Tool Demo: Scala-Virtualized

Adriaan Moors Tiark Rompf Philipp Haller * Martin Odersky
EPFL
first.last@epfl.ch

Abstract

This paper describes Scala-Virtualized, which extends the Scala language and compiler with a small number of features that enable combining the benefits of shallow and deep embeddings of DSLs. We demonstrate our approach by showing how to embed three different domain-specific languages in Scala. Moreover, we summarize how others have been using our extended compiler in their own research and teaching. Supporting artifacts of our tool include web-based tutorials, nightly builds, and an Eclipse update site hosting an up-to-date version of the Scala IDE for Eclipse based on the Virtualized Scala compiler and standard library.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

1. Introduction

In his PEPM'10 invited talk "General Purpose Languages Should be Metalanguages" [15], Jeremy Siek eloquently made the case that general purpose languages should be designed to host embedded DSLs and concluded his talk saying "Our current languages are OK but not great". We have argued in a similar way and proposed the notion of *language virtualization* [3]. Just as 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 languages, each of which should feel as "real" as possible.

In this paper, we present Scala-Virtualized, our effort to improve the DSL hosting capabilities of Scala. Scala is a mature language that is seeing widespread adoption in industry. Scala-Virtualized is a suite of minimal extensions to Scala that is based on the same codebase and undergoing the same rigorous testing process as the main Scala distribution.

A main feature of Scala-Virtualized is that it allows to blend *shallow* and *deep* embedding of DSLs. In other words, we can write embedded DSL code with very low syntactic overhead and integrated tightly with the surrounding Scala code but nonetheless obtain an AST-like representation that is amenable to analysis and optimization. This makes Scala-Virtualized an attractive target platform for applying code generation and program transformation techniques to embedded programs, which is often difficult.

Implementing program transformation techniques beyond proof-of-concepts is a laborious task. Scala-Virtualized can provide a useful tool for researchers in the program transformation community to implement their techniques on embedded programs.

Plain Scala is used successfully for pure library-based DSLs, such as actors [8], parser combinators [10], and testing frameworks. The following features make Scala attractive for these DSLs:

- Infix operator syntax for method calls: Scala syntax is quite flexible, code can look (almost) like English sentences. Here is an example from the Specs[1] testing framework:

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

- For-comprehensions: Scala has no built-in for loops. Expressions of the form

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

  are desugared into method calls like

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

  where the class of `foo` defines the implementation and type signature of `map`. This allows DSLs to provide non-standard implementations of looping constructs.
- Implicit definitions and parameters (see Section 4.2). Implicit resolution enables logic programming on the type level [4]. DSLs can implement domain-specific type restrictions (e.g. through phantom types).
- Manifests (see Section 3). Run-time descriptors of static types enable DSLs to leverage type information, e.g. for code generation.

These features are sufficient for DSLs as pure libraries. In many cases, however, we require better performance, we need to verify additional program properties for safety, or we need to generate code for a different platform. In short, we need an accessible representation of embedded programs that we can analyze and transform in various ways. Scala-Virtualized takes the given ideas one step further by extending Scala in the following ways:

- Redefining most control structures (e.g. conditionals, variable definitions) in terms of method calls (see Section 2). This allows DSLs to override the meaning of core language features.
- Implicit source contexts (see Section 3) lift static source information such that it becomes part of the object program (e.g. to improve debugging, error messages, etc.)

Scala-Virtualized is actively used for a number of DSL projects [1, 13, 16, 17]. Obtaining an analyzable intermediate representation has allowed us and others to write embedded DSL programs

* EPFL and Stanford

[Copyright notice will appear here once 'preprint' option is removed.]

[1] `http://code.google.com/p/specs/`

that outperform handwritten C code, compile to GPUs, or query databases using generated SQL.

2. Everything is a Method Call

The overarching idea of embedded languages is that user-defined abstractions should be first class in a broad sense. User-defined abstractions should have the same rights and privileges as built-in abstractions. Scala-Virtualized redefines many built-in abstractions as method calls. In this way, the corresponding method definitions may be redefined by a DSL, just like any other method. Similar to the "finally tagless" [2] or polymorphic embedding [9] approach, and going back to an old idea of Reynolds [12], we represent 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, which is typically only provided by a deep embedding using explicit data constructors. To give a quick example, the expression

```
if (c) a else b
```

is defined as the method call

```
__ifThenElse(c,a,b)
```

This approach fits well with the overall Scala philosophy: for-comprehensions and parser combinators have been implemented like this for years now. Unlike approaches that lift host language expression trees 1:1 using a fixed set of (host language oriented) data types, the DSL implementor has control over which language constructs are lifted and which are not and the lifting is not tied to the host language grammar but may pick a representation that closer fits the DSL grammar.

3. Lift Static Information into Object Programs

Our approach for lifting static information into the object program is based on a single principle, namely *implicit parameters*. The basic idea of implicit parameters is simple: instead of providing an argument explicitly at the call site, the compiler automatically passes an implicit value which is either defined in the current scope or which is synthesized based on its type.

Scala-Virtualized includes two instantiations of this principle. The first mechanism provides run-time type descriptors via implicit parameters. The idea is that a method can demand that a run-time type descriptor, called a *manifest*, be available for certain polymorphic types (typically type parameters). For example, the following polymorphic method requires a manifest for its `T` type parameter:

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

Manifests are automatically generated by the Scala compiler (they are already available in plain Scala [6]). This means that wherever a method with an implicit manifest parameter is invoked, the compiler generates a manifest and passes it implicitly. Manifests are an extension of Scala's standard implicits [5]: they are not generated if there is already an implicit argument that matches the expected manifest type.[2]

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 primitive type, say, even though the object program is constructed using generic types).

The second principle mechanism for lifting static information into the object program is concerned with source locations. A fundamental problem of embedded programs is that static source information, such as line numbers, is typically lost at the point where the DSL program is executed, in particular when the embedding contains an intermediate staging step. The idea is that methods can demand a source context for their current invocation by declaring an implicit parameter of type `SourceLocation`:

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

Inside the method `m`, the source context of its invocation, i.e. file name, line number, character offset, etc. is available as `ctx`. Like manifests, source contexts are generated by the compiler. Just like manifests, they extend Scala's standard implicit mechanism: if there is already an implicit `SourceContext`, it is passed as the parent of the current source context (see Section 4.3).

In summary, manifests and source contexts allow capturing essential static information and making it available to embedded DSLs.

4. Putting Principles Into Action

This section contains further details and examples on how these principles are applied to support more flexible syntactic embedding, static checking and dynamic debugging in Scala-Virtualized.

4.1 Syntax

Scala's syntax is quite flexible as it is [7] and can accommodate many flavors of embedded languages easily. Complementary to flexibility, Scala's type system also allows enforcing certain restrictions. For example, it is desirable to restrict DSL expressions to a given grammar. Here is an example how adherence of DSL expressions to a context-free grammar ($a^n b^n$) can be enforced in the type system:

```
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
```

This can be done in plain vanilla Scala, too, but it is cumbersome. Scala-Virtualized makes it easier through the addition of infix methods, which provide a straightforward way of (re-)defining methods on objects outside of their class: here, `x a` is turned into `infix_a(x)` instead of the conventional `x.a`.

4.2 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 [11] 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,

[2] The Scala standard library contains a hierarchy of four manifest classes; see Section 7.5 of the Scala language specification.

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. `That` is 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. In Scala 2.8.1 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 ${To}.")
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.

4.3 Dynamic Debugging

When lifting DSL constructs, we typically lose source information. For example, the IR nodes 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 generated IR. In that case, failing checks should produce meaningful error messages that refer back to the location in the source that contains the error.

Virtualized Scala allows DSL authors to capture source information of DSL constructs, such as line numbers and method names, using implicit parameters of type `SourceLocation`. For this, methods used in the embedding of a DSL can be augmented as follows:

```
implicit def selectOps(self: Exp[_ <: Result])
  (implicit loc: SourceLocation) = new {
  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 `Result`. To improve debugging of problematic field selections we add an implicit `SourceLocation` 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 `SourceLocation` object can then be passed to the constructors of the IR. 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 data base. For example, a query might try to access a column that doesn't exists in a data base table. This typically leads to an exception, such that the stack trace points to the expression which submitted the query to the data base. 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 `SourceLocation` parameters this information can be provided in DSL-specific error messages. For this, we extend the DSL constructs for which we would like to have source information, in this case, the table constructor:

```
case class Table[Tuple <: Result](name: String)
  (implicit val loc: SourceLocation)
  extends Exp[List[Tuple]]
```

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)
```

Virtualized Scala provides a refinement of the `SourceLocation` type, called `SourceContext`. An implicit parameter of type `SourceContext` allows capturing source information that is impossible to recover from exception stack traces, say. Consider the following example:

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

Here, the `outer` method 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. However, since the parameter has type `SourceContext` as opposed to its super type `SourceLocation`, the compiler will not only pass the `SourceContext` corresponding to the current invocation but also the `outerCtx` context as the parent of the current `SourceContext`. As a result, when invoking the closure inside `m` this chain of source contexts is available and 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. In contrast, using only stack traces provided by exceptions, this information is lost since it is not available on the dynamic call stack.

5. Usage

There are two modes of using the "tool" that we are describing: virtualized Scala as a language, in which DSLs can be implemented efficiently in a certain way, and virtualized Scala as an ecosystem of tool support, user community, documentation, etcetera. The former is illustrated by the concrete embeddings that are outlined in the appendix. In this section we consider the ecosystem dimension.

By shallowly embedding the DSL, both the DSL implementer and the DSL user (who programs *in* the DSL) reap the benefits of Scala's toolchain: the general-purpose Scala compiler has been used in production for several years, the IDE support is stable and maturing steadily, there are several build systems, several books have been written, catering to various audiences, and so on.

We have discussed some of the challenges of tailoring this usage modality to a specific domain: how to generate (slightly) more descriptive error message, how to retain source information, etc. Many more interesting research questions remain, and we are actively pursuing several of them. One example is a type debugger, which visualizes the state of the type checker and how it got to a certain error. This information could then further be interpreted in a domain-specific way, or it could assist the DSL implementer when "debugging" the complex types involved in expressing a domain-specific type system.

6. Applications

Scala-Virtualized has been used extensively by students and researchers outside of our research group.

6.1 Delite

Delite [1, 14] is a research project from Stanford University's Pervasive Parallelism Laboratory (PPL). Delite is a compiler framework and runtime for parallel embedded domain-specific languages (DSLs). To enable the rapid construction of high performance, highly productive DSLs, Delite provides several facilities:

- Code generators for Scala, C++ and CUDA
- Built-in parallel execution patterns
- Optimizers for parallel code
- A DSL runtime for heterogeneous hardware

6.2 OptiML

OptiML [16] is a domain-specific language (DSL) for machine learning developed using Scala-Virtualized. It uses the building blocks provided by Delite (see above) to simplify its implementation. 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.

6.3 A DSL Course

Scala-Virtualized has been the compiler platform for a course on DSLs for performance and productivity at Stanford University.[3] This advanced undergraduate/graduate level course provides an in-depth introduction to developing high-performance DSLs in important scientific and engineering domains. As such, it is aimed both at (a) domain experts, that is, students who can define key domain-specific language elements that capture domain knowledge, and at (b) computer scientists who can implement these DSLs in Scala. During the second half of the course, students developed their own DSLs using the Delite DSL framework, which simplifies the process of implementing DSLs for parallel computation. This first edition of the course had 11 registered students and presented 7 projects.

The course material includes a complete set of lecture slides on (a) embedding DSLs using Scala-Virtualized, (b) the Delite DSL framework, (c) the OptiML DSL for Machine Learning, and (d) DSL projects of other research groups and/or industrial projects.[4] In addition to that, there is a "getting started" guide on the Delite web site,[5] as well as a complete tutorial on developing a small parallel DSL using Delite.

6.4 Scala Integrated Query (SIQ)

SIQ [17] 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.

7. Supporting Artifacts

These artifacts are available on the Scala-Virtualized wiki[6]:

- an introduction and a reference of the virtualized constructs
- nightly builds, published as Maven artifacts for build tools
- a Virtualized Scala version of the Scala Eclipse plugin
- a GitHub repository with various self-contained tutorials, including an Eclipse project and an sbt build file
- a short description of our development process (a dashboard of continuous integration results, links to the development and the stable branches of our GitHub repository)
- the mailing list where we can be reached

8. Discussion

Even though Scala's syntax is quite flexible, it has its limits: the syntax of the DSL cannot be completely arbitrary. A similar restriction applies for the type system. However, as mentioned in Section 4.2, implicit resolution can be used to do logic programming at the type level to a certain degree, as witnessed by the `CanBuildFrom` relation, which also illustrates how some type error messages can be customized (using the `@implicitNotFound` annotation). Other classic tricks, such as phantom types, provide further expressive power. Furthermore, Scala does not yet have an effect system, so that side-effects cannot be restricted mechanically, and correctness of the analyses performed by the embedding framework thus follows from unchecked adherence to conventions.

On the bright side, our approach combines the ease of implementation of a shallow embedding with the performance characteristics of deep embedding. Scala's type system can enforce several common domain-specific invariants, and it can be used to hide several implementation aspects of the DSL from the DSL user, and even the DSL implementer.

In short, we promise no silver bullet, but a robust and efficient, though pragmatic, approach that has worked well for us and others, and that is steadily getting both more robust and more flexible.

9. Acknowledgments

The authors would like to thank Arvind Sujeeth, Hassan Chafi, Kevin Brown, HyoukJoong Lee, Zach DeVito, Kunle Olukotun, Christopher Vogt, Grzegorz Kossakowski and Nada Amin.

[3] CS442 was offered by Stanford's PPL for the first time in Spring 2011.

[4] Some of the slides are only accessible to Stanford students and faculty.

[5] See `http://stanford-ppl.github.com/Delite/`.

[6] `https://github.com/TiarkRompf/scala-virtualized/wiki`

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA/SPLASH'10*, 2010.

[6] G. Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, Lausanne, 2011. URL `http://library.epfl.ch/theses/?nr=5007`.

[7] G. Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, Lausanne, 2011.

[8] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[9] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. GPCE, 2008.

[10] 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`.

[11] 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.

[12] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.

[13] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. GPCE, 2010.

[14] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Oderksy, and K. Olukotun. Building-blocks for performance oriented DSLs. *Electronic Proceedings in Theoretical Computer Science*, 2011.

[15] 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. doi: http://doi.acm.org/10.1145/1706356.1706358. URL `http://doi.acm.org/10.1145/1706356.1706358`.

[16] 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.

[17] J. C. Vogt. Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany, 2011.

A. Proposed Demo

We propose a tutorial-style demo that introduces Virtualized Scala by example, from the ground up. The full source for the tutorial is available on GitHub[7].

A.1 Language Virtualization in Scala

The embedding of domain specific languages in Scala is based on a simple principle: the domain program looks like it's written in its own language with its own syntax, but the domain program is actually just a plain Scala program, where "everything is a method call".

Applying this principle naively, you end up with a "shallow embedding": DSL programs are just thinly veiled Scala programs. By implementing the methods that make up the DSL so that they build a representation of the domain program, the main advantage of a "deep embedding" can be recovered, without the associated implementation cost. As the DSL programs is represented explicitly, it can be analyzed (and thus optimized) by the DSL implementation. In other words, we want to shallowly embed our domain programs and optimize them too.

The shallow embedding is enabled by Scala's flexible syntax. The deep embedding relies on lightweight modular staging [13], which uses type information to drive staging. For the purpose of the demo it suffices to think of staging as `delaying execution` of programs by turning them into their own representation, which can then first be analyzed and then executed. This representation is usually structured as an abstract syntax tree.

The virtualized version of the Scala compiler allows implementers of domain-specific languages to reuse as much of Scala (the host language) as possible (its syntax, type system, module system,...), without sacrificing efficient execution of the domain program. The latter is enabled by what we call "language virtualization": Scala's built-in constructs are exposed as method definitions that can be overridden by the DSL implementer in order to construct a representation of the programs written in the DSL (the "domain programs").

The essential difference between Virtualized Scala and plain vanilla Scala is that more of a Scala program is expressed in terms of method calls. Concretely, an expression such as `if(c) a else b` is translated into a method call `__ifThenElse(c, a, b)`. By providing its own implementation of this method, the DSL can have it generate an AST for this part of the domain program, which can thus further be analyzed and optimized by the DSL implementation. When no alternative implementation is provided, the if-then-else has the usual semantics.

For example, we could change `if` to print its condition and return the then-branch, discarding the else-branch:

```
scala > def __ifThenElse[T](cond: => Boolean,
                thenp: => T, elsep: => T): T
                = {println("if: "+cond); thenp}
__ifThenElse: [T](cond: => Boolean,
                  thenp: => T, elsep: => T)T

scala > if(false) 1 else 2
// virtualized to: `__ifThenElse(false, 1, 2)`
if: false
res0: Int = 1
```

We'll discuss a more useful implementation of `__ifThenElse` in section A.5.3.

Besides `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)`
`a == (b_1,..., b_n)`	`__equal(a, b_1, ..., b_n)`

These methods are defined as follows (in EmbeddedControls, a trait that is mixed into Predef to provide the virtualization hooks at the top-level in any Scala program, although it can also be mixed into DSL traits to tweak the virtualization further):

[7] `https://github.com/adriaanm/scala-virtualized-tutorial`

```
  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
```

Since `Predef` inherits `EmbeddedControls`, these methods are visible everywhere. You can either shadow them by defining a synonymous method, or override them by inheriting `EmbeddedControls`.

A.2 Deep Embedding: Expressions

The CoreExps trait provides the core functionality for representing expressions. It defines the super trait `Exp[T]` for expressions that evaluate to a value of type `T`, and it provides a way of automatically lifting (staging-time) constants into their (trivial) representation, `Const[T]`.

```
trait CoreExps {
  trait Exp[T]

  case class Const[T](x: T) extends Exp[T]
  implicit def liftString(x: String): Exp[String]
    = Const(x)
}
```

It's really as simple as that: when a DSL program contains the string `"hello"`, all we need to know in order to represent it, is its literal value, `"hello"`, which is captured quite readily by the expression tree `Const("hello")`.

We'll now extend `CoreExps` with more interesting types.

A.3 Embedding SQL

Using the embedding of basic expressions introduce in the previous section, we now show how to generate SQL from a small query DSL embedded in Scala. Its representation is defined in the trait `SQLExps`, which extends the `CoreExps` trait.

```
trait SQLExps extends CoreExps {
  case class ResultRow[T](fields: Map[String, Exp[_]])
               extends Exp[T]

  case class Select[T, U](tgt: Exp[U], field: String)
               extends Exp[T]

  case class Table[Tuple <: Record](name: String)
               extends Exp[List[Tuple]]

  case class ListSelect[Tuple <: Record, T]
                       (t: Exp[List[Tuple]],
                        f: Exp[Tuple] => Exp[T])
               extends Exp[List[Tuple]] {
    def projectedNames: Iterable[String] = f(null) match {
      case ResultRow(fields) => fields.keys
    }
  }

  ...
}
```

An expression object of type `ResultRow` 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 tuple, e.g., `item.customerName`. To represent database tables we use the `Table` case class. It takes a type parameter `Tuple` that abstracts over the type of tuples stored in the table. `Tuple` extends the `Record` class, introduced below. Essentially, `Record` is used to create result tuples in queries. The `Table` class extends `Exp[List[Tuple]]`, i.e., it represents an expression returning a list of `Tuple` elements. This allows us to treat a table literal (which only contains the name of the corresponding database table) as a list of tuples, for which the standard `Select` clauses are defined.

To represent `Select` clauses we use the `ListSelect` case class. Like `Table`, it extends `Exp[List[Tuple]]` where `Tuple` is a type parameter for the type of the returned tuples. A `ListSelect` expression node points to a list expression of type `Exp[List[Tuple]]`, which is the list that we are selecting elements from, and an expression of type `Exp[Tuple] => Exp[T]`, which is the selector function used for determining (a) how to select elements from the list, and (b) how to transform a selected element to a result tuple. 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[Tuple]]`, and the function literal is lifted to an expression tree of type `Exp[Tuple] => Exp[T]`. This means the selector function can use all of the fields defined in the `Tuple` type to select columns to be included in the result.

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's now define the methods that are concerned with the actual embedding of query expressions in Scala. For this, we'll create a sub trait of `SQLExps` called `EmbedSQL`:

```
trait EmbedSQL extends SQLExps {
  ...
}
```

The first method we'll add provides `Select` clauses on lists of tuples:

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

It relies on Scala's standard implicit conversion mechanism: since an expression of type `Exp[List[Tuple]]` does not provide a method called `Select`, the compiler will turn an expression such as `table Select {f => ... }`, where `table : Exp[List[Tuple]]`, 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[Tuple]]`.

As we have seen in the previous example, result tuples 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`. We use the `Row` type constructor, which is part of virtualized Scala's library, as a marker to indicate when an instance creation `new T` should be virtualized. This reification will take place whenever `T <: Row[R]`, for some type constructor `R`. For convenience, we declare the following class inside our `SQLExps` trait:

```
class Record extends Row[Exp]
```

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

```
  def __new[T](args: (String, Exp[T] => Exp[_])*)
    : Exp[T] = new ResultRow(args map {
```

```
      case (n, rhs) => (n, rhs(null))}
    toMap)
```

The `__new` method takes a variable number of pairs as arguments. Each pair contains the name of a field in our row type, as well as 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 `ResultRow`, using the arguments to fill its map.

To support projections in queries, we need to be able to select fields of tuples. Therefore, we have to have a way to create expression trees for field selections. In Virtualized Scala 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, (which we add to our `EmbedSQL` trait):

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[Tuple]` does not define a field `customerName`, the Virtualized Scala compiler will generate an invocation of the `selectDynamic` method on `e`. Since it knows from the row type that the selection is supposed to produce an expression that evalues to a String, it will pass this information along to `selectDynamic` as a type argument.

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

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.

Let's create an example query that only uses the DSL elements introduced so far:

```
object Example extends EmbedSQL with SQLCodeGen {
  type Tuple = Record {
    val itemName: String
    val customerName: String
  }

  def prog = {
    val items = Table[Tuple]("items")
    items Select (e => new Record {
          val customerName = e.customerName })
  }
}
```

Compiling the `Example` object and running its `prog` method should create an expression tree corresponding to the following SQL query:

```
SELECT customerName FROM items
```

A.3.1 SQL code generation

Let's turn the expression trees into SQL. This is straightforward using methods that traverse and print our expression trees (|PrintWriter| is imported from the `java.io` package):

```
trait SQLCodeGen extends SQLExps {
  def emitPlain(out: PrintWriter, s: String,
                more: Boolean = false) = {
    if (more) out.print(s) else out.println(s)
  }

  def emitExpr[T](out: PrintWriter, expr: Exp[T]): Unit
    = expr match {
        case Select(_, field) =>
          emitPlain(out, field, true)
      }

  def emitSelector[T, S](out: PrintWriter,
                         f: Exp[T] => Exp[S]): Unit
    = f(null) match {
        case ResultRow(fields) =>
          var first = true
          for ((name, value) <- fields) {
            if (first) { first = false }
            else emitPlain(out, ", ", true)
            emitExpr(out, value)
          }
      }

  def emitQuery[T](out: PrintWriter,
                   expr: Exp[T]): Unit
    = expr match {
      case Table(name) =>
        emitPlain(out, name, true)

      case ListSelect(table, selector) =>
        emitPlain(out, "SELECT ", true)
        emitSelector(out, selector)
        emitPlain(out, " FROM ", true)
        emitQuery(out, table)
        emitPlain(out, "")
    }
  ...
}
```

A.4 Row types in depth

As discussed in the previous section, the virtualizing Scala compiler turns **new** `C{val x_i: T_i = v_i}` into the method call `__new( ("x_i", (self_i: R) => v'_i) )`. This section discusses this translation in more detail. It is also used in the JavaScript embedding below.

(Note: the subscripted index `_i` denotes implicit repetition of the smallest syntax tree that encompasses the subtrees subscripted by the same index.)

There's no definition of `__new` in `EmbeddedControls`, as its signature would be too unwieldy. Virtualization is not performed unless there exists a type constructor `Rep`, so that `C` is a subtype of `Row[Rep]`, where the marker trait `Row` is defined in `EmbeddedControls`:

```
trait Row[+Rep[x]]
```

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`)

Finally, the call `__new(("x_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, Rep[T] => Rep[_])*): Rep[T]`.

A.4.1 Type-safe selection on rows

When `e` refers to a representation of a row, `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 rows. No type argument will be supplied when the field's type cannot be determined (i.e., it's not in the row's refinement).

A.5 Embedding JavaScript

To embed JavaScript, we must add a new kind of AST node to our program representation: statements.

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"
}
```

The following section will show how to set up the DSL so that we can embed this programs as follows:

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

A.5.1 Statements

Virtualized Scala does not provide support for reifying (or virtualizing) the sequencing of statements. Luckily, it doesn't have to, as we can rely 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's consider the following fragment of our running example.

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

The virtualizing Scala compiler rewrites this to:

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

(See the reference in the beginning of this appendix for the details.)

These methods are defined as follows (see the full source for details):

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

What you don't see here is that `VarInit` and `VarAssign` are not expressions (`Exp[T]`); they are statements (`Def[T]`)! The infrastructure for dealing with statements provides the crucial missing ingredient, `toAtom`: an implicit conversion from `Exp[T]` to `Def[T]`. Let's first make the conversion explicit:

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

The crucial insight is that the order in which these `toAtom`'s 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 = toAtom(VarInit(...))
toAtom(VarAssign(selectOps(kim).selectDynamic("age"),
                 liftInt(21)))
```

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 `toAtom` populates this scope in the order in which it is called. On each invocation, `toAtom` 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.

A.5.2 Statements in depth

Statements in the domain program (values of type `Def[T]`) are embedded as Scala statements that have a single side-effect (from the host language's point of view): registering a new definition in the current scope (the scope that represents a part of the domain program).

This side-effect is implemented by `toAtom`, which largely remains hidden from the user (both of the DSL framework and the DSL itself) by virtue of being an implicit conversion. To achieve this, it is driven by type information: a statement is of type `Def[T]`, but the virtualizing methods are set up to return `Exp[T]`'s. This potential in type-difference drives Scala's implicit conversion mechanism to insert the `toAtom` conversion whenever a statement must be turned into an expression, the real representation of a DSL program.

The DSL scope book keeping 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.

The method `reifyBlock` creates a block by collecting 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: call it when entering a block in the DSL program.

The method `toAtom` appends a new definition to the current scope. This reifies the sequencing of definitions.

```
trait CoreDefs extends CoreExps {
  abstract class Def[T]

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

  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
    // toAtom 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 toAtom[T](d: Def[T]): Exp[T] = {
    // make a fresh symbol (: Exp[T]) to refer to the def
    val sym = Sym[T]()
    // append it to the current scope
    scopeDefs = (scopeDefs.head :+ ScopeEntry(sym, d)) ::
                scopeDefs.tail
    sym // the expression-representation of the definition
  }
}
```

A.5.3 If-then-else

Finally, with all this machinery in place, we can explain the virtualization of an if-then-else statement.

Its 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. In a sense this is "type-directed aspect-oriented programming": the concern of reifying sequence is factored out into implicit conversions (the "advice"), which are triggered by type annotations (the "join points").

Making the implicit conversions explicit, we see what's really going on:

```
        = toAtom(IfThenElse(cond,
                            reifyBlock(thenp),
                            reifyBlock(elsep)))
```

A.5.4 JavaScript Code Generation

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"
}
```

The full source[8] completes the example by generating a html page so you can easily run the generated code in your browser.

A.6 Compiling and Running Embedded Code (Staging)

As a final example we show why you might want to embed Scala in Scala. Generic matrix multiplication in Scala is very slow because primitive values are boxed and for comprehensions are translated into method calls, which entails allocating and garbage collecting a significant amount of closures. Additionally, Range objects are allocated simply to denote the start and end of the iteration's index.

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[T]` internally, we can reify the matrix operations and optimize them later.

```
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 `a.until(b)` where `a` and `b` have type `Exp[Int]` using virtualized Scala's `infix_*` external method introduction mechanism, which achieves the same effect as implicitly converting `Exp[Int]` to an anonymous class that has an `until` method (as we have been doing until now), but without the associated overhead.

Our more efficient `RangeExp` provides a `foreach` method that reifies the block that is passed into it.

```
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] = {
    val x = Sym[Int]()
    val y = reifyBlock { f(x) }
    Foreach(this, x, y)
  }
}
```

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.

```
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]` provides easy access to the implicit manifest that is in scope due to the `Manifest` context bound on `T`.

Finally, we can generate more efficient code for this representation.

First of all, we compile for loops down to while loops, as shown by this snippet from the code generator:

```
def emitNode[T](s: Sym[T], d: Def[T]): Unit = d match {
  case Foreach(r,x,y) =>
    emitPlain("var " + x + ": Int = " + r.start)
```

[8] https://github.com/adriaanm/scala-virtualized-tutorial/blob/master/src/org/scala_lang/virtualized/js/embedjs.scala

```
    emitPlain("while (" + x + " < " + r.end + ") ", true)
    emitBlock(y, false, x + " += 1")
    emitValDef(s, "()")
  ...
```

We also remove generic dispatch and instantiate all generic types. We get 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. We refer to the online sources for the details.

```
val f = compile {
  val m = randomMatrix(500, 100
  val n = randomMatrix(100, 500)

  val p = multGeneric(m,n)
  p.print
}
// call f() to execute block
```

Informal benchmarks indicate these simple optimizations result in a 20x speedup: the polymorphic multiplication takes 1.4s, when specializing to matrices of `double`s the run time is reduced to 1s, and the staged implementation reduces this time further by a factor of 20.

```
compilation: ok
3117192.0
--- generic took 2.691s
3117192.0
--- generic took 1.4s
3117192.0
--- generic took 1.464s
3117192.0
--- generic took 1.359s
3117192.0
--- generic took 1.244s

3117192.0
--- double took 1.062s
3117192.0
--- double took 1.228s
3117192.0
--- double took 1.076s
3117192.0
--- double took 1.03s
3117192.0
--- double took 1.076s

3117192.0
--- staged took 0.088s
3117192.0
--- staged took 0.058s
3117192.0
--- staged took 0.055s
3117192.0
--- staged took 0.054s
3117192.0
--- staged took 0.056s
```

This is just the tip of the iceberg of the optimizations enabled by our approach. Using Lightweight Modular Staging[9] and Delite[10] we can add parallelism, loop fusion, code motion and other advanced optimizations. We have a full program representation, so we can do anything a regular compiler can do, at a fraction of the implementation cost.

[9] http://github.com/TiarkRompf/virtualization-lms-core

[10] http://github.com/stanford-ppl/Delite