Towards a Tight Integration of a Functional Web Client Language into Scala

Christoph Höger
Technische Universität Berlin
christoph.hoeger@tu-berlin.de

Martin Zuber
Technische Universität Berlin
martin.zuber@tu-berlin.de

ABSTRACT

The web has become an application platform. Therefore it is essential to bring the expressiveness and safety of established languages into the world of web development.

Yet it seems hard if not impossible for any such language to succeed over JavaScript in terms of availability or acceptance. Hence, application developers are practically enforced to embrace JavaScript and at least partially implement their applications in this dynamic language. For this reasons, web applications often contain duplication between type-safe code on the server side and dynamic code on the client side.

In this work we demonstrate how SL, a small, type-safe, purely functional language can be translated into JavaScript from within a Scala web service. We show how type checking and code generation of client code can be integrated into the overall build process of the web application by the means of macro evaluation. Additionally, it is possible to share data-structures between client and server by means of Scala's reflection library.

This tight integration reduces code size, allows early error detection and should generally improve the ability to write good web applications.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Functional Languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Design, Languages

Keywords

Web development, Scala, JavaScript, Static typing, Integration, Macros, Reflection

Scala '13, Montpellier, France

1. INTRODUCTION

Web applications usually follow the Model-View-Controller pattern (see [8]). The border between controller and view is the gap between the server, i.e., an http server, and the client, i.e., an http user agent. Usually the view is in some way "intelligent", i.e., it contains code to perform some validation or assistance operations before invoking an action on the controller. Naturally, this requires the view to have some understanding of the model.

Developing this client-side code for web applications usually means to develop in a dynamic language and most of the time the dynamic language is JavaScript. Thus, even if the server is implemented in Scala, there are always non-trivial parts of the model that have to be implemented in a dynamic language. This is highly unsatisfying for developers used to the productivity and quality of a statically typed functional language like Scala.

The solution to this problem is quite clear: Remove JavaScript from the development completely. It is highly unlikely that any other language (and especially a statically typed one) will see the broad adoption of JavaScript among user agents soon. Therefore, the solution is to compile another language to JavaScript and ship the result. Since the client code is naturally effectful, it is essential that this language encapsulates side-effects in a type safe manner.

In this work we demonstrate how such a client language can be integrated into web development with Scala. We also provide the means to generate some parts of the client code directly from Scala expressions. That way we can reduce code duplication between client and server.

The rest of the paper is organized as follows: First we introduce SL, a small type-safe functional language. Afterwards we describe how SL can be integrated tightly into Scala.

2. THE SL PROGRAMMING LANGUAGE

SL (*Simple Language*) is a small, statically-typed, purely functional programming language that ships with a compiler translating SL programs to JavaScript. Its keyword-heavy style to structure definitions and expressions is inspired by languages like Opal [5], its syntax for expressions and data type definitions is close to Haskell's one [11].

This section illustrates most of the language by example. A full description of SL's grammar can be found in Appendix A.

2.1 Data type and function definitions

A data type definition introduces a type name and one or

more data constructors. Data types may be recursive and parametric. We can define the type of polymorphic lists with the following piece of code:

```
DATA List a = Cons a (List a) | Nil
```

In addition to the types defined by a program's data type definitions, SL has predefined types for integers (`Int`), characters (`Char`), and strings (`String`) respectively, as well as the unit type `Void`. Common functions on these types are defined as built-ins.

Top-level function definitions are pattern based and consist of one or more clauses. The clauses of a function are tried in top-down order until the first matching pattern is found (first-fit pattern matching). As an example, we define the well-known `map` function on lists:

```
DEF map f Nil         = Nil
DEF map f (Cons x xs) = Cons (f x) (map f xs)
```

In addition to regular function definitions the programmer can define custom binary operators. An operator definition is a regular function definition where the operator name is stated infix.

Note, that we do not have to write down types for functions. The SL compiler supports full type inference.

2.2 Expressions

On the expression level SL provides λ-abstractions, application of functions, conditionals, local definitions, and JavaScript quotes.

Function application of a function f to an argument a is written as juxtaposition of f and a without parentheses: $f\ a$. A λ-abstraction introduces an anonymous function. Like in top-level definitions, pattern matching is used for the arguments.

SL has two kinds of conditionals: if- and case-expressions. The condition in an if-expression must be of type `Bool` (defined in the SL prelude). Case-expressions perform pattern-matching for a single expression, i.e., we can write a `length` function for lists in a single clause using a case-expression:

```
DEF length l =
  CASE l OF Nil       THEN 0
         OF Cons x xs THEN 1 + (length xs)
```

Similar to pattern-based top-level definitions, pattern matching uses a top-down first-fit strategy.

An expression may contain local definitions using a let-expression, e. g.

```
LET
  even = \ n. IF n==0 THEN True ELSE odd (n-1)
  odd  = \ n. IF n==1 THEN True ELSE even (n-1)
IN even 5
```

The names introduced in a let-expression may be used in the right-hand sides, even mutually recursive. There is, however, an important restriction due to SL's eager evaluation strategy: In a set of mutually recursive definitions, all right-hand sides must be λ-expressions.

Additionally, SL allows the programmer to "embedded" JavaScript snippets with the help of a hard-wired JavaScript quasiquoter (see [2] or [10]). JavaScript quotes are SL expressions which run inside the `DOM` monad and can be combined using the standard yield and bind functions:

```
yield : a -> DOM a
&=    : DOM a -> (a -> DOM b) -> DOM b
&     : DOM a -> DOM b -> DOM b
```

SL provides a mechanism to reference to SL expressions in the quoted JavaScript code. As an example, let us consider the definition

```
DEF log msg = {| console.log($msg) |})
```

A JavaScript quote is defined using the `{|` and `|}` braces. The JavaScript code given in a quote can refer to any SL value visible in the current scope by prefixing its name with a `$` character. In our example, we refer to the pattern variable `msg` of the current definition to print a given message on the console.

Since JavaScript quotes run inside the built-in `DOM` monad, a plain JavaScript quote always has the type `DOM Void`. To be able to observe a value encapsulated by the `DOM` monad the programmer has to ascribe the JavaScript quote with a corresponding type, e.g.

```
DEF width = {| window.outerWidth |} : DOM Int
DEF height = {| window.outerHeight |} : DOM Int
DEF ratio = width &= (\ w. height &= (\ h.
            yield (w / h)))
```

In this example, we use JavaScript quotes to access the DOM to determine the current height and width of the browser window. The function `ratio` uses these information to calculate the current ratio of the browser window, the type of this function is `DOM Int`.

3. TIGHT INTEGRATION

As already mentioned, we consider a web application to be implemented along the MVC pattern, where the view is "intelligent" and thus shares some functionality with the model and controller implementations. To *integrate* a client-side language into this setting means to safely allow expressions of that language in the view definitions. *Tight* integration is achieved, when the client code operates on the model defined by the server-code. In this section we demonstrate how tight integration can be achieved. We implemented our web applications in Scala Play [1], although any other web framework that supports Scala should also work.

3.1 Compilation

The first step of integration is to provide means of compilation of SL. There are two ways to integrate a language into a Scala program: It might be integrated *internally* by exposing facilities to create a Scala representation of the program programmatically, while *external* integration is achieved by working with the concrete syntax.

As Scala provides good support for easy internal integration (e.g. LMS [14]), this technique is more common. Yet in our use-case, SL is an external DSL (the domain being view interaction) to the Scala web application for two reasons:

- We want to be able to type check SL-expressions at compile-time of the server application. As an internal DSL is implemented by the means of Scala-expressions, any checks of such a language have to be deferred to the execution time.
- Further development of SL will bring features like a module system and separate compilation. An internal DSL would not benefit from those features.

The recent version of Scala (2.10) provides macros. With this feature a developer can partially control the Scala compiler. Besides other interesting aspects, it is astonishingly easy to invoke the SL compiler that way. Below you see the stripped down macro `slci` that allows for direct compilation of inlined SL statements in Scala code:

```
case Expr(Literal(Constant(sl))) => {
  val result = run(prelude::sl.toString::Nil)
  result match {
    case Left(e) => { c.abort(...) }
    case Right(js) =>  {
      val out = preludeJs + "\n" + js
      c.Expr(Literal(Constant(out)))
}}}
```

This relatively simple approach allows us to check and compile SL expressions (expressed as string literals as Scala does not have a quotation mechanism) embedded in arbitrary Scala programs during compilation. It is straightforward to abort compilation and provide meaningful error messages to the user:

```
Scala> slci("DEF x = Cons 1 Nul")
<console>:11: error: Undefined constructor 'Nul'
              slci("DEF x = Cons 1 Nul")
                                   ^
```

Without macros, processing an external language during the compilation of a Scala program could be implemented by the extension of a build system like maven or sbt. A language developer could provide e.g. a plugin that handles the invocation of the compiler, detect changes to sources, etc. While this is a feasible approach for simple tasks like packaging resources, it becomes cumbersome for an integrated DSL: There are many build systems to support, making it practically impossible to support every use-case as well. Even worse, if the DSL shall be usable from inside a non-Scala context like Play's view-templates, the language of that context needs to be parsed and understood well enough to find all relevant (to the DSL) expressions in it.

Another approach would be to implement a compiler plugin for `scalac`. While this would provide a common ground for all users and thus avoid the pitfall of handling multiple build systems, it would still require us to parse and understand the context language. As we want to be able to embed SL in any such context, a more lightweight and standardized embedding is needed.

Naturally, the direct inline approach is somewhat limiting. It is fairly simple to extend the macro to read from an external file from Scalac's class path (we did this with `slc`). Future versions of SL may also provide a file-based output mechanism.

3.2 Model Sharing

While the integration of SL expressions into a Scala program is easy, there is still no simple way to interact between the server and client code.

To re-use the model defined in the Scala code in SL, it needs to be translated into an equivalent datatype definition (a `DATA`-clause, see appendix A). Since SL only supports algebraic datatypes, we restrict this transformation to a subset of the Scala classes:

$$
\begin{array}{lcl}
\mathit{Model} & ::= & \mathit{Base}\ ;\ \mathit{Sub}^{+} \\
\mathit{Param} & ::= & \texttt{[}\ \mathtt{T}_1\ \ldots \mathtt{T}_n\ \texttt{]} \\
\mathit{Base} & ::= & \texttt{sealed trait}\ \mathit{BaseName}\ \mathit{Param} \\
\mathit{Sub} & ::= & \texttt{case class}\ \mathit{TypeName}\ \mathit{Param} \\
 & & (\ \mathit{Field}\ )\ \texttt{extends}\ \mathit{BaseName}\ \mathit{Param} \\
\mathit{Field} & ::= & \mathit{Name}\ :\ \mathit{TypeName}
\end{array}
$$

In this restricted subset, we assume that *BaseName* is a constant local typename, i.e., all *Sub* declaration inherit from the same type) and *TypeName* and *Name* are valid Scala identifiers. To avoid a non well-formed SL declaration, all type parameters of the sub-classes need to be declared at the base class, i.e., *Param* is also a constant for every instance of *Model*. During this section we write $\equiv$ to express equality of language expressions and function definitions since $=$ is a syntactic element of SL.

In a first step, we interpret the sealed base class as an algebraic datatype. The transformation function *Scala2sl* takes the environment of Scala types Γ and a model definition M and yields an SL data type definition:

$$\mathit{Scala2sl} : (\mathit{TypeName} \times \mathit{Model}) \times \mathit{Model} \rightarrow \mathit{DEF}$$

The result of *Scala2sl* is a `DATA`-clause of the same name as the base class. All type parameters $\mathtt{t}_i$ are preserved and converted to lowercase (a syntactical requirement of SL).

$$
\begin{array}{l}
\mathit{Scala2sl}(\Gamma, M) \equiv \texttt{DATA}\ N\ \mathtt{t}_1 \ldots \mathtt{t}_n = P_1 \mid \ldots \mid P_m \\
\quad \text{where} \\
M \equiv \texttt{sealed trait}\ N\ [\mathtt{T}_1\ \ldots\ \mathtt{T}_n]\ ;\ S_1\ \ldots\ S_m \\
P_i \equiv \mathit{sc2Sl}(\Gamma, S_i)
\end{array}
$$

`case`-classes are interpreted as constructor definitions. To that end, every subclass S_i is handed over to the helper function *sc2Sl*, which returns a constructor definition:

$$\mathit{sc2Sl} : (\mathit{TypeName} \times \mathit{Model}) \times \mathit{Sub} \rightarrow (\texttt{cons} \times \mathit{Type}^{*})$$

Again, the name of the constructor is taken directly from the Scala declaration. Every field of the `case`-class is preserved. The translation depends on the type: If the field is a parametric type or an SL-built-in (like Integer or String), it can be directly translated to an SL-typename. If the type is again an algebraic datatype, the translation proceeds recursively. In any other case the function fails (as general classes cannot be translated):

$$
\begin{array}{l}
\mathit{sc2Sl}(\Gamma, S) \equiv M\ \mathtt{R}_1\ \ldots\ \mathtt{R}_k \\
\quad \text{where} \\
S \equiv \texttt{case class}\ M\ [\mathtt{T}_1\ \ldots\ \mathtt{T}_n](\ f_1 : F_1 \ldots f_k : F_k\ ) \ldots \\
\mathtt{R}_i \equiv \begin{cases} F_i & \text{if } F_i \text{ is built-in to SL} \\ \mathit{type} & \text{if } \mathit{Scala2Sl}(\Gamma, \Gamma(F_i)) \equiv \texttt{DATA}\ \mathit{type} \ldots \\ \mathtt{t}_j & \text{if } F_i \equiv \mathtt{T}_j \end{cases}
\end{array}
$$

If one collects the results of the recursive invocations of *Scala2Sl*, a whole hierarchy of classes might be compiled. As Scala uses its own mechanism to normalize operator-names, an additional renaming might be necessary (e.g. Scala's `cons` operator, `::`, translates to `$colon$colon`).

We implemented the translation and made it available to our macro implementation of SL by adding an additional type parameter to the macro. Thus the user can provide both a source file name and a model type to the macro. To verify this method, we used the SL abstract syntax itself as

the model. Below is a small snippet of that abstract syntax containing the definition of an SL data type:

```
sealed abstract class ASTType
...
case class TyExpr(conType: TConVar,
  typeParams: List[ASTType],
  attribute: Attribute) extends ASTType
```

The corresponding (generated) SL type is:

```
DATA ASTType =
  ...
  TyExpr String (List ASTType) Attribute |
  ...
```

Together with a (handwritten) conversion of Scala objects to the corresponding SL runtime values, we were able to provide a simple web service for parsing and pretty printing type signatures. The pretty printing is completely handled by SL.

4. RELATED WORK

JavaScript has become a platform for many projects, ranging from languages compiling to JavaScript, languages enhancing JavaScript, e.g. with additional security features or static typing, as well as ports of existing languages. A rich source for all these projects is *altjs.org*.

The main feature of the SL programming language presented in this paper is the use of JavaScript quotes in a statically-typed, functional language. Therefore, this related work section focuses on languages and tools combining a statically typed environment with JavaScript.

Bazerman's *JMacro* library [3] is a Haskell library for the programmatic generation of JavaScript code. The library ships with a standalone executable which can be used to compile JMacro syntax directly to JavaScript. JMacro provides a simple, lightweight quasi-quoted syntax which is statically checked at compile time. Additionally, JMacro expressions may contain anti-quoted Haskell code. This code may generate further JMacro code, or it may generate any of a range of standard Haskell types, which are able to be marshalled into JMacro through type class methods.

In contrast to JMacro's library approach, Ekblad's *Haste* compiler [6, 7] generates JavaScript code from Haskell. Haste comes with a basic environment for writing client side web applications in a reactive fashion. The main goal of this project is to provide a drop-in replacement for *GHC* that generates relatively lean code.

Instead of translating an existing language into JavaScript, McKenna presented *Roy* [12], a statically typed, functional programming language targeting JavaScript with the goal of yielding light-weight, readable output. Roy tries to meld JavaScript semantics with features common in static functional languages, like Hindley-Milner type inference, pattern matching, structural typing, and monad syntax. The Roy compiler is written in JavaScript, which lets it compile source code inside the browser and execute it on the fly.

Meyerovich et al. take on the same idea of building up a language on top of JavaScript and presented *Flapjax*, a language designed for contemporary web applications [13]. They argue, that the language's event-driven, reactive fashion provides a natural programming model for web applications and compare it to the callback-driven programming style used in JavaScript. Flapjax is implemented using a "languages as libraries" approach [15], which allows the programmer to compile Flapjax source code to JavaScript as well as to use this source code with little extra effort as a JavaScript library.

Kossakowski et al. proposed an approach to embed JavaScript as a DSL in Scala based on Lightweight Modular Staging (LMS) [9]. The JavaScript DSL is statically typed through the host language, but still allows the programmer to use dynamic typing which might come in handy when incorporating external JavaScript libraries and APIs. The embedding allows the authors to implement advanced abstractions in the host language and thus enhance the DSL by re-using host language features, e.g. they facilitate the selective CPS transformation already existing in Scala to provide an abstraction over the typically callback-driven programming style used in JavaScript. To evaluate their approach the authors integrated the DSL in the Play web framework, too. In this setting, code sharing between client and server can be achieved by executing the JavaScript DSL directly in the host language.

Regarding the integration of JavaScript, SL builds upon some of the ideas used in the languages and tools above. Being an external DSL, syntactic quasi-quotation is used to combine JavaScript and SL code. This approach is close to the one used in JMacro, even though the SL compiler does not provide any static checks for the quoted JavaScript code. As with Roy and Haste, we favored a stand-alone compiler for our language over the EDSL approach used in the JMacro library.

5. CONCLUSION

Scala has always made it easy to embed another language into programs or libraries. As a functional language it is also well suited for the development of a complete language toolchain. With the addition of macros it is now easily possible to embed such a toolchain. With the help of the reflection API one can also directly exchange data type definitions between Scala and the embedded language.

In case of the web development domain this means that a server side application written in Scala can easily provide type-safe client-side code without having to give up cross platform support. Additionally, the tight integration of client code into the server can save some development effort by allowing direct usage of data structures written in Scala.

5.1 Future Work

The implementation so far is merely a demonstration of the base technique. There are multiple fields of on going research: As a first step it is obviously dissatisfactory that a user still has to encode object values into the SL representation manually. In theory, the system could also generate the interaction code, i.e., the AJAX function calls, between client and server. Secondly, a broader evaluation regarding the effectiveness of the used approach is needed, especially when it comes to integrating existing JavaScript libraries.

Further down the road it is also foreseeable that SL will not be a broadly adopted web development language (as it is more or less developed for teaching purposes). It would e.g. certainly be an interesting undertaking to implement this method for a restricted subset of Scala.

6. ACKNOWLEDGMENTS

The first version of the SL language as well as a corresponding Haskell front-end was developed by Andreas Büchele, Florian Lorenzen, and Judith Rohloff. Initial versions of the JavaScript back-end and the JavaScript quotes where developed as part of a bachelor thesis and a student project. The current Scala implementation of the SL compiler was developed with the help of Fabian Linges.

7. REFERENCES

[1] Play Framework. `http://www.playframework.com/`.

[2] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '99, pages 4–12, San Antonio, Texas, USA, Jan. 1999.

[3] G. Bazerman. JMacro. `http://www.haskell.org/haskellwiki/Jmacro`.

[4] A. Büchele, C. Höger, F. Linges, F. Lorenzen, J. Rohloff, and M. Zuber. The SL language and compiler. `https://github.com/choeger/sl2`, 2013.

[5] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In *Proceedings of the International Conference on Programming Languages and System Architectures*, pages 228–244, Zürich, Switzerland, Mar. 1994.

[6] A. Ekblad. The Haste Compiler. `https://github.com/valderman/haste-compiler`.

[7] A. Ekblad. *Towards a Declarative Web*. Master of Science Thesis, University of Gothenburg, 2012.

[8] E. Gamma, R. Helm, and R. E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1st edition, 1994.

[9] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript as an Embedded DSL. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434. Springer Berlin Heidelberg, June 2012.

[10] G. B. Mainland. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 73–82. ACM, 2007.

[11] S. Marlow, editor. *The Haskell 2010 Language Report*. 2010.

[12] B. McKenna. Roy: A Statically Typed, Functional Language for JavaScript. *IEEE Internet Computing*, 16(3):86–91, 2012.

[13] 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, Orlando, Florida, 2009. ACM, New York, USA.

[14] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9h International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, Eindhoven, The Netherlands, 2010. ACM, New York, USA.

[15] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, San Jose, California, USA, June 2011. ACM, New York, USA.

APPENDIX

A. SL GRAMMAR

We use standard EBNF descriptions for the languages' syntax. Symbols in `monospaced` font represent terminals, symbols in *italics* represent nonterminals. We furthermore use the following operators:

α^*	zero ore more repetitions of α
α^+	one ore more repetitions of α
$\alpha \otimes$ `T`	zero ore more repetitions of α separated by `T`
$\alpha \oplus$ `T`	one ore more repetitions of α separated by `T`

The grammar of SL is shown in Fig. 1, where the lexical sorts `var`, `cons`, and `type` range over identifiers, constructor names, and type names. A full description of the lexical structure of SL is omitted here for brevity reasons, but can be found in the SL language report [4].

Program	::=	*Def*$^+$
Def	::=	`FUN var :` *Type*
	|	`DEF var` *PPat** `=` *Expr*
	|	`DATA type var`* `=` $(\texttt{cons}\ \mathit{Type}^*) \oplus \texttt{\|}$
PPat	::=	`var` | `cons` | `(` `cons` *PPat** `)`
Type	::=	`var` | `type` | `(` `type` *Type** `)`
	|	*Type* `->` *Type*$\oplus$`->` | (*Type* `->` *Type*)
Expr	::=	`IF` *Expr* `THEN` *Expr* `ELSE` *Expr*
	|	`\` *PPat*$^+$ `.` *Expr*
	|	`CASE` *Expr Alt*$^+$
	|	`LET` *LocalDef*$^+$ `IN` *Expr*
	|	*Expr Expr*
	|	`(` *Expr* `)`
	|	*Expr* `binop` *Expr*
	|	*JSQuote*
	|	`var` | `cons` | [`-`] `num` | `char` | `string`
JSQuote	::=	`{|` `string` `|}` [`:` *Type*]
LocalDef	::=	`var =` *Expr*
Alt	::=	`OF` *Pat* `THEN` *Expr*
Pat	::=	`var` | `cons` | `cons` *PPat*$^+$

Grammar 1: The grammar of SL.