scala-Virtualized therefore introduces the concept of DSL scopes: Instead of defining traits and objects explicitly, DSL users can just write

```
ProbDSL {
  flip(0.5) && flip(0.5)
}
```

and the compiler will desugar this expression into

```
class DSLprog extends ProbDSL {
  def apply = {
    flip(0.5) && flip(0.5)
  }
}
(new DSLprog with ProbDSLExp).result
```

It is instructive to compare DSL scopes with regular block scopes in Scala. In plain Scala, if we define a function with a by-name argument

```
def MyBlock[A](body: => A): A
```

We can invoke it like this:

```
MyBlock {
  ...
}
```

The same approach also works for functions with multiple parameter lists, and general function arguments (a by-name argument is conceptually a zero-argument function). For example, we can define a function `using` to close a file handler or other managed resource after using:

```
def using[A](x: Closeable[A])(body: x => A): A = try body(x) finally x.close
```

A possible use could look like this:

```
using(new File("output.txt")) { f =>
  f.write("hello")
}
```

We might be tempted to try and use this mechanism to implement our desired DSL syntax:

```
ProbDSL {
  flip(0.5) && flip(0.5)
}
```

However, in plain Scala this would not work because scopes, in general, are hygienic. If we define `ProbDSL` as a method with a by-name parameter, there is no way for `ProbDSL` to modify the bindings available at the call-site. We thus need a facility with relaxed hygiene conditions.

Scala-Virtualized recognizes and translates method calls that return an object of a marker class `Scope`, defined in `EmbeddedControls`:

```
class Scope[Interface, Implementation, Result](body: => Result)
```

To implement a DSL scope facility for a given DSL, DSL authors just need to provide a method with an appropriate name and a `Scope` return type. The type parameters to `Scope` determine the traits used as interface and implementation, and the result type of the scope, respectively.

```
def ProbDSL[R](b: => R) = new Scope[ProbDSL, ProbDSLExp, R](b)
```

This definition enables the desugaring of `ProbDSL { .. }` blocks as shown above.

6.5 Example: Compiling and Running Embedded Code (Staging)

As a final example, we show a small embedded language that does not add any new functionality on top of Scala, but just implements existing Scala functionality in a more efficient way. More precisely, it removes abstraction overhead by constructing a deep embedding that is simpler than the original program, relying on deep linguistic reuse to translate away features such as first class functions or generic types. This is an example of multi-stage programming (staging) [56], but without the usual quasi-quotation syntax and with a restricted object language.

High-level numeric algorithms such as generic matrix multiplication are quite slow in Scala because primitive values passed to generic methods are boxed unless all generic type parameters on the call path are annotated as `@specialized`. Furthermore, for comprehensions are translated into method calls, which entails allocating and garbage collecting a significant amount of closures. In the same vein, `Range` objects are allocated simply to hold the start and end of the iteration range.

Consider the following naive implementation of matrix multiplication:

```
def multGeneric[T:Numeric:Manifest](m: Matrix[T], n: Matrix[T]) = {
  val p = new Matrix[T](m.rows, n.cols)
  for (i <- 0 until m.rows) {
    for (j <- 0 until n.cols) {
      for (k <- 0 until n.rows) {
        p(i, j) += m(i, k) * n(k, j)
  }}}
  p
}
```

By implementing `Matrix` in terms of `Exp[Array[A]]` internally, the deep embedding does not contain any trace of a matrix abstraction, just low-level operations on arrays.

```
class Matrix[A: Manifest](val rows: Exp[Int], val cols: Exp[Int]) {
  private val arr: Exp[Array[A]]                 = ArrayNew[A](rows * cols)
  def apply(i: Exp[Int], j: Exp[Int]): Exp[A]    = arr(i*cols + j)
  def update(i: Exp[Int], j: Exp[Int], e: Exp[A]) = { arr(i*cols + j) = e }
}
```

Similarly, we provide an implementation of for loops that strips away the abstraction overhead of range objects and higher-order functions:

```
def infix_until(x: Exp[Int], y: Exp[Int]) = RangeExp(x,y)
case class RangeExp(val start: Exp[Int], val end: Exp[Int]) {
  def foreach(f: Exp[Int] => Exp[Unit]): Exp[Unit] = {
    var i = start
    while (i < end) { // generates deep embedding of while loop
      f(i)
      i += 1
    }}}
```

A for comprehension like `for (k <- 0 until n.rows)` will be interpreted as method calls `infix_until(0,n.rows).foreach(k => ...)`. The implementation of `foreach` given above takes a function parameter of type `Exp[Int] => Exp[Unit]`, which means that the call inside the while loop will always be inlined. The deep embedding will not contain any trace of closures or higher-order functions, just plain and simple while loops. It is important to notice, however, that we can leverage the full expressive power of higher-order functions while composing the deep embedding representation. For example, we could easily define other higher-order functions like `map` or `count` that can use `foreach` internally. Thus, we obtain deep linguistic reuse of higher-order functions: we profit from their expressive power but do not need to pay the price on the deep embedding level.

By splicing in matrices and ranges that perform reification, in addition to intercepting array update, multiplication, and addition in the usual way, our naive program now generates a representation of itself.

The reification of array creation provides a nice use case for type manifests (see Section 7.1). From our representation that uses generic types, we can emit code that is instantiated to concrete types:

```
def ArrayNew[T: Manifest](n: Exp[Int]): Exp[Array[T]] = ArrayNewOp[T](n, manifest[T])
case class ArrayNewOp[T](n: Exp[Int], tp: Manifest[T]) extends Def[Array[T]]
def emitNode[T](s: Sym[T], d: Def[T]): Unit = d match {
  case ArrayNewOp(n, tp) =>
    emitValDef(s, "new Array[" + tp + "](" + n + ")")
  ...
```

Here, the standard library's `manifest[T]` function provides easy access to the implicit manifest that is in scope due to the `Manifest` context bound on `T`. The net effect is that we remove generic dispatch and instantiate all generic types, another case of deep linguistic reuse. We get all of these optimizations essentially for free due to the way the embedding is set up. Our minimal "optimizer" yields the following program:

```
var x27 = 500 * 500
var x28 = new Array[Double](x27)
var x29: Int = 0
while (x29 < 500) {
```

```
  var x30: Int = 0
  while (x30 < 500) {
    var x31: Int = 0
    while (x31 < 100) {
      ...
      x31 += 1
    }
    var x46 = ()
    x46
    x30 += 1
  }
  var x47 = ()
  x47
  x29 += 1
}
```

Finally, we simply instantiate a Scala compiler instance and use its API to generate bytecode for the optimized Scala code that we generated, and run the resulting program. Informal benchmarks indicate these simple optimizations result in a 20x speedup for multiplying two random 500 x 100 and 100 x 500 matrixes of doubles: the polymorphic multiplication takes 1.4s, when specializing to matrices of primitive `doubles` the run time is reduced to 1s, and the staged implementation reduces this time further by a factor of 20:

generic:	specialized:	staged:
2.691s	1.062s	0.088s
1.400s	1.228s	0.058s
1.464s	1.076s	0.055s
1.359s	1.030s	0.054s
1.244s	1.076s	0.056s

This is just the tip of the iceberg of the optimizations enabled by our approach. Using Lightweight Modular Staging [43] and Delite [6,28] we can add parallelism, loop fusion, code motion and other advanced optimizations. We have a full program representation, so we can do any analysis or transformation a regular compiler can do. In addition, we profit from the fact that the deep embedding is simpler than the original program, because lots of abstraction overhead has already been removed while the deep representation was constructed [44,40].

7 Debugging Support for DSLs

Debugging is an important concern for developing DSLs beyond proof-of-concepts. The main difficulty is issuing error messages in terms of the DSL, not in terms of its host language embedding. This section surveys the debugging support in Scala-Virtualized, in particular an extension of Scala's implicit parameters that enables DSLs to access source file and line number information (Section 7.2). Section 7.1 sets the stage by reviewing Scala's implicits and the use of manifests to access static type information at

runtime, after which the source info facility was modeled. To complete the debugging subject, Section 7.3 reviews Scala's support for influencing type error messages at compile time.

7.1 Virtualizing Static Type Information

Scala's implicits [11] provide a convenient way of deriving run-time information from static types. When the last argument list of a method is marked as **implicit**, a call to this method need not specify its actual arguments. For each missing implicit argument, the compiler will (statically) determine the (unique) implicit value of the correct type in order to complete the method call. The **implicit** keyword is used to mark regular value definitions as potential implicit arguments. By overriding a virtualized language feature to include certain implicit parameters we can require additional static information or predicate virtualization on some static condition.

Certain types of implicit values are treated specially by the compiler: when no user-defined implicit value of the expected type can be found, the compiler synthesizes the value itself. In standard Scala, manifests, which provide a run-time representation of static types, are the only implicit values that are treated this way [13].

As an example of manifests, consider the following polymorphic method that requires a manifest for its type parameter `T`:

```
def m[T](x: T)(implicit m: Manifest[T]) = ...
```

When this method is called at type `String`, and assuming there is no implicit value of type `Manifest[String]` in scope, the compiler will synthesize a factory call that generates a run-time representation of the class `String`, like this:

```
reflect.Manifest.classType(classOf[String])
```

The main use of manifests in the context of embedded DSLs is to preserve information necessary for generating efficient specialized code in those cases where polymorphic types are unknown at compile time (e.g., to generate code that is specialized to arrays of a primitive type, even though the object program is constructed using generic types).

7.2 Virtualizing Static Source Information

Scala-Virtualized extends the idea of `Manifest` and introduces `SourceContext` to provide run-time information about the static *source code* context. Implicit source contexts reify static source information, such as the current file and line number, which is otherwise lost after the program is compiled. The idea is for a method to declare an implicit parameter of type `SourceContext`:

```
def m[T](x: T)(implicit pos: SourceContext) = ...
```

Inside the method `m`, the source context of its invocation, i.e., the file name, line number, character offset, etc., is available as `pos`. Like manifests, source contexts are generated by the compiler on demand.

Implicit `SourceContext` objects are chained to reflect the static call path. Thus they can provide source information that is impossible to recover from exception stack traces, say. Consider the following example:

```
def m()(implicit pos: SourceContext) = ...
def outer()(implicit outerPos: SourceContext) =
  () => m()
val fun = outer()
fun() // invoke closure
```

Here, the method `outer` returns a closure which invokes method `m`. Since `m` has an implicit `SourceContext` parameter, the compiler generates an object containing source information for the invocation of `m` inside the closure. The compiler will not only pass the `SourceContext` corresponding to the current invocation but also the `outerPos` context as the parent of the current `SourceContext`. As a result, when invoking the closure inside `m` the chain of source contexts remains available. Both inside `m` as well as inside the closure, the static source context of the closure is known. This means that even if the closure escapes its static creation site, when the closure is invoked, the source context of its creation site can be recovered. Stack traces would not be able to expose this information since it can not be recovered from the dynamic call stack.

When lifting DSL constructs, we typically lose source information; expression trees no longer contain line numbers and variable names that correspond to the original source. This can be problematic if the DSL performs checks on the deep embedding. In that case, failing checks should produce meaningful error messages that refer back to the location in the source that contains the error. To fix this, methods used in the embedding of a DSL can be augmented as follows:

```
implicit class selectOps(self: Exp[_ <: Record])(implicit loc: SourceContext) {
  def selectDynamic[T](n: String): Exp[T] =
    Select(self, n)(loc)
}
```

The above method can be used to provide a `Select` operation on expression trees (e.g., for generating SQL expressions). The implicit is applied whenever a field is selected on an expression tree whose result type extends `Record`. To improve debugging of problematic field selections we add an implicit `SourceContext` parameter. As a result, whenever the `selectOps` method is called, the `loc` argument describes the *invocation site*. The compiler automatically generates this `loc` object for each invocation; it contains source information specific to the static invocation site. This source information is accessible through methods such as `fileName`, `line`, and `charOffset`.

The `SourceContext` object can then be passed to the data constructors of the embedding. In the above example, we are passing `loc` to the constructor of `Select`. This way, each `Select` node is equipped with source location information which can be used when processing the IR subsequently.

For example, when processing erroneous SQL queries, the DSL generator can output error messages that contain precise source location information. Consider what happens if an invalid query expression is submitted to a database. For example, a query might try to access a column that doesn't exists in a database table. This typically leads to an exception, such that the stack trace points to the expression which submitted the query to the database. Moreover, the information about which elements of the DSL program were involved in the problematic situation is lost. However, in the above case what the user would really like to know is where the queried table was *declared in the DSL program*; this would allow pin-pointing the error much easier, by giving the user a chance to check the correctness of their declarations.

Using implicit `SourceContext` parameters this information can be provided in DSL-specific error messages. To that purpose, we extend the DSL constructs for which we would like to have source information, in this case, the table constructor:

```
case class Table[R <: Record](name: String)(implicit val loc: SourceContext)
  extends Exp[List[R]]
```

Subsequently, we can make use of this source information when handling run-time exceptions, and provide DSL-specific error messages pointing to the precise source location of elements of our DSL, in this case, table declarations:

```
case e: Exception => expr match {
  case ListSelect(table @ Table(name), _) =>
      println("error in query on table " + name +
        " declared at " + table.loc.line + ": " + e)
```

7.3 Static Error Checking

Simple annotations placed on types and constructs of the library that should not be visible to the user can be used for adapting error messages of the compiler. Such annotations can already improve the user experience substantially; at the same time the approach places only a small burden on the library author.

In fact, this approach is already finding its way into Scala's standard collections library. Scala's collections [32] use implicit parameters to support operations on collections that are polymorphic in the type of the resulting collection. These implicit parameters should not be visible to the application developer. However, in previous Scala versions, error messages when using collections incorrectly could refer to these implicits. In recent versions of Scala, a lightweight mechanism has been added to adapt error messages involving implicits: by adding an annotation to the type of the implicit parameter, a custom error message is emitted when no implicit value of that type can be found.

For instance, immutable maps define a `transform` method that applies a function to the key/value pairs stored in the map resulting in a collection containing the transformed values:

```
def transform[C, That](f: (A, B) => C)(implicit bf:
                    CanBuildFrom[This, (A, C), That]): That
```

This function transforms all the values of mappings contained in the current map with function `f`. Here, `This` is the type of the actual map implementation and `That`, the type of the updated map. The implicit parameter ensures that there is a builder factory that can be used to construct a collection of type `That` given a collection of type `This` and elements of type `(A, C)`.

Actual implicit arguments passed to transform should not be visible to the application developer. However, wrong uses of maps may result in the compiler not finding concrete implicit arguments; this would result in confusing error messages. Error messages involving type `CanBuildFrom` are improved using a type annotation:

```
@implicitNotFound(msg = "Cannot construct a collection of
  type ${To} with elements of type ${Elem} based on a
  collection of type ${From}.")
trait CanBuildFrom[-From, -Elem, +To] { ... }
```

The `implicitNotFound` annotation is understood by the implicit search mechanism in Scala's type checker. Whenever the type checker is unable to determine an implicit argument of type `CanBuildFrom`, the compiler emits the (interpolated) error message specified as the argument of the `implicitNotFound` annotation. Thereby, a low-level implicit-not-found error message is transformed to only mention the types `From`, `Elem`, and `To`, which correspond to types occurring in user programs.

8 Related Work

Embedded languages have a long history. Early work by Landin recognized that "most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things" and proposed to have many different languages, i.e. sets of primitive operations for different problem domains, in one general-purpose language [27]. Hudak introduced the concept of embedding DSLs as pure libraries [21, 22]. Steele proposed the idea of "growing" a language by adding domain specific language extensions, again in the form of libraries [51]. The concept of linguistic reuse goes back to Krishnamurthi [26]. Language virtualization was introduced by Chafi et al. [9]. The idea of representing an embedded language abstractly as a set of available methods was introduced as finally tagless embedding by Carette et al. [8] and as polymorphic embedding by Hofer et al. [20], going back to much earlier work by Reynolds [39].

In contrast to virtualization, there are many other language extension mechanisms. Some are external, like preprocessors or generators, and some use built-in meta-language facilities, such as Template Haskell [47], Metaborg [5] and SugarJ [16]. We refer to [15] for an overview.

8.1 Existing Cases of Virtualization in Other Languages

Many languages virtualize a certain set of features using patterns that are similar to the "everything is a method call" approach we applied in Scala-Virtualized. We review some of these cases next.

C++: Expression templates [61] are a way of reifying program expressions in C++. Making crucial use of operator overloading, templates are used to represent high-level operations. An extensive range of libraries has been built using expression templates to represent complex computations and perform symbolic optimizations at compile time, during template specialization. Operator overloading enables expression templates to look like regular arithmetic operations, the approach is thus very similar to the pattern followed in Scala-Virtualized. In C++, just like in plain Scala, it is not possible to override built-ins like if-then-else, therefore expression templates need to use different and less natural syntax to implement conditionals or, in general, expressions that are composed out of multiple statements. Therefore, in many cases only isolated arithmetic expressions are reified as templates which limits the extent and power of applicable optimizations. Bassetti et al. review these and other challenges in obtaining performance on par with low-level C or Fortran code from expression template approaches [4]. Examples of successful expression template packages are the Matrix Template Library [49], POOMA [23], ROSE [12] and the portable expression template engine PETE [10]. Many of these packages contain reusable generic components that are applicable beyond the immediate use-case of the particular library.

Haskell: Since monads [29,63] are a ubiquitous pattern in Haskell programming [36], Haskell provides syntactic support for monadic composition in form of its 'do'-notation. The 'do'-notation can be used with any type that implements the monadic `return` and `>>=` operations. In our view, this is a case of virtualizing Haskell's imperative sublanguage, which consists of variable binding and sequential statement composition, by defining syntax that desugars into method calls. Haskell is a popular host language for embedded DSLs. To give just one example that uses techniques related to this paper, the Feldspar DSL [3] combines a shallow front-end with a deep embedding back-end and makes clever use of deep linguistic reuse. In particular, functions in the shallow embedding provide automatic fusion of array/vector operations without any further work on the deep embedding level.

F#: Like Haskell's 'do'-notation, computation expressions [54] in F# can be seen as a general monadic syntax for F#. Just like Scala's for-comprehensions, a computation expression is translated to method calls on objects. In contrast to Scala, in F# a computation expression is required to identify one particular object/monad (the "builder" object) which needs to provide a well-defined set of methods that are used as invocation targets of the translation. Like for-comprehensions, computation expressions are restricted to a small sublanguage. Scala-Virtualized is more expressive, since it virtualizes also expressions like pattern matching. Compared to Haskell, F# generalizes monadic computations and monadic containers using *delayed computations* [35]. This generalization enables expressions of impure languages like F# , such as while-loops and try-with exception-handling, inside computation expressions.

Racket: Racket is a dialect of Scheme that allows programmers to change almost all aspects of the language using a carefully designed revision of a Scheme-style macro system. Rackets enables "languages as libraries" [58], i.e. individual modules of a program can be implemented in different languages, and language implementations have full control over the syntax and semantics of the module, including lexicographic and parsed notation, static semantics, module linking, and optimizations. Typed Scheme [57] can be implemented entirely as macros in Racket. This flexibility goes considerably beyond what Scala-Virtualized offers, but the starting point is very different as well (Scheme vs Scala).

8.2 Virtualization and Reflection

The ability of a programming language to represent its (meta) programs as object programs is called reflection [50]. In other words, reflection enables a program to inspect and reason about itself.

Virtualization can be seen as a (static) dual of (dynamic) reflection: Where a reflective language allows programs to inspect language elements they are composed of, a virtualizable language allows programs to give language elements new meaning. In a reflective language programs can use information obtained by reflection to trigger a certain behavior. In a virtualizable language the language elements can be customized to trigger the behavior directly within programs.

Using virtualization to create expression trees is a mechanism to achieve reification of programs, and reification is an important mechanism in reflective languages. On the other hand, an accessible program representation alone, such as provided by a lifting mechanism that reifies expression trees, is not always sufficient. First, reified expression trees can contain arbitrary host language expressions, not just those that are also part

of the embedded DSL. Second, in many cases it is desirable to freely mix lifted DSL code and non-lifted host language code. We give an example in the section on Scala macros below (Section 8.3).

8.3 Scala Macros

Recent versions of Scala have added a macro facility to the language [7]. We view macros and virtualization as complementary technologies with many possible synergies. A key benefit of macros is the ability to raise domain specific error messages at compile time, which means, for example, that these error message can show up in an IDE as squiggly underlines just like regular Scala errors. Virtualized methods could be implemented as macros, and thus perform extended domain-specific checking at compile time. Conceptually, entire domain-specific transformations could be done at compile time, although there are certain limits posed by separate compilation requirements and related open-world assumptions of the Scala compiler.

Macros can be used to reify expression trees directly, and macros receive reified expression trees as their arguments. Compared to deep embeddings constructed using virtualization, which have fine-grained control over which parts of a DSL programs are reified into a deep embedding and which parts are not, the macro approach is more coarse-grained. For example, macros that take functions as their arguments will receive these functions as values of type `Exp[A=>B]`, which may represent a closure literal but might also represent an identifier if the function is defined elsewhere. It is instructive to compare this to Section 6.5, where we represented DSL functions as values of type `Exp[A]=>Exp[B]`, i.e. as regular Scala functions that compute a reified representation given a reified argument, without reifying functions in the deep embedding. The first (macro) version will need to be interpreted fully, the second one may directly perform arbitrary computations while building the expression tree of the result. Essentially, the difference is in shallow vs. deep reuse of functions. Scala macros also have no special support for maintaining evaluation order (see Section 6.1) when splicing program fragments but use customary, context-free expansion (similar to quasi-quotation).

It would be highly desirable to implement some of the virtualization logic itself as macros, for example DSL scopes (Section 6.4). However, in the current implementation, Scala macros require macro invocations to type check before macro expansion. Thus, we could not easily express the semantics of DSL scopes, which rely on relaxed hygiene to introduce bindings for DSL methods into the argument block. Support for untyped macros is planned for some future Scala release but the design has not been finalized.

8.4 Built with Scala-Virtualized

Scala-Virtualized has been in used successfully in several DSL projects. Lightweight Modular Staging (LMS) [42,43,44] is a set of techniques and a core compiler framework for building embedded DSLs in Scala-Virtualized. The key idea of LMS is to use combinations of shallow and deep embeddings for explicit multi-stage programming [56], but without the usual syntactic quasi-quotations (similar to the example in Section 6.5). When building DSLs using LMS, developers can rely on modules of core functionality that mirror parts of the Scala language and standard library. Deep

embeddings can be built by extending a common graph-based IR (intermediate representation) that captures dependencies and effect information in a generic way. LMS also provides a wide range of generic compiler optimizations on the IR level, including common subexpression elimination, algebraic rewrites, code motion, or loop fusion.

Delite [6, 45, 28] is a research project from Stanford University's Pervasive Parallelism Laboratory (PPL) that makes key use of Scala-Virtualized and LMS to build a framework and runtime for parallel embedded DSLs. To simplify the construction of high-performance, highly productive DSLs, Delite provides:

- code generators for Scala, C++ and CUDA,
- built-in parallel execution patterns,
- optimizers for parallel code,
- scheduling and runtime support for heterogeneous hardware.

OptiML [52] is a DSL for machine learning developed using Scala-Virtualized, LMS and Delite. OptiML programs can be compiled for a variety of parallel hardware platforms, including CMPs (chip multi-processors), GPUs (by automatically generating CUDA code), and eventually even FPGAs and other specialized accelerators. Cluster support is a topic of ongoing work. Furthermore, compilation employs aggressive domain-specific optimizations, resulting in high-performance generated code which outperforms parallel MATLAB on many common ML kernels.

OptiML makes it easy to express iterative statistical inference problems. Most of these problems are expressed using dense or sparse linear algebra operations which can be parallelized using a large number of fine-grained map-reduce operators. OptiML programs make use of three fundamental data types, Vector, Matrix, and Graph, which support all of the standard linear algebra operations used in most ML algorithms. These data types are polymorphic and are compiled to efficient code leveraging BLAS or GPU support if they are used with scalar values.

Other DSLs build using Scala-Virtualized and LMS include SIQ, StagedSAC and Jet. Scala Integrated Query (SIQ) [62] compiles an embedded subset of Scala into SQL for execution in a DBMS. Advantages of SIQ over SQL are type safety, familiar syntax, and better performance for complex queries by avoiding avalanches of SQL queries. StagedSAC [60] is an embedded DSL for functional programming with multidimensional arrays, modeled after the language SAC [46]. A key feature is a domain-specific type inference pass that pre-computes array shapes. Jet [1] is a DSL that provides a collection-like interface for "BigData" processing on clusters and can generate code to run on a variety of cluster programming frameworks.

9 Conclusion

In this paper, we have presented Scala-Virtualized, a set of small extensions to the Scala language to provide even better support for hosting embedded DSLs. Scala-Virtualized extends concepts already present in Scala and in many other languages; most importantly the idea of virtualizing certain language features by defining them as method calls, so that they can be redefined within the language. Scala-Virtualized redefines most of Scala's expression sub-language in this way, enabling DSL implementations to give domain-specific meaning to core language constructs such as conditionals, pattern matching, etc. A key use for this facility is in reifying DSL expressions into a data structure representation, i.e. a deep embedding. By redefining e.g. conditionals to construct

an expression tree, a deeply embedded DSL can still use the familiar Scala syntax for conditionals: On the surface, an `if` in the DSL is just a Scala `if`, although it may have a different type and do something different. This means that using virtualization, deep embeddings can benefit from linguistic reuse in much the same way as shallow embeddings, for which linguistic reuse is automatic.

In addition to the obvious syntactic, shallow, reuse, we have also presented examples of what we call deep linguistic reuse. By combining shallow and deep components, DSLs can often implement certain features such as higher order functions or generic types completely at the shallow embedding level, resulting in a deep embedding that is simpler to analyze and faster to execute.

Scala-Virtualized has been used successfully in several projects to build DSLs which combine the lightweight appearance of a shallow embedding with the performance characteristics and flexibility of a deep embedding. We believe that the techniques we applied in the context of the Scala language are generic, and could be applied to other expressive languages as well.

Acknowledgments

The authors would like to thank Arvind Sujeeth, Hassan Chafi, Kevin Brown, HyoukJoong Lee, Zach DeVito, Kunle Olukotun, Christopher Vogt, Vlad Ureche, Grzegorz Kossakowski, Stefan Ackermann, Vojin Jovanovic, Manohar Jonnalagedda, Sandro Stucki and Julien Richard-Foy. The authors would also like to thank the anonymous PEPM'12 and HOSC reviewers.

References

1. S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded dsl for high performance big data processing. BigData, 2012. `http://infoscience.epfl.ch/record/181673/files/paper.pdf`.
2. J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.
3. E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of feldspar an embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.
4. F. Bassetti, K. Davis, and D. J. Quinlan. C++ expression templates performance issues in scientific computing. In *IPPS/SPDP*, pages 635–639, 1998.
5. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 365–383. ACM, 2004.
6. K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, October 2011.
7. E. Burmako and M. Odersky. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.
8. J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
9. H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.
10. J. Crotinger, S. Haney, S. Smith, and S. Karmesin. PETE: The portable expression template engine. *Dr. Dobb's J.*, Oct. 1999.
11. B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 341–360. ACM, 2010.

12. K. Davis and D. J. Quinlan. Rose: An optimizing transformation system for c++ array-class libraries. In *ECOOP Workshops*, pages 452–453, 1998.
13. G. Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, Lausanne, 2011.
14. B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, pages 273–298, 2007.
15. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the 12th Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012.
16. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 391–406. ACM, 2011.
17. A. Filinski. Representing monads. In *POPL*, pages 446–457, 1994.
18. A. Filinski. Monads in action. In *POPL*, pages 483–494, 2010.
19. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.
20. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. GPCE, 2008.
21. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
22. P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, 1998.
23. S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array design and expression evaluation in pooma ii. In *ISCOPE*, pages 231–238, 1998.
24. O. Kiselyov and C. chieh Shan. Embedded probabilistic programming. In W. M. Taha, editor, *DSL*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2009.
25. G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. Javascript as an embedded dsl. In *ECOOP*, 2012.
26. S. Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.
27. P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.
28. H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.
29. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
30. A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html`.
31. A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *PEPM*, pages 117–120, 2012.
32. M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*, volume 4, pages 427–451, 2009.
33. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, Second Edition*. Artima Press, 2010.
34. M. Odersky and M. Zenger. Scalable component abstractions. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.
35. T. Petricek and D. Syme. Syntax matters: Writing abstract computations in F#. In *TFP*, 2012.
36. S. Peyton Jones [editor], J. Hughes [editor], L. Augustsson, D. Barton, B. Boutel, W. Burton, S. Fraser, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, feb 1999.
37. G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
38. N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

39. J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.
40. T. Rompf. *Lightweight Modular Staging and Embedded Compilers.* PhD thesis, IC, Lausanne, 2012.
41. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 317–328. ACM, 2009.
42. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In E. Visser and J. Järvi, editors, *GPCE*, pages 127–136. ACM, 2010.
43. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.
44. T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.
45. T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. In *DSL*, pages 93–117, 2011.
46. S.-B. Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.
47. T. Sheard and S. L. P. Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.
48. J. G. Siek. General purpose languages should be metalanguages. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 3–4, New York, NY, USA, 2010. ACM.
49. J. G. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, pages 59–70, 1998.
50. B. C. Smith. *Procedural reflection in programming languages.* PhD thesis, MIT, 1982.
51. G. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.
52. A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.
53. K. N. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, pages 160–169, 2006.
54. D. Syme, A. Granicz, and A. Cisternino. *Expert F#.* Apress, 2007.
55. W. Taha. A sound reduction semantics for untyped cbn multi-stage computation. or, the theory of metaml is non-trivial (extended abstract). In *PEPM*, pages 34–43, 2000.
56. W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
57. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In G. C. Necula and P. Wadler, editors, *POPL*, pages 395–406. ACM, 2008.
58. S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.
59. E. Torreborre. Specs: Software specifications for Scala, 2011.
60. V. Ureche, T. Rompf, A. K. Sujeeth, H. Chafi, and M. Odersky. Stagedsac: a case study in performance-oriented dsl development. In *PEPM*, pages 73–82, 2012.
61. T. L. Veldhuizen. Expression templates. 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
62. J. C. Vogt. Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany, 2011.
63. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.
64. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.