JavaScript as an Embedded DSL

Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky

Ecole Polytechnique Fédérale de Lausanne (EPFL)
{first.last}@epfl.ch

Abstract. Developing rich web applications requires mastering different environments on the client and server sides. While there is considerable choice on the server-side, the client-side is tied to JavaScript, which poses considerable software engineering challenges, such as moving or sharing pieces of code between the environments. We embed JavaScript as a DSL in Scala, using Lightweight Modular Staging. DSL code can be compiled to JavaScript or executed as part of the server application. We use features of the host language to make client-side programming safer and more convenient. We use gradual typing to interface typed DSL programs with existing JavaScript APIs. We exploit a selective CPS transform already available in the host language to provide a compelling abstraction over asynchronous callback-driven programming in our DSL.

Keywords: JavaScript, Scala, DSL, programming languages

1 Introduction

Developing rich web applications requires mastering a heterogeneous environment: though the server-side can be implemented in any language, on the client-side, the choice is limited to JavaScript. The trend towards alternative approaches to client-side programming (as embodied by CoffeeScript [8], Dart [13] & GWT [16]) shows the need for more options on the client-side. How do we bring advances in programming languages to client-side programming?

One challenge in developing a large code base in JavaScript is the lack of static typing, as types are helpful for maintenance, refactoring, and reasoning about correctness. Furthermore, there is a need for more abstraction and modularity. "Inversion of control" in asynchronous callback-driven programming leads to code with control structures that are difficult to reason about. A challenge is to introduce helpful abstractions without a big hit on performance and/or code size. Communication between the server side and the client side aggravates the impedance mismatch: in particular, data validation logic needs to be duplicated on the client-side for interactivity and on the server-side for security.

There are three widely known approaches for addressing the challenges outlined above. One is to create a standalone language or DSL that is compiled to JavaScript and provides different abstractions compared to JavaScript. Examples include WebDSL [34], Links [9,10] and Dart [13]. However, this approach usually requires a lot effort in terms of language and compiler design, and tooling

support, although WebDSL leverages Spoofax [18] to alleviate this effort. Furthermore, it is not always clear how these languages interact with other languages on the server-side or with the existing JavaScript ecosystem on the client-side.

Another approach is to start with an existing language like Java, Scala or Clojure and compile it to JavaScript. Examples include GWT [16], Scala+GWT [28] and Clojurescript [7]. This approach addresses the problem of impedance mismatch between client and server programming but comes with its own set of challenges. In particular, compiling Scala code to JavaScript requires compiling Scala's standard library to JavaScript as any non-trivial Scala program uses Scala collections. This leads to not taking full advantage of libraries and abstractions provided by the target platform which results in big code size and suboptimal performance of Scala applications compiled to JavaScript. For example, a map keyed by strings would be implemented natively in JavaScript as an object literal, while, in Scala, one would likely use the hash map from the standard library, causing it to be compiled to and emulated in JavaScript. Moreover, both approaches tend to not accommodate very well to different API design and programming styles seen in many existing JavaScript libraries.

A third approach is to design a language than is a thin layer on top of JavaScript but provides some new features. A prime example of this idea is CoffeeScript [8]. This approach makes it easy to integrate with existing JavaScript libraries but does not solve the impedance mismatch problem. In addition, it typically does not give rise to new abstractions addressing problems seen in callback-driven programming style, though some JavaScript libraries such as Flapjax [21] and Arrowlets [19] are specifically designed for this purpose.

We present a different approach, based on Lightweight Modular Staging (LMS) [26], that aims to incorporate good ideas from all the approaches presented above but at the same time tries to avoid their described shortcomings. LMS is a technique for embedding DSLs as libraries into a host language such as Scala, while enabling domain-specific compilation / code-generation. The program is split into two stages: the first stage is a program generator that, when run, produces the second stage program. Whether an expression belongs to the first or second stage is decided by its type. Expressions belonging to the second stage, also called "staged expressions", have type `Rep[T]` in the first stage when yielding a computation of type `T` in the second stage. Expressions evaluated in the first stage become constants at the second stage. Other approaches to staging include MetaML [32], LISP quasiquotations, and binding-time analysis in partial evaluation. Previous work has established LMS as a pragmatic approach to runtime code generation and compiled DSLs. In particular, the Delite framework [3,27,6] uses this approach to provide an extensible suite of high-performance DSLs targeting heterogeneous parallel platforms (with options to generate code to Scala, C and Cuda) [20], for domains such as machine learning [31], numeric array processing [33] and mesh-based partial differential equation solvers [5]. LMS has also been used to generate SQL queries [35].

We propose to embed JavaScript as a DSL in a host language. Through LMS, we tackle the challenges outlined above with minimal effort, as most of

the work is off-loaded to the host language. In particular, we make the following contributions:

- Our DSL is statically typed through the host language, yet supports gradual typing notably for incorporating external JavaScript libraries and APIs (section 3).
- In addition to generating JavaScript code, our DSL can be executed directly in the host language, allowing code to be shared between client and server (section 4).
- We use advanced object-oriented techniques to achieve modularity in our DSL: each language primitive and API is defined in a separate module (section 5).
- Our DSL supports typed object literals and class-based objects. The translations to JavaScript are lightweight and intuitive: the object literals translate to JSON-like object literals and the class-based objects to JavaScript constructor-based objects (section 6).
- With minimal effort, we exploit the selective CPS transform already existing in the host language to provide a compelling abstraction over asynchronous callback-driven programming in our DSL (section 7). This case-study demonstrates the fruitfulness of re-using host language features to enhance our embedded DSL.

In section 8, we describe our experience in using the DSL, and conclude in section 9.

2 Introduction to LMS

In LMS, a DSL is split into two parts, its interface and its implementation. Both parts can be assembled from components in the form of Scala traits. DSL programs are written in terms of the DSL interface only, without knowledge of the implementation.

Part of each DSL interface is an abstract type constructor `Rep[_]` that is used to wrap types in the DSL programs. The DSL implementation provides a concrete instantiation of `Rep` as IR nodes. When the DSL program is staged, it produces an intermediate representation (IR), from which the final code can be generated. In the DSL program, wrapped types such as `Rep[Int]` represent staged computations while expressions of plain unwrapped types (`Int`, `Bool`, etc.) are evaluated at staging time as in [4,17].

Consider the difference between these two programs:

```
def prog1(b: Bool, x: Rep[Int]) = if (b) x else x+1
def prog2(b: Rep[Bool], x: Rep[Int]) = if (b) x else x+1
```

The only difference in these two programs is the type of the parameter `b`, illustrating that staging is purely type-driven with no syntactic overhead as the body of the programs are identical.

In `prog1`, `b` is a simple boolean, so it must be provided at staging time, and the `if` is evaluated at staging time. For example, `prog1(true, x)` evaluates to `x`.

In `prog2`, `b` is a staged value, representing a computation which yields a boolean. So `prog2(b, x)` evaluates to an IR node for the **`if`**: `If(b, x, Plus(x, Const(1)))`.

For `prog2`, notice that the **`if`** got transformed into an IR node. To achieve this, LMS uses an extension of Scala, Scala-Virtualized [23], in which control structures such as **`if`** can be reified into method calls, so that alternative implementation can be provided. In our case, we provide an implementation of **`if`** that constructs an IR node instead of acting as a conditional. In addition, the `+` operation is overloaded to act on both staged and unstaged expressions. This is achieved by an implicit conversion from `Rep[Int]` to a class `IntOps`, which defines a `+` method that creates an IR node `Plus` when executed. Both of `Plus`'s arguments must be staged. We use an implicit conversion to stage constants when needed by creating a `Const` IR node.

2.1 Example: a DSL program and its generated JavaScript code

The following DSL snippet creates an array representing a table of multiplications:

```
def test(n: Rep[Int]): Rep[Array[Int]] =
  for (i <- range(0, n); j <- range(0, n)) yield i*j
```

Here is the JavaScript code generated for this snippet:

```
function test(x0) {
  var x6 = []
  for(var x1=0;x1<x0;x1++){
    var x4 = []
    for(var x2=0;x2<x0;x2++){
      var x3 = x1 * x2
      x4[x2]=x3
    }
    x6.splice.apply(x6, [x6.length,0].concat(x4))
  }
  return x6
}
```

2.2 Walkthrough: defining a DSL component

To conclude the introduction to LMS, we show how to add a component for logging in a DSL, generating JavaScript code which calls `console.log`.

We start by defining the interface:

```
trait Debug extends Base {
  def log(msg: Rep[String]): Rep[Unit]
}
```

The `Base` trait is part of the core LMS framework and provides the abstract type constructor `Rep`.

Now, we define the implementation:

```
trait DebugExp extends Debug with EffectExp {
  case class Log(msg: Exp[String]) extends Def[Unit]
  def log(msg: Exp[String]): Exp[Unit] = reflectEffect(Log(s))
}
```

The `EffectExp` trait is part of the core LMS framework. It inherits from `BaseExp` which instantiates `Rep` as `Exp`. `Exp` represents an IR via two subclasses: `Const` for constants, `Sym` for named values defining a `Def`. `Def` is the base class for all IR nodes. In our `DebugExp` trait, we extend `Def` to support a new IR node: `Log`.

IR nodes are defined as `Def`s but they are never referenced explicitly as such. Instead each `Def` has a corresponding symbol (an instance of `Sym`). IR nodes refer to each other using their symbols. This is why, in the code shown, the `msg` parameter is of type `Exp` (not `Def`). The method `log` returns an `Exp`. Calling `reflectEffect` is what creates this symbol from the `Def`.

In general, the framework provides an implicit conversion from `Def` to `Exp`, which performs common subexpression elimination by re-using the same symbol for identical definitions. We don't use it here, because `log` is a side-effecting operation, and we don't want to (re)move any such calls even if their message is the same.

The framework schedules the code generation from the graph of `Exp`s and their dependencies through `Def`s. It chooses which `Sym`/`Def` pairs to emit and in which order. To implement code generation to JavaScript for our logging IR node, we simply override `emitNode` to handle `Log`:

```
trait JSGenDebug extends JSGenEffect {
  val IR: DebugExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case Log(s) => emitValDef(sym, "console.log(" + quote(s) + ")")
      case _ => super.emitNode(sym, rhs)
    }
}
```

Notice that in order to compose nicely with other traits, the overridden method just handles the case it knows and delegates to other traits, via **super**, the emitting of nodes it doesn't know about.

3 Gradual Typing for Interfacing with Existing APIs

Since our DSL is embedded in Scala, it inherits its static type system. However, the generated JavaScript code doesn't need the static types. Therefore, to help integrate external JavaScript libraries and APIs (for example, the browser's DOM API), we support a form of gradual typing. This has proved especially useful for rapid-prototyping, where external libraries are first incorporated dynamically, and later declared as typed APIs. Various practical and theoretical aspects of gradual typing have been studied by [36,2,1,29,30].

3.1 Typed APIs

First, we show how to incorporate an external JavaScript API in a fully-typed way into our DSL. As an example, consider the following DSL snippet, which gets the context of an HTML5 canvas element selected by id:

```
val context = document.getElementById("canvas").as[Canvas].geContext()
```

At the DSL interface level, we declare our typed APIs as abstract Scala traits:

```
trait Dom {
  val document: Rep[Element]
  trait Element
  trait ElementOps {
    def getElementById(id: Rep[String]): Rep[Element]
  }
  trait Canvas extends Element
  trait CanvasOps extends ElementOps {
    def getContext(context: Rep[String]): Rep[Context]
  }
  trait Context
  trait ContextOps {
    def lineTo(x: Rep[Int], y: Rep[Int]): Rep[Unit]
    // etc.
  }
}
```

Notice that `document` has type `Rep[Element]`, and needs to implement the interface of `ElementOps`, so that `document.getElementById("canvas")` is well-typed. We achieve this using an implicit conversion from `Rep[Element]` to `ElementOps`. At the DSL implementation level, the `ElementOps` returned by this implicit conversion needs to generate an IR node for each method call, as shown in the walkthrough in 2.2. For example, `document.getElementById("canvas")` becomes the IR node `MethodCall(document, "getElementById", List("canvas"))`. This is a mechanical transformation, implemented by `repProxy` using reflection to intercept method calls and generate IR nodes based on the method name and the arguments of the invocation. Note that this use of reflection is purely at staging time, so there is no overhead in the generated code.

```
trait DomLift extends Dom with JSProxyBase {
  implicit def repToElementOps(x: Rep[Element]): ElementOps =
    repProxy[Element,ElementOps](x)
  implicit def repToCanvasOps(x: Rep[Canvas]): CanvasOps =
    repProxy[Canvas,CanvasOps](x)
  implicit def repToContextOps(x: Rep[Context]): ContextOps =
    repProxy[Context,ContextOps](x)
}
```

Note also that since `getElementById` returns an arbitrary DOM element, we need to cast it to a `Canvas` using `as[Canvas]`. `as` is implemented simply as a cast in the host language (no IR node is created):

```
  trait AsRep {
    def as[T]: Rep[T]
```

```
  }
  implicit def asRep(x: Rep[_]): AsRep = new AsRep {
    def as[T]: Rep[T] = x.asInstanceOf[Rep[T]]
  }
```

Instead of this no-op implementation, it is possible to insert run-time check-cast assertions in the generated JavaScript code.

3.2 Casting and Optional Runtime Type Checks

The need for casting arises in a few contexts. One of them is the boundary between typed and untyped portions of a program [36,2]. Passing a value from an untyped portion to a typed one usually requires a cast. Another situation where casts are needed is interaction with external services. For example, to process data from an external service such as Twitter, we cast it to its expected type:

```
type TwitterResponse = Array[JSLiteral {val text: String}]
def fetchTweets(username: Rep[String]) = {
 val raw = ajax.get { ... }
 raw.as[TwitterResponse]
}
```

A more complete example is provided in section 7. In this situation, it is useful to generate runtime checks either as an aid during development and debugging time or as a security mechanism that validates data coming from an external source. In the example above, if runtime checks are enabled, by failing early, we obtain a guarantee that all data returned from `fetchTweets` conforms to type `TwitterResponse` which means that any later access to the `text` field of any element of the data array will never fail, and always return a string.

Notice that when a typed API is defined for an external library, there are implicit casts introduced for argument and return types of the defined methods. These casts can also be checked at runtime to ensure compliance.

We've implemented a component that generates JavaScript code that asserts casts at runtime. It was fairly straightforward as the host language allows us to easily inspect types involved in casting. Since this component just provides a different implementation of the same casting method `as`, it can be enabled selectively for performance reasons.

3.3 Scala Dynamic

Since our DSL compiles to JavaScript, which is dynamically typed, it is appealing to allow expressions and APIs in our DSL to also, selectively, be dynamically typed. This is especially useful for rapid-prototyping.

We provide a component, `JSDynamic`, which allows any expression to become dynamically typed, by wrapping it in a `dynamic` call. The `dynamic` wrapper returns a `DynamicRep`, on which any method call, field access and field update is possible. A dynamic method call and field access returns another `DynamicRep` expression, so dynamic expressions can be chained.

`DynamicRep` exploits a new Scala feature, based on a marker trait `Dynamic`: An expression whose type `T` is a subtype of `Dynamic` is subject to the following rewrites:

- `x.f` rewrites to `x.selectDynamic("f")`,
- `x.m(a, ..., z)` rewrites to `x.applyDynamic("m")(a, ..., z)`,
- `x.f = v` rewrites to `x.updateDynamic("f")(v)`.

These rewrites take place when the type `T` doesn't statically have field `f` and method `m`.

At the implementation level, as these rewriting take place, we generate IR nodes which allow us to then generate straightforward JavaScript for the original expressions. For example, for the expression `dynamic(x).foo(1, 2)`, we would generate an IR node like `MethodCall(x, "foo", List(Const(1), Const(2)))`. From this IR node, it is easy to generate the JavaScript code `x.foo(1, 2)`. Note the similarity with the IR nodes generated for typed APIs.

3.4 From Dynamic to Static

This possibility to escape into dynamic typing is particularly useful in simplifying the incorporation of external JavaScript APIs and libraries. Sometimes, the user might not want to build a statically typed API for each external JavaScript library. In addition, for some library, it might awkward to come up with such a statically-typed interface. In general, we expect users to start with a dynamic API for an external library, and progressively migrate it to a typed API as the code matures. Consider again the example introduced in the typed API section:

```
val context = document.getElementById("canvas").as[Canvas].getContext()
```

In a fully dynamic scenario, we declare the DOM API simply as:

```
trait Dom extends JSDynamic {
  val document: DynamicRep
}
```

The `as[Canvas]` cast is not necessary in this dynamically typed setting:

```
val context = document.getElementById("canvas").geContext()
```

As a first step towards statically typing the API, we declare the type `Element`:

```
trait Dom extends JSProxyBase with JSDynamic {
  val document: Rep[Element]
  trait Element
  trait ElementOps {
    def getElementById(id: Rep[String]): DynamicRep
  }
  implicit def repToElementOps(x: Rep[Element]): ElementOps =
    repProxy[Element,ElementOps](x)
}
```

Since the method `getElementById` returns a `DynamicRep`, only the emphasized part of the expression is statically typed:

```
val context = document.getElementById("canvas").geContext()
```

We can then complete the static typing by declaring types for `Canvas` and `Context` as seen in the typed API section.

In our gradual typing scheme, an expression is either completely statically typed or completely dynamically typed. Once `document` is declared as `Rep[Element]` instead of `DynamicRep`, it is a type error to call an arbitrary method which is not part of its declared `ElementOps` interface. If needed, it is always possible to explicitly move from a statically-typed expression to a dynamically-typed one by wrapping it in a `dynamic` call.

4 Sharing Code between Client and Server

In addition to generating JavaScript / client-side code, we want to be able to re-use our DSL code on the Scala / server-side. In the LMS approach, the DSL uses the abstract type constructor `Rep` [22]. When generating JavaScript, this abstract type constructor is defined by IR nodes. Another definition, which we dub "trivial embedding", is to use the identity type constructor: `Rep[T] = T`. By stripping out `Rep`s in this way, our DSL can operate on concrete Scala types, replacing staging with direct evaluation. Even in the trivial embedding, when the DSL code operates on concrete Scala types, virtualization still occurs because the usage layer of the DSL is still in terms of abstract `Rep`s.

As an example, consider the following DSL snippet, which computes the absolute value:

```
trait Ex extends JS {
  def abs(x: Rep[Int]) = if (x < 0) -x else x
}
```

We can use this DSL snippet to generate JavaScript code:

```
new Ex with JSExp { self =>
  val codegen = new JSGen { ... }
  codegen.emitSource(abs _, "abs", ...)
}
```

We can also use this DSL snippet directly in Scala via the trivial embedding (defined by `JSInScala`):

```
new Ex with JSInScala { self =>
  println(abs(-3))
}
```

In the JavaScript example, when `abs` gets evaluated, through mixing in `JSExp`, it results in an IR tree roughly equivalent to `If(LessThan(Sym("x"), Const(0)), Neg(Sym("x")), Sym("x"))`. In the trivial embedding, when `abs(-3)` gets called, it evaluates to `3` by executing the virtualized **`if`** as a normal condition. In short, in the trivial embedding, the virtualized method calls are evaluated in-place without constructing IR nodes.

In the previous section, we showed how to define typed APIs to represent JavaScript external libraries or dependencies. In the trivial embedding, we need to give an interpretation to these APIs. For example, we can implement a Canvas

context in Scala by using native graphics to draw on a desktop widget instead of a web page. We translated David Flanagan's Canvas example from the book "JavaScript: The Definitive Guide" [15], which draws Koch snowflakes on a canvas [14]. First, the translation from JavaScript to our DSL is straightforward: the code looks the same except for some minor declarations. Then, from our DSL code, we can generate JavaScript code to draw the snowflakes on a canvas as in the original code. In addition, via the trivial embedding, we can execute the DSL code in Scala to draw snowflakes on a desktop widget. Screenshot presenting snowflakes rendered in a browser using HTML5 Canvas and Java's 2D are presented in figure 1.

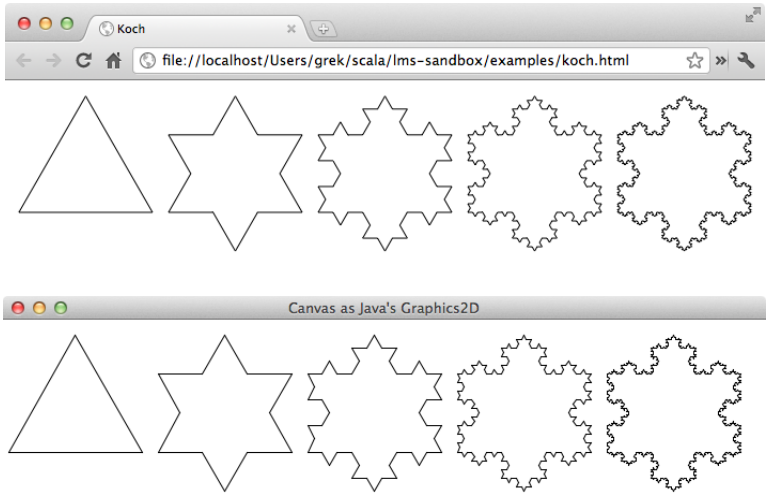


Fig. 1. Snowflakes rendered using HTML5 Canvas and Java's 2D

HTML5 Canvas is a standard that is not implemented by all browsers yet so a fall-back mechanism is needed to support users of older browsers. This can be achieved through the trivial embedding by drawing using Java's 2D API, saving the result as an image and sending it to the browser. The decision to either generate a JavaScript snippet that draws on canvas and send it to the browser, or render the image on the server can be made at runtime (e.g. after inspecting information about the client's browser). In the case of rendering on the server-side, one can store computation that renders an image using Java's graphics 2D in a hash map and send back to the client the key as an url for an image. When a browser makes a second request, computation can be restored from the hash map, executed and the result sent back to the browser. All of that is possible because the computation itself is expressed against an abstract DSL API so we can swap implementations to either generate JavaScript or run the computation at the server side. Moreover, our DSL is just a library and computations are

expressed as first-class values in a host language so they can be passed around the server program, stored in a hash map, etc.

A drawback of the trivial embedding is that the virtualization overhead is incurred each time the program is evaluated. To avoid this, we could generate Scala code in the same way as we now generate JavaScript code, relying on a mechanism to incorporate the generated Scala code into the rest of the program. In fact, this approach is taken by the Delite framework.

5 Modularity Interlude

The design of our DSL supports modularity at many levels. We use Scala's traits heavily to allow our DSL to be assembled from and extended with components [24].

For example, the feature to escape into dynamic typing is implemented as an independent component that can be mixed and matched with others. Similarly, users specify external APIs as components. The separation between the interface level and the implementation level is also done by having distinct components for each level. This allows the same DSL program to be interpreted in multiple ways, as has been shown with the trivial embedding to Scala.

As the features available in a DSL program are specified by composing components, it is possible to use this mechanism to enforce that a subprogram only uses a restricted set of features. For example, worker threads in JavaScript ("WebWorkers") are not allowed to manipulate the DOM. This can be enforced by not mixing in the DOM component in the subprogram for a worker thread.

The code generation level is assembled from components as in the interface and implementation levels. Furthermore, optimizations in code generation can be implemented as optional components to be mixed in.

6 Reification of Objects

By exploiting staging, the generated code can remain simple and relatively unstructured as many of the high-level constructs can be evaluated away at staging time. However, it is sometimes useful to be able to reify more complex structures. For example, APIs sometimes expect arguments or return results as object literals. Therefore, we support a few type-safe ways to create more complex staged structures, which we explain below.

6.1 Functions

A function in the host language acting on reified types has type: `Rep[A] => Rep[B]`. For example, the function `inc` has type `Rep[Int] => Rep[Int]`:

```
val inc = (x: Rep[Int]) => x + 1
```

Invoking such a function at staging time simply inlines the call: `inc(4*x)` results in `4*x + 1`. This is useful and nice, because it removes abstraction overhead from the generated code.

We also want the ability to treat functions as first-class staged values, since JavaScript supports them. In order to do this, we provide a higher-order function `fun` which takes a function of type `Rep[A] => Rep[B]` and converts[1] it to a staged function of type `Rep[A => B]`. For example, if we define `inc` in the following way, its type is `Rep[Int => Int]`:

```
val inc = fun { (x: Rep[Int]) => x + 1 }
```

Calling `inc(4*x)` results in an `Apply` IR node. We actually generate JavaScript code for the staged `inc` function, while we didn't for the unstaged one, since it's inlined at every call site during staging. The generated code looks roughly like the following:

```
var inc = function(a) {
  return a+1
}
inc(4*x)
```

First-class functions are widely used in JavaScript. One particular common case is for callback-driven programming. Therefore, staged functions are important to interface with existingn libraries. They will also play a crucial role in section 7, where we abstract over callback-driven programming.

6.2 Typed Object Literals

Our DSL provides typed immutable object literals to represent JavaScript object literals. As an example:

```
val o = new JSLiteral {
  val a = 1
  val b = a + 1
}
```

`o` has type `Rep[JSLiteral {val a: Int; val b: Int}]`. All the fields of `o` are `Rep`s, so `o.a` has type `Rep[Int]`. The translation to JavaScript is straightforward:

```
var o = { a : 1, b : 2 }
```

This straightforward translation makes it possible to pass the typed object literals of our DSL to JavaScript functions which expects object literals, such as `JSON.stringify` or jQuery's `css`.

As for implementation, notice that the type of a `new JSLiteral {...}` expression is not `JSLiteral {...}` but `Rep[JSLiteral {...}]`. This is achieved with support from the Scala-Virtualized compiler. JSLiteral inherits from a special marker trait, which indicates its `new` expressions should be reified. So the `new` expression is turned into a method call, with information about all the field definitions. A complication is that a field definition might reference another field being defined (such as `b` being defined in terms of `a` in the example above). So

[1] We refer the reader to [26] for implementation details

each definition is represented by its name and a *function* which takes a self type for the object literal. These definition functions are evaluated in an order which allows the self references to be resolved.

6.3 Classes

Our DSL also supports reified classes. For convenience, these are defined as traits, use the `repProxy` mechanism underlying typed APIs and are also implemented using reflection. The translation to JavaScript, based on constructors and prototypes, is straightforward. Through the trivial embedding, classes implemented in our DSL can be used on both the server and client sides.

7 CPS Transformation for Asynchronous Code Patterns

A callback-driven programming style is pervasive in JavaScript programs. Because of lack of thread support, callbacks are used for I/O, scheduling and event-handling. For example, in an Ajax call, one has to provide a callback that will be called once the requested data arrives from the server. This style of programming is known to be unwieldy in more complicated scenarios. To give a specific example, let's consider a scenario where we have an array of Twitter account names and we want to ask Twitter for the latest tweets of each account. Once we obtain the tweets of all users, we would like to log that fact in a console.

We'll implement this program both in JavaScript and in our DSL. Let's start by implementing logic that fetches tweets for a single user by using the jQuery library for Ajax calls (listings 1.1 & 1.2).

Listing 1.1. Fetching tweets in JavaScript

```
function fetchTweets(username, callback) {
  jQuery.ajax({
    url:
"http://api.twitter.com/1/statuses/user_timeline.json/",
    type: "GET",
    dataType: "jsonp",
    data: {
      screen_name : username,
      include_rts : true,
      count : 5,
      include_entities : true
    },
    success: callback
  })
}
```

Listing 1.2. Fetching tweets in DSL

```
def fetchTweets(username: Rep[String]) = (ajax.get {
  new JSLiteral {
    val url =
"http://api.twitter.com/1/statuses/user_timeline.json"
    val `type` = "GET"
    val dataType = "jsonp"
    val data = new JSLiteral {
      val screen_name = user
      val include_rts = true
      val count = 5
      val include_entities = true
    }
  }
}).as[TwitterResponse]

type TwitterResponse = Array[JSLiteral {val text: String}]
```

Note that JavaScript version takes a callback as second argument that will be used to process the fetched tweets. We provide the rest of the logic that iterates over array of users and makes Ajax requests (listings 1.3 & 1.4).

Listing 1.3. Twitter example in JavaScript

```
var processed = 0
var users = ["gkossakowski", "odersky", "adriaanm"]
users.forEach(function (user) {
  console.log("fetching " + user)
  fetchTweets(user, function(data) {
    console.log("finished fetching " + user)
    data.forEach(function (tweet) {
      console.log("fetched " + tweet.text)
    })
    processed += 1
    if (processed == users.length) {
      console.log("done")
    }
  })
})
```

Listing 1.4. Twitter example in DSL

```
val users = array("gkossakowski", "odersky", "adriaanm")
for (user <- users.parSuspendable) {
  console.log("fetching " + user)
  val tweets = fetchTweets(user)
  console.log("finished fetching " + user)
  for (t <- tweets)
    console.log("fetched " + t.text)
}
console.log("done")
```

Because of the inverted control flow of callbacks, synchronization between callbacks has to be handled manually. Also, the inverted control flow leads to a code structure that is distant from the programmer's intent. Notice that the in JavaScript version, the call to console that prints "done" is put inside of the foreach loop. If it was put it after the loop, we would get "done" printed *before* any Ajax call has been made leading to counterintuitive behaviour.

As an alternative to the callback-driven programming style, one can use an explicit monadic style, possibly sugared by a Haskell-like "do"-notation. However, this requires rewriting possibly large parts of a program into monadic style when a single async operation is added. Another possibility is to automatically transform the program into continuation passing style (CPS), enabling the programmer to express the algorithm in a straightforward, sequential way and creating all the necessary callbacks and book-keeping code automatically. Links [9] uses this approach. However, a whole-program CPS transformation can cause performance degradation, code size blow-up, and stack overflows. In addition, it is not clear how to interact with existing non-CPS libraries as the whole program needs to adhere to the CPS style. We suggest using a *selective* CPS transformation, which precisely identifies what needs to be CPS transformed.

In fact, the Scala compiler already does selective, `@suspendable` type-annotation-driven CPS transformations of Scala programs [25,11,12]. We show how this mechanism can be used for transforming our DSL code before staging and stripping out most CPS abstractions at staging time. The generated JavaScript code doesn't have any CPS-specific code but is written in CPS-style by use of JavaScript anonymous functions.

7.1 CPS in Scala

Before presenting how CPS transformations are used in our DSL, let's consider a typical situation where CPS rewrites act on Scala programs.

As an example, we'll consider a `sleep` method implemented in non-blocking, asynchronous style. This is useful, for example, when using `ThreadPool`s as no

thread is being blocked during the sleep period. Let's see how our `sleep` method written in CPS can be used:

```
def foo() = {
  sleep(1000)
  println("Called foo")
}
reset {
  println("look, Ma ...")
  foo()
  sleep(1000)
  println(" no threads!")
}
```

The `reset` delimits the scope of CPS rewrite. Let's see how the rewrite itself looks like:

```
def foo(): ControlContext = {
  sleep(1000).map((tmp1: Unit) => println("Called foo"))
}
reset {
  println("look, Ma ...")
  foo().flatMap((tmp2: Unit) =>
    sleep(1000).map((tmp3: Unit) => println(" no threads!"))
  )
}
```

There are a few things worth noting here. First, the return type of `foo` method is rewritten to be `ControlContext`[2]. This is due to the fact that `sleep` is used in the body of `foo`. Also, note that the code to be executed after sleeping is captured as a continuation (anynomous function) and passed to the `ControlContext` through a call of the `map` method. The body of the `reset` block is rewritten in a similar vein.

Now, let's have a closer look how `sleep` itself is implemented:

```
import java.util.{Timer,TimerTask}
val timer = new Timer
def sleep(delay: Int): Unit @suspendable = shift { k =>
  val task = new TimerTask { def run() = k() }
  timer.schedule(task, delay.toLong)
}
```

Notice the `@suspendable` type annotation attached to `sleep`'s return type `Unit`. The `@suspendable` annotation means that the `sleep` method can be used in a side-effecting continuation context. In the definition of the `sleep` method, we use Java's `Timer` and `TimerTask` abstractions for asynchronous, delayed task execution. The `TimerTask` interface has one method, `run`, that will be executed

[2] The `ControlContext` class implements the continuation monad and is provided by Scala's standard library. `ControlContext` is similar to C#'s `Task<T>`, but more general.

after a specified period of time. The `shift` control abstraction allows us to capture the continuation as a first-class value and then use it in the body of the `run` method. Both the `reset` and `shift` control abstractions are described in detail in [11].

After the CPS transformation, the code presented above becomes

```
def sleep(time: Int): ControlContext =
  shiftR { k =>
    val task = new TimerTask { def run() = k() }
    timer.schedule(task, delay.toLong)
  }
```

After the rewriting, all CPS-related type annotations are dropped and use of the `ControlContext` class that supports continuation passing is introduced. The `shiftR` method is an internal method that takes a block (which itself is a function) and wraps it in `ControlContext` structure.

7.2 CPS and Staging

Let's write the example from listing 7.1 in our DSL. We need to define `sleep` to use JavaScript's `setTimeout` as a replacement for the `Timer` abstraction.

```
def sleep(delay: Rep[Int]) = shift { k: (Rep[Unit]=>Rep[Unit]) =>
  window.setTimeout(fun(k), delay)
}
```

The `setTimeout` method expects an argument of type `Rep[Unit=>Unit]` which denotes a *representation* of a function of type `Unit=>Unit`. The `shift` method offers us a function of type `Rep[Unit] => Rep[Unit]`, so we need to reify it to obtain the desired representation. The reification is achieved by the `fun` function (described in 6.1) provided by our framework and performed at staging time.

It's important to note that reification preserves function composition. Specifically, let `f: Rep[A] => Rep[B]` and `g: Rep[B] => Rep[C]` then `fun(g compose f) == (fun(g) compose fun(f))` where we consider two reified functions to be equal if they yield the same observable effects at runtime. That property of function reification is at the core of reusing the continuation monad in our DSL. Thanks to the fact that the continuation monad composes functions, we can just insert reification at some places (like in a `sleep`) and make sure that we reify *effects* of the continuation monad without the need to reify the monad itself.

7.3 CPS for Interruptible Traversals

We need to be able to interrupt our execution while traversing an array in order to implement functionality from listing 1.4. Let's consider a simplified example where we want to iterate over an array and sleep during each iteration. We present both code written in JavaScript and our DSL that achieves that (listings 1.5 & 1.6).

Both programs, when executed, will print following to the JavaScript console:

Listing 1.5. JavaScript

```
var xs = [1, 2, 3]
var i = 0
var msg = null
function f1() {
  if (i < xs.length) {
    window.setTimeout(f2, xs[i]*1000)
    msg = xs[i]
    i++
  }
}
function f2() {
  console.log(msg)
  f1()
}
f1()
```

Listing 1.6. DSL

```
val xs = array(1, 2, 3)
// shorthand for xs.suspendable.foreach
for (x <- xs.suspendable) {
  sleep(x * 1000)
  console.log(String.valueOf(x))
}
log("done")
```

```
//pause for 1s
1
//pause for 2s
2
//pause for 3s
3
done
```

In the DSL code, we use a `suspendable` variant of arrays, which is achieved through an implicit conversion from regular arrays:

```
implicit def richArray(xs: Rep[Array[A]]) = new {
  def suspendable: SArrayOps[A] = new SArrayOps[A](xs)
}
```

The idea behind `suspendable` arrays is that iteration over them can be interrupted. We'll have a closer look at how to achieve that with the help of CPS. The `suspendable` method returns a new instance of the `SArrayOps` class defined here:

Listing 1.7. Suspendable foreach

```
class SArrayOps(xs: Rep[Array[A]]) {
  def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
    Rep[Unit] @suspendable = {
      var i = 0
      suspendableWhile(i < xs.length) { yld(xs(i)); i += 1 }
    }
}
```

Note that one cannot use while loops in CPS but we can simulate them with recursive functions. Let's see how regular while loop can be simulated with a recursive function:

```
def recursiveWhile(cond: => Boolean)(body: => Unit): Unit = {
  def rec = () => if (cond) { body; rec() } else {}
  rec()
```

```
}
```

By adding CPS-related declarations and control abstractions, we implement `suspendableWhile`:

```
def suspendableWhile(cond: => Rep[Boolean])(
  body: => Rep[Unit] @suspendable): Rep[Unit] @suspendable =
  shift { k =>
    def rec = fun { () =>
      if (cond) reset { body; rec() } else { k() }
    }
    rec()
  }
```

7.4 Defining the Ajax API

With the abstractions for interruptible loops and traversals at hand, what remains to complete the Twitter example from the beginning of the section is the actual Ajax request/response cycle.

The Ajax interface component provides a type `Request` that captures the request structure expected by the underlying JavaScript/jQuery implementation and the necessary object and method definitions to enable the use of `ajax.get` in user code:

```
trait Ajax extends JS with CPS {
  type Request = JSLiteral {
    val url: String
    val `type`: String
    val dataType: String
    val data: JSLiteral
  }
  type Response = Any
  object ajax {
    def get(request: Rep[Request]) = ajax_get(request)
  }
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable
}
```

Notice that the `Request` type is flexible enough to support an arbitrary object literal type for the `data` field through subtyping. The `Response` type alias points at `Any` which means that it is the user's responsibility to either use `dynamic` or perform an explicit cast to the expected data type.

The corresponding implementation component implements `ajax_get` to capture a continuation, reify it as a staged function using `fun` and store it in an `AjaxGet` IR node.

```
trait AjaxExp extends JSExp with Ajax {
  case class AjaxGet(request: Rep[Request],
    success: Rep[Response => Unit]) extends Def[Unit]
```

```
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable =
    shift { k =>
      reflectEffect(AjaxGet(request, fun(k)))
    }
}
```

During code generation, we emit code to attach the captured continuation as a callback function in the `success` field of the request object:

```
trait GenAjax extends JSGenBase {
  val IR: AjaxExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case AjaxGet(req, succ) =>
        stream.println(quote(req) + ".success = " + quote(succ))
        emitValDef(sym, "jQuery.ajax(" + quote(req) + ")")
    case _ => super.emitNode(sym, rhs)
  }
}
```

It's interesting to note that, since the request already has the right structure for the `jQuery.ajax` function, we can simply pass it to the framework-provided `quote` method, which knows how to generate JavaScript representations of any `JSLiteral`.

The Ajax component completes the functionality required to run the Twitter example with one caveat: The outer loop in listing 1.4 uses `parSuspendable` to traverse arrays instead of the `suspendable` traversal variant we have defined in listing 1.7.

In fact, if we change the code to use `suspendable` instead of `parSuspendable` and run the generated JavaScript program, we'll get following output printed to the JavaScript console:

```
fetching gkossakowski
finished fetching gkossakowski
fetched [...]
fetched [...]
fetching odersky
finished fetching odersky
fetched [...]
fetched [...]
fetching adriaanm
finished fetching adriaanm
fetched [...]
fetched [...]
done
```

Notice that all Ajax requests were done sequentially. Specifically, there was just one Ajax request active at a given time; when the callback to process one request is called, it would resume the continuation to start another request, and

so on. In many cases this is exactly the desired behavior, however, we will most likely want to perform our Ajax request in parallel.

7.5 CPS for Parallelism

The goal of this section is to implement a parallel variant of the `foreach` method from listing 1.7. We'll start with defining a few primitives like futures and dataflow cells. Let's start with cells, which we decide to define in JavaScript, as another example of integrating external libraries with our DSL:

Listing 1.8. JavaScript-based implementation of a non-blocking Cell

```
function Cell() {
  this.value = undefined
  this.isDefined = false
  this.queue = []
  this.get = function (k) {
    if (this.isDefined) {
      k(this.value)
    } else {
      this.queue.push(k)
    }
  }
  this.set = function (v) {
    if (this.isDefined) {
      throw "can't set value twice"
    } else {
      this.value = v
      this.isDefined = true
      this.queue.forEach(function (f) {
        f(v) //non-trivial spawn could be used here
      })
    }
  }
}
```

A cell object allows us to track dependencies between values. Whenever the `get` method is called and the value is not in the cell yet, the continuation will be added to a queue so it can be suspended until the value arrives. The `set` method takes care of resuming queued continuations. We expose `Cell` as external library using our typed API mechanism and we use it for implementing an abstraction for futures.

```
def createCell(): Rep[Cell[A]]
trait Cell[A]
trait CellOps[A] {
  def get(k: Rep[A => Unit]): Rep[Unit]
  def set(v: Rep[A]): Rep[Unit]
}
```

```
implicit def repToCellOps(x: Rep[Cell[A]]): CellOps[A] =
  repProxy[Cell[A],CellOps[A]](x)

def spawn(body: => Rep[Unit] @suspendable): Rep[Unit] = {
  reset(body) //non-trivial implementation uses
              //trampolining to prevent stack overflows
}
def future(body: => Rep[A] @suspendable) = {
  val cell = createCell[A]()
  spawn { cell.set(body) }
  cell
}
```

The last bit of general functionality we need is `RichCellOps` that ties `Cell`s and continuations together inside of our DSL.

```
class RichCellOps(cell: Rep[Cell[A]]) {
  def apply() = shift { k: (Rep[A] => Rep[Unit]) =>
    cell.get(fun(k))
  }
}
implicit def richCellOps(x: Rep[Cell[A]]): RichCell[A] =
  new RichCellOps(x)
```

It's worth noting that `RichCellOps` is not reified so it will be dropped at staging time and its method will get inlined whenever used. Also, it contains CPS-specific code that allows us to capture the continuation. The `fun` function reifies the captured continuation.

We are ready to present the parallel version of `foreach` defined in listing 1.7.

```
def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
  Rep[Unit] @suspendable = {
    val futures = xs.map(x => future(yld(x)))
    futures.suspendable.foreach(_.apply())
  }
```

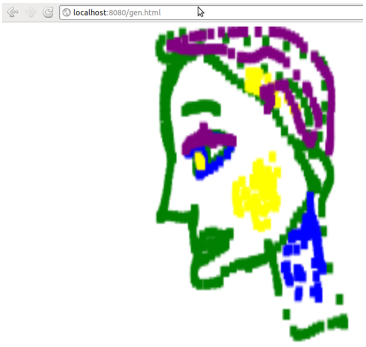
We instantiate each future separately so they can be executed in parallel. As a second step we make sure that all futures are evaluated before we leave the `foreach` method by forcing evaluation of each future and "waiting" for its completion. Thanks to CPS transformations, all of that will be implemented in a non-blocking style.

The only difference between the parallel and serial versions of the Twitter example 1.4 is the use of `parSuspendable` instead of `suspendable` so the parallel implementation of the `foreach` method is used. The rest of the code stays the same. It's easy to switch between both versions, and users are free to make their choice according to their needs and performance requirements.

8 Evaluation

We've implemented our DSL in Scala [3]. We used our DSL to develop a few web applications, which are simple but not trivial. First, we implemented a few drawing examples like the snowflakes of figure 1. We extended the Twitter example from section 7, which presents the latest tweets from selected users in an interactive way. In order to do so, we incorporated a useful subset of the DOM API and jQuery library using our typed APIs.

Finally, we developed a collaborative drawing application, which includes a server-side component (implemented using the Jetty web server). We use web sockets to communicate between the server and clients. Each client transmits drawing commands to the server, which broadcasts them to all the clients. When a new client joins, the server sends the complete drawing history to the new client, and the client reconstructs the image by playing the commands. A very simple improvement is to make the server execute the drawing commands as well, and keep an up-to-date bitmap of the drawing – this can easily be achieved by using the trivial embedding described in section 4. New clients then just obtain the bitmap instead of replaying the history, which is potentially be large.



9 Conclusion

In this paper, we have shown how to embed a JavaScript DSL in Scala using LMS. A recurring theme of our approach is to exploit the features of the host language to enhance the DSL with minimal effort. Moreover, through staging, we can use many abstractions at the code-generation stage, without complexity and performance overhead in the generated code. We believe this approach addresses some important challenges of developing rich web applications.

10 Acknowledgments

We thank Adriaan Moors for maintaining the Scala-Virtualized compiler, adding our feature requests, fixing our reported bugs, and for insightful discussions.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13:237–268, April 1991.

[3] The code can be found at `http://github.com/namin/lms-sandbox`.

2. A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, pages 1–13, New York, NY, USA, 2009. ACM.
3. K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.
4. J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19:509–543, September 2009.
5. 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.
6. H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.
7. `https://github.com/clojure/clojurescript/wiki`.
8. `http://jashkenas.github.com/coffee-script/`.
9. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *Proceedings of the 5th international conference on Formal methods for components and objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag.
10. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. The essence of form abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag.
11. O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM.
12. O. Danvy and A. Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
13. `http://http://www.dartlang.org/`.
14. D. Flanagan. `https://github.com/davidflanagan/javascript6_examples/blob/master/examples/21.06.koch.js`, 2011.
15. D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly Media, Inc., 6th edition, 2011.
16. `http://code.google.com/webtoolkit/`.
17. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.
18. L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.
19. Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing javascript with arrows. *SIGPLAN Not.*, 44:49–58, October 2009.
20. H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31:42–53, Sept. 2011.

21. L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.
22. A. Moors, F. Piessens, and M. Odersky. Generics of a Higher Kind. *Acm Sigplan Notices*, 43:423–438, 2008.
23. A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *PEPM12*, 2012.
24. M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of OOPSLA 2005*, 2005.
25. T. Rompf, I. Maier, and M. Odersky. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 2009. ACM.
26. T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*, 2010.
27. 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.
28. `http://scalagwt.github.com/`.
29. J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.
30. J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP'07: 21st European Conference on Object-Oriented Programming*, 2007.
31. 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.
32. W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248:211–242, October 2000.
33. V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky. Stagedsac: A case study in performance-oriented dsl development. In *PEPM*, 2012.
34. E. Visser. Webdsl: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer Berlin / Heidelberg, 2008.
35. J. C. Vogt. Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany, 2011.
36. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.