Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Franklinstr. 28/29
10587 Berlin

Diplomarbeit

Integration von funktionalen Web-Client- und Server-Sprachen am Beispiel von SL und Scala

Tom Landvoigt, Matrikelnummer: 222115

15. Juli 2014, Berlin

Prüfer:	Prof. Dr. Peter Pepper
	Prof. Dr.-Ing. Stefan Jähnichen
Betreuer:	Martin Zuber
	Christoph Höger

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

SL	Simple Language	v
JS	JavaScript	iii
AST	Abstract Syntax Tree	13
TUB	Technische Universität Berlin	1
MVC	Model View Controller	16
URL	Uniform Resource Locator	18

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 15. Juli 2014

_____

Unterschrift

Einleitung

Das World Wide Web ist ein integraler Bestandteil unseres Lebens geworden. Ein großer Teil der Software mit der wir in Berührung kommen, benutzt Webseiten als Benutzerschnittstelle. Deshalb muss sich jede moderne Programmiersprache daran messen lassen wie leicht es ist mit ihr Webprojekte zu erstellen. Daher bieten Java, Scala, Ruby und viele andere Programmiersprachen Frameworks an um schnell und einfach strukturierte Webprojekte zu erstellen. Ein gemeinsames Problem dieser Frameworks ist es, insbesondere mit dem aufkommen von Rich Internet Applications, das clientseitig Code ausgeführt werden muss. In diesem Bereich hat sich JavaScript (JS) zum Quasistandard entwickelt[1]. Dadurch ist man beim Schreiben von Browser-seitigen Funktionen auf die von den JS-Entwicklern bevorzugten Programmierparadigmen wie dynamische Typisierung festgelegt. Bei größeren Bibliotheken kann dies die Wartung und Weiterentwicklung erschweren.

Innerhalb der letzten Jahre kam es zu Entwicklungen die eine mögliche Lösung für dieses Problem bieten. Einerseits haben Büchle et al. im Rahmen eines Projektes an der Technische Universität Berlin (TUB) einen Compiler entwickelt, der die Typ-sichere, funktionale Sprache Simple Language (SL) nach JS übersetzt [BHL+13]. Andererseits wurde durch die Einführung von compiler makros [Bur13], die Metaprogrammierung innerhalb von Scala erheblich vereinfacht.

Mit Hilfe dieser beiden Voraussetzungen konnte SL als Abstraktion für JS in Scala eingebettet werden [HZ13] um Probleme mit der dynamischen Typisierung von JS zu lösen. Dazu wurde eine Beispielanwendung im Play Framework geschrieben [Pla].

Diese Einbettung wurde im Zuge dieser Diplomarbeit erweitert. Nun ist es möglich Scala-Funktionen und -Werte in einem gewissen Rahmen automatisch zu übersetzen und typsicher im SL Code zu benutzen.

Mit Scala Werten ist der Inhalt einer Scala Variable gemeint. Wenn dieser sich in SL abbilden lässt, kann er jetzt in statischen SL Code eingebunden werden. Für Scala-Funktionen werden eine SL Funktionen erzeugt, die dann die entsprechenden Scala-

[1] Es gibt weitere Alternativen wie Java oder Flash, die aber Browserplugins vorraussetzen.

Funktionen aufrufen, falls sich die Ein- und Rückgabetypen in SL Typen übersetzen lassen.

Dazu wurde die Möglichkeit geschaffen Scala Type und Werte in SL zu übersetzen sowie zwei Makros geschrieben, die die Einbettung von SL in Scala ermöglichen.

Für das Verständnis der Diplomarbeit werden Kenntnisse im Bereich funktionaler Programmierung sowie Grundlagen in den Sprachen Scala und JS vorausgesetzt.

Im 1. Kapitel wird die Sprache SL kurz vorgestellt. Daraufhin wird beschrieben wie Scala Typen und Werte in SL übersetzt werden (Kapitel 2). Dies wird im 3. Kapitel benutzt um mit Hilfe der dort beschriebenen Makros SL in Scala einzubetten. Darauf hin wird kurz auf den SL Compiler eingegangen und welche Erweiterungen im Zuge dieser Arbeit an ihm gemacht wurden. Abschließend wird diese Einbettung von JS in Scala mit anderen Varianten aus dem Scala Universum verglichen (Kapitel 5).

1. Einführung in Simple Language

Zunächst wird im diesem Kapitel die Sprache SL vorgestellt, da sie essenziell für das Verständnis dieser Arbeit ist.

SL ist eine einfache strikt getypte funktionale Sprache, die als Lehrsprache für den Studienbetrieb der TUB entwickelt wurde. SL hat einen sehr einfach modularen Compiler. Das ermöglicht es leicht neuer Konzepte aus zu probieren. In den folgenden Abschnitten werden die für diese Arbeit relevanten Eigenschaften erklärt.

1.1. Struktur eines SL Programms

Im Rahmen des Compilerbauprojekts im Sommersemester 2013 wurde SL von den Studierenden um die Möglichkeit der Modularisierung erweitert [BJLP13]. Daher besteht ein SL-Programm aus einer Menge von Modulen. Ein Modul ist eine Quelldatei mit der Endung '.sl'. In ihm können Funktionen und Typen definiert werden. Durch die Übersetzung eines SL Moduls werden zwei Dateien erzeugt. Die Datei mit der Endung '.ls.js' enthält den ausführbaren JS-Code. Die zweite Datei mit der Endung '.signature' enthält Informationen darüber welche Funktionen und Datentypen in anderen Modulen verwendet werden können. Das Modul prelude.sl beschreibt alle vordefinierten Funktionen und Datentypen und wird in alle Programme eingebunden.

1.2. Syntax von SL

Im folgenden wird die Syntax von SL anhand des Beispielprogramms in Listing 1.1 erklärt.

1.2.1. Import von Modulen

Mit `IMPORT "<Pfad>" AS <Bezeichner>` können Module nachgeladen werden. Typen und Funktionen die aus Fremdmodulen benutzt werden müssen mit dem `<Bezeichner>` qua-

Listing 1.1: Beispielmodul

```
1  -- Kommentar
2
3  IMPORT "std/basicweb" AS Web
4  IMPORT EXTERN "foo/_bar"
5
6  DATA StringOrOther a = Nothing | StringVal String | OtherVal a
7
8  PUBLIC FUN getOtherOrElse : StringOrOther a -> a -> a
9  DEF getString (OtherVal x) y = x
10 DEF getString x y = y
11
12 PUBLIC FUN main : DOM Void
13 DEF main = Web.alert(intToString (getOtherOrElse(exampleVar, 3)))
14
15 FUN exampleVar : StringOrOther Int
16 DEF exampleVar = OtherVal 5
17
18 FUN getDocumentHight : DOM Int
19 DEF getDocumentHight = {| window.outerHeight |} : DOM Int
```

lifiziert werden. Ein Beispiel dafür ist `Web.alert(...)`.

Mit `IMPORT EXTERN` können JS-Quelldateien eingebunden werden. In diesem Fall wird der Inhalt der Datei `\_bar.js` im Ordner `foo` an den Anfang des Kompilats kopiert.

1.2.2. Basistypen

`DOM a` und `Void` sind einige der Vordefinierten Typen. `Void` bezeichnet den leeren Typen, also keinen Rückgabewert. `DOM a` ist der Typ der JS-quoating Monade. Mit ihr können JS Snippets in SL eingebunden werden (Beispiel: `{| window.outerHeight |} : DOM Int`). Weiter vordefinierte Typen sind `Char` und `String` um Zeichen(ketten) darzustellen, sowie `Int` für ganzzahlige Werte und `Real` für Gleitkommazahlen. Der letzte vordefinierte Typ ist `Bool` für boolesche Werte.

1.2.3. Funktionsdefintionen

Die optionale Signatur einer Funktion kann mit `FUN <Funktionsname> : <Typ>` angegeben werden. Wenn ein `PUBLIC` vorgestellt wird, ist die Funktion auch außerhalb des Moduls sichtbar. Darauf folgen eine oder mehrere pattern basierte Funktionsdefinition der Form `DEF <Funktionsname> = <Funktionsrumpf>`.

1.2.4. Programmeinstiegspunkt

Ein Spezialfall bildet die Funktion `main`. Sie bildet den Einstiegspunkt in ein SL Programm. Sie hat den festen Typ `DOM Void`.

1.2.5. Typdefinitionen

Mit `DATA <Typname> [<Typprameter> ...] = <Konstruktor> [<Typparameter> ...] | ...` können eigene Typen definiert werden. Wie wir Scala Typen und Werte nach SL und zurück übersetzen wird Stoff des Kapitels 2 sein.

2. Model Sharing

Im Zuge dieser Arbeit sollten Scala-Werte und -Funktionen in SL eingebettet werden. Dazu muss einem Scala Typ ein SL Typ zugeordnet und ihre Werte in einander überführt werden. Dies wird im allgemeinen unter dem Begriff Model Sharing zusammengefasst. Betrachten wir dazu beispielhaft die Scala Funktion `scala_foo` im Listing 2.1.

Listing 2.1: Beispielfunktion scala_foo

```
1 def foo( i: Float ): Double = {...}
```

Für die Typen `Float` und `Double` müssen wir ihre SL-Entsprechung finden. Um die Implementation zu vereinfachen setzen wir voraus, das jedem Scala Typ genau ein SL-Typ zugeordnet wird. Andernfalls müssten wir für alle möglichen Permutationen einen SL-Funktionsrumpf erstellen. Bei eingebetteten Scala Werten müsste der SL-Code analysiert werden, um die passende Übersetzung zu finden[1]. Wir erhalten die partielle Funktion $translate_{type}(Type_{Scala}) = Type_{SL}$. Diese wird in Abschnitt 2.1.3 behandelt.

Haben wir einen passenden Typen gefunden, müssen auch die Werte in einander überführt werden. Dies sollte eine bijektive Abbildung sein. Das dies nicht immer möglich ist, wird in Abschnitt 2.2 behandelt.

Für `Float` und `Double` ist der SL Typ `Real` die semantisch beste Wahl. Im Ergebnis erhalten wir schematisch die SL-Funktion `sl_foo` aus Listing 2.2.

Listing 2.2: Übersetzung von scala_foo

```
1 FUN sl_foo : Real -> Real
2 DEF sl_foo p0 = double_to_real (call_via_ajax (
3                   scala_foo (real_to_float p0)
4                 ) )
```

[1] Das ist keine besonders große Einschränkung, da wie wir später sehen werden, das das Typsystem von SL sehr einfach ist und dadurch viele Scala-Typen auf ein und den selben SL Typen abgebildet werden.

2.1. Typübersetzung

In den nächsten Abschnitten wird die Typübersetzung betrachtet. Also welche Scala Typen mit welchen SL Typen assoziiert werden. Dazu werden die beiden Typsysteme kurz erläutert und dann die Funktion $translate_{type}$ näher beschrieben.

2.1.1. SL Typsystem

Das Typsystem von SL ist besteht aus einer Reihe von vordefinierten Typen. Vordefiniert sind `Int`, `Real`, `Char`, `String`, `Bool` und `Void` sowie der Typ der JS-Quoting Monade `DOM a`.

Mit dem Stichwort `DATA` können eigene Konstruktor-/Summentypen definiert werden [PH07, S. 123].

Bei Typdefinitionen werden Typen groß und Typvariablen klein geschrieben. Über Typvariablen können allgemeine Typen definiert werden, die für den Gebrauch spezialisiert werden. Mögliche Spezialisierungen für den Typ `Either a b` wären zum Beispiel `Either Int Real` oder `Either Void String`.

Listing 2.3: Beispiele für selbst definierte Datentypen in SL

```
1 -- Summentyp
2 DATA Fruits = Apple | Orange | Plum
3
4 -- Konstruktortyp
5 DATA CycleKonst = Cycle Int Int
6
7 -- Mischung aus Konstruktor- und Summentyp mit Typvariablen
8 DATA Either a b = Left a | Right b
```

2.1.2. Scala Typsystem

Das Scala Typsystem in Gänze zu erklären würde den Rahmen dieser Arbeit bei weitem sprengen [Ode13]. Im Rahmen dieser Arbeit wurden nur einige wenige vordefinierte Typen übersetzt.

Scala ist strikt Objektorientiert. Es kennt keine primitiven Typen. Alle Typen sind Objekte, aber es gibt vordefinierte Objekttypen die den primitiven Datentypen von Java zugeordnet werden können [Pag13]. Im Folgenden werden die Typen `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char`, `String` und `Unit` trotzdem als die primitiven Typen

von Scala bezeichnet. Die Vererbungshierarchie einiger vordefinierter Objekttypen kann dem Bild 2.1 entnommen werden.

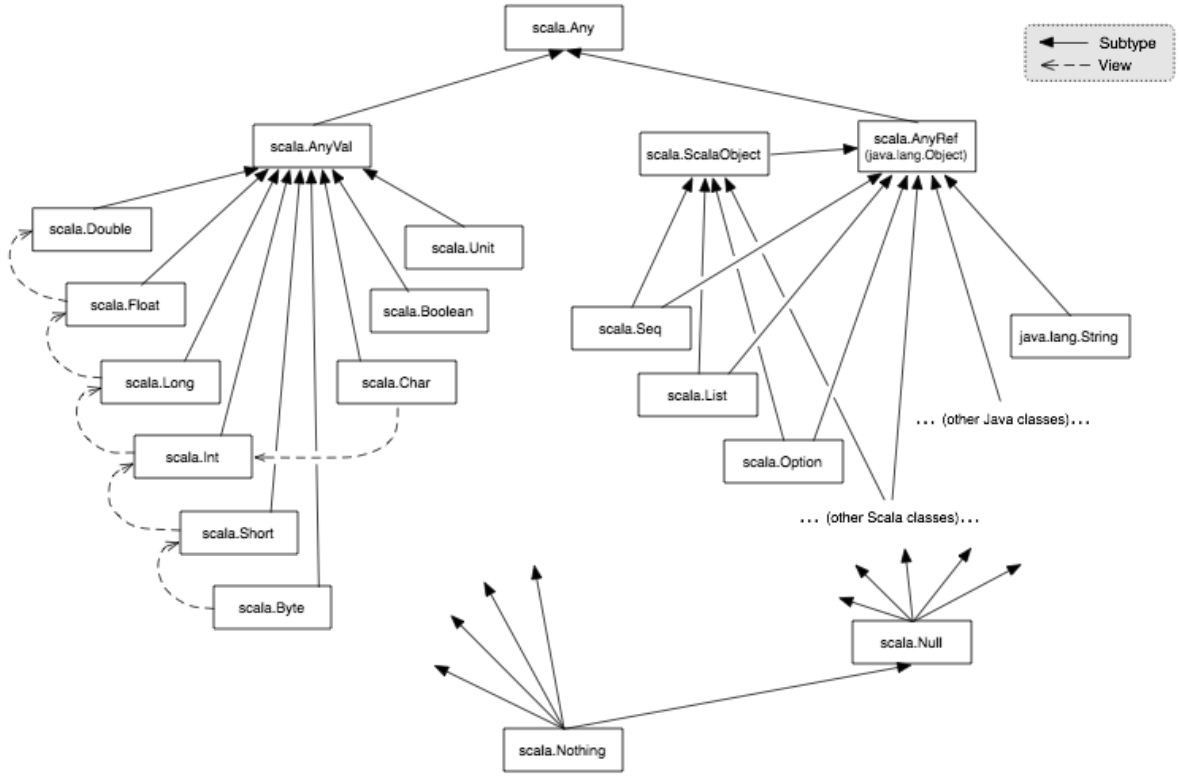

Abbildung 2.1.: Vererbungshierarchie einiger Scala Klassen [Unb09]

Es gibt in Scala-Konstrukte, die den selbst definierten Typen aus SL sehr ähnlich sind. Das wird anschaulich am Beispiel von Option (siehe Listing 2.4). Für sie kann die Übersetzung formalisiert werden. Mehr dazu im Abschnitt 2.1.3. Es wurden aber auch andere Typen wie `Seq[A]` übersetzt, deren innere Struktur sich stark von ihrem SL Äquivalent `List a` unterscheiden.

2.1.3. Funktion $translate_{type}$

Bei der Wahl eines SL Partnertyps für einen Scala Typ sollte auf zwei Bedingungen geachtet werden:

1. Die Typen sollten semantisch ähnlich sein.
2. Es sollte eine semantisch sinnvolle bijektive Abbildung zwischen den Werten der beiden Typen existieren.

Listing 2.4: Option in SL und Scala

```
1 Option in SL:
2 PUBLIC DATA Option a = Some a | None
3
4 Option in Scala:
5 sealed abstract class Option[+A] ... { ... }
6
7 final case class Some[+A](x: A) extends Option[A] { ... }
8
9 case object None extends Option[Nothing] { ... }
```

Wie wir im Abschnitt 2.2 sehen werden, wird die zweite Bedingung für einige primitiven Datentypen von Scala verletzt. Insbesondere für die ganzzahligen Primitiven kann sie nicht eingehalten werden. Da dadurch eine entsprechende Fehlerbehandlung unumgänglich wurde und um die Bedienung zu erleichtern wurden alle Fließkommaprimitiven von Scala mit `Real` und die ganzzahligen Primitiven mit `Int` assoziiert.

Tabelle 2.1.: Die Funktion $translate_{type}$

Scala Typ	Byte Short Int Long	Float Double	Char	Boolean	Unit	String	Seq[A]	Option[A]
SL Typ	Int	Real	Char	Bool	Void	String	List a	Option a

Bei generischen Datentypen wie `Seq[A]` folgt aus den oben genannten Bedingungen, das die Anzahl der Typparameter der Partnertypen gleich sein sollte. Wenn ein generischer Datentyp übersetzt werden soll, wird versucht die Typparameter rekursiv zu übersetzen. Ist dies möglich kann auch der gesamte Typ übersetzt werden. Also `Seq[Option[Long]]` würde zu `List Option Int` übersetzt werden. Eine vollständige Auflistung von $translate_{type}$ findet sich in Tabelle 2.1.

2.1.4. Formalisierung von $translate_{type}$

Bei primitiven Datentypen in Scala wurden ihre SL-Äquivalente durch ihre semantische Gleichheit vorgegeben (siehe Tabelle 2.1).

Für bestimmte Typ-Konstrukte aus Scala kann die Übersetzung formalisiert werden. Sei dafür ein $Model$ durch folgende Grammatik beschrieben:

$$
\begin{array}{lll}
\mathit{Model} & ::= & \mathit{Base}\ ;\ \mathit{Sub}^{+} \\
\mathit{Param} & ::= & \mathbf{[\ V_1, \ldots, V_n\ ]} \\
\mathit{Base} & ::= & \textbf{sealed abstract class}\ \mathit{BaseName}\ \mathit{Param} \\
\mathit{Sub} & ::= & \textbf{final case class}\ \mathit{TypeName}\ \mathit{Param}\ (\ \mathit{Field}^{+}\ )\ \textbf{extends}\ \mathit{BaseName}\ \mathit{Param} \\
 & | & \textbf{case object}\ \mathit{TypeName}\ \textbf{extends}\ \mathit{BaseName}\ \mathbf{[\ Nothing, \ldots, Nothing\ ]} \\
\mathit{Field} & ::= & \mathit{Name}\ \mathbf{:}\ \mathit{TypeName}
\end{array}
$$

Dabei sei $BaseName$ ein fester lokaler Typname, also alle Definitionen von Sub erben von der gleichen abstrakten Klasse und $TypeName$ und $Name$ sind gültige Scala-Bezeichner. Damit wir diese Art von Klassen in einen SL-Typ übersetzen können, müssen alle Typvariablen die in den Sub Definitionen benutzt werden bereits in der $Base$-Klasse definiert werden. Also $Param$ ist konstant für jede Instanz von $Model$.

Dann kann $translate_{type}$ folgendermaßen definiert werden, wobei $=$ durch $\hat{=}$ ersetzt wird da es in SL Teil der Syntax ist:

$$translate_{type}(Model) \mathrel{\hat{=}}$$
$$\textbf{DATA}\ t_{Name}(BaseName)\, t_{Param}(Param) = t_{Case}(Sub_1)\ |\ \ldots\ |\ t_{Case}(Sub_n)$$

mit:

$$t_{Param}(\mathbf{[\ V_1, \ldots, V_n\ ]}) \mathrel{\hat{=}} t_{TypVar}(\mathbf{V_1}) \ldots t_{TypVar}(\mathbf{V_n})$$

$$t_{Case}(\textbf{final case class}\ TypeName\ Param\ \mathbf{(N_1 : T_1, \ldots, N_n : T_n)}\ \textbf{extends}\ \ldots) \mathrel{\hat{=}}$$
$$t_{Name}(TypeName)\, t_{Type}(T_1) \ldots t_{Type}(T_n)$$
$$t_{Case}(\textbf{case object}\ TypeName\ \textbf{extends}\ BaseName\ \mathbf{[\ Nothing, \ldots, Nothing\ ]}) \mathrel{\hat{=}}$$
$$t_{Name}(TypeName)$$

und

$$t_{Type}(x) \mathrel{\hat{=}} \begin{cases} t_{TypeVar}(x) & |\, x \in \{V_1, \ldots, V_n\} \\ translate_{type}(x) & |\, \text{sonst} \end{cases}$$

Wobei t_{Name} einem Scala-Bezeichner einen SL-konformen Bezeichner und $t_{TypeVar}$ einer Typvariable eine eindeutige SL-konforme Typvariable zuordnet. Eine komplette Übersetztung ist nur möglich wenn t_{Type} jedes T_x übersetzen kann.

Dieses Schema wurde von einem Schema zum Übersetzen von **sealed trait**s im Paper von Höger et al. [HZ13] inspiriert. Mit Hilfe dieser Schemata können einige der Scala-Typen in SL-Typen übersetzt werden. Wie man im Listing 2.4 sehen kann folgt die

Definition von `Option[A]` dem hier vorgestellten Schema. Die Definition von `Option a` ist eine mögliche Lösung des Aufrufs von $translate_{type}$. Ein weiterer übersetzbarer Typ ist `Either[A]`.

Im Rahmen dieser Arbeit wurden alle Übersetzungen händisch programmiert, da Typen der Standartbibliotheken von Scala und SL mit einander assoziiert wurden[2]. Es wurde sich aber bei einigen Typen an dem hier beschriebenen Schema orientiert. Bei anderen wie `Seq[A]` war dies nicht möglich, da sich die innere Struktur von seinem SL-Äquivalent zu sehr unterscheidet.

2.2. Darstellungsübersetzung

Wie bereits in der Einführung dieses Kapitels erwähnt, wählen wir die Wertübersetzungsfunktionen anhand des Scala Typs. Da SL nach JS kompiliert muss ein Scala Wert entsprechend seines Typs in eine passende JS Darstellung übersetzt werden. Für die Gegenrichtung, also SL nach Scala gilt dies analog.

2.2.1. Übersetzung von primitiven Werten

Vor allem bei der Übersetzung von Primitiven existiert das Problem der unterschiedlichen Wertebereiche. Man kann zwar jeden Wert des Scala Typs `Byte` in einen Wert des SL Typs `Int` übersetzen, aber nicht umgekehrt. In der Tabelle 2.2 werden die Wertebereiche für primitive Typen aufgelistet. Kann ein Wert von einer Darstellungsform nicht in die andere Darstellungsform umgewandelt werden muss dieser Fehler behandelt werden (siehe Abschnitt 2.3.1). Insbesondere bei den ganzzahligen Primitiven ist das Problem unumgänglich. Für ihre Wertebereiche gilt:

$$|\texttt{Int}| < |\texttt{Number}| < |\texttt{Long}|$$

[2]Es wurden also keine Typen für SL zu Scala-Typen generiert. Da Klassenmethoden im Moment nicht automatisch übersetzt werden können, hätte man keine Funktionen die auf den generierten Typen opperieren.

[3]Alle Zahlendatentypen werden in JS durch den primitiven Number Datentyp dargestellt. Dies ist eine Gleitkommazahldarstellung nach dem IEEE 754 Standard mit einer Breite von 64 Bit. In dieser Darstellung können Ganzzahlwerte von $-2^{53}+1$ bis $2^{53}-1$ korrekt dargestellt werde.

[4]Die maximale Länge von Strings in JS und Scala ist Implementationsabhängig.

Tabelle 2.2.: Umfang der primitiven Datentypen in Scala und SL (JS) [Ecm11, S. 28-30] [Ora11]

SL	JS Darstellung	Scala
Int	Number[3] $[-2^{53}+1, 2^{53}-1]$	Byte $[-128, 127]$
Int	Number $[-2^{53}+1, 2^{53}-1]$	Short $[-2^{15}, 2^{15}-1]$
Int	Number $[-2^{53}+1, 2^{53}-1]$	Int $[-2^{31}, 2^{31}-1]$
Int	Number $[-2^{53}+1, 2^{53}-1]$	Long $[-2^{63}, 2^{63}-1]$
Real	Number (IEEE 754 64-Bit)	Float (IEEE 754 32-Bit)
Real	Number (IEEE 754 64-Bit)	Double (IEEE 754 64-Bit)
Bool	Boolean $true, false$	Boolean $true, false$
Char	String (Länge 1) (16-Bit)	Char (16-Bit)
String	String[4] (maximale Länge: ?)	String (maximale Länge: ?)

2.2.2. Übersetzung von komplexen Werten

Bei nicht primitiven Werten ist mehr Aufwand nötig. Dafür müssen wir zunächst die JS-Darstellung von selbst definierten SL Typen verstehen[5].

Listing 2.5: Beispiel eines selbstedefinierten Typs

```
1 DATA People a b = Alice | Bob Int | Cesar a b | Octavian
```

Die einzelnen Konstruktoren erhalten entsprechend ihrer Reihenfolge eine Konstruktor-ID (`_cid`) beginnend bei 0. Hat ein Konstruktor keine Parameter, wird er nur durch seine `_cid` dargestellt. Andernfalls wird ein Objekt erzeugt. Dies besitzt das Attribut `_cid` sowie entsprechend der Anzahl der Parameter Attribute die von `_var0` bis `_varN` benannt sind. Die JS Darstellung von dem Beispieltyp aus Listing 2.5 findet sich in der Tabelle 2.3.

Mit Hilfe dieser Informationen und dem Schema aus Absatz 2.1.4 können jetzt Option-Werte zwischen SL und Scala ausgetauscht werden. In der Tabelle 2.4 wurden beispielhaft einige Werte vom Typ `Option[Int]` übersetzt. In Abschnitt 2.3.1 wird auf die Implementation der Übersetzung noch einmal genauer eingegangen.

Die Übersetzung von anonymen Funktionen also Werten die eine Funktion darstellen ist im Moment nicht möglich, da kein adäquater Weg gefunden wurde um ihre Darstel-

[5]Das beschriebene Schema wurde aus dem SL Compiler generierten Code abgeleitet. Es ist nicht dokumentiert.

Tabelle 2.3.: JS Darstellung des SL Typen `People Char Bool`

SL	JS Darstellung
`Alice`	`0`
`Bob 42`	`{ "_cid" => 1, "_var0" => 42 }`
`Cesar "a" true`	`{ "_cid" => 2, "_var0" => "a", "_var1" => true }`
`Octavian`	`3`

Tabelle 2.4.: Übersetzung von Option Werten

Scala	JS Darstellung	SL
`Option[Int]`		`Option Int`
`Some(15)`	`{ "_cid" => 0, "_var0" => 15 }`	`Some(15)`
`None`	`1`	`None`

lung zwischen Scala und JS aus zu tauschen.

2.3. Erleuterung der Implementation

In der momentanen Implementation wird $translate_{type}$ durch `Seq[AbstractTranslator]` dargestellt. Dadurch ist es möglich auch Teilmengen von $translate_{type}$ zu benutzen. Dabei stellt eine Klasse die von `AbstractTranslator` erbt ein Paar zwischen einem Scala-Typ und einem SL-Typ dar. Die Klasse ist nach dem jeweiligen Scala Typen den sie übersetzt benannt[6]. Die Hauptmethode von `AbstractTranslator` ist `translate` (siehe Listing 2.6). Ihr wird ein Scala Typ übergeben. Wenn der übergebene Scala Typ der Klasse entspricht erhält man als Rückgabewert den entsprechenden SL Typen, die Import Statements um die entsprechenden SL Module zu laden[7] sowie die Abstract Syntax Tree (AST)-Repräsentation der Wertübersetzungsfunktionen von Scala nach SL und umgekehrt. Andernfalls wird `None` zurückgegeben.

Listing 2.6: Hauptfunktion in AbstractTranslator

```
1 def translate
2   ( context: MacroCtxt )
3   ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
4 : Option[( String,
5            Set[String],
```

[6] zB. `SeqTranslator`

[7] Bei primitiven SL Typen sind diese leer. Für den SL Typ `List.List Opt.Option Int` würde `IMPORT "std/option" AS Opt, IMPORT "std/list" AS List` zurück gegeben werden.

```
6             context.Expr[Any => JValue],
7             context.Expr[JValue => Any] )]
```

Weiter Parameter sind `context` und `translators`. `context` ist der Makro Kontext[8]. Er wird benötigt um ASTs aufzubauen und den übergebenen Typen zu prüfen. Mit `translators` wird der Teil von $translate_{type}$ übergeben mit denen Spezialisierungen eines generischen Typs übersetzt werden können.

Möchte man einen Scala Typ nicht nur gegen eine Klasse prüfen kann man die Hilfsfunktion `useTranslators` aus dem companion object von `AbstractTranslator` nutzen.

Listing 2.7: Statische Hilfsfunktion in AbstractTranslator

```
1 def useTranslators
2   ( context: MacroCtxt )
3   ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
4 : Option[( String,
5             Set[String],
6             context.Expr[Any => JValue],
7             context.Expr[JValue => Any] )]
```

`translators` gibt hier an welche Teilmenge der Funktion $translate_{type}$ man nutzen möchte[9].

Die Wertübersetzungsfunktionen haben die Signatur `Any => JValue` bzw. `JValue => Any`. `JValue` ist Teil der json4s Bibliothek [Jso], die benutzt wird um JS-Werte zu erzeugen. Insbesondere übernimmt sie in der aktuellen Implementation die Übersetzung der primitiven Werte.

2.3.1. OptionTranslator als Beispiel

Der `OptionTranslator` bildet die Verbindung zwischen dem Scala-Typ `Option[A]` und seinem SL-Pendant `Option a` ab.

Wie bereits erwähnt ist die Hauptfunktion `translate` (siehe Listing 2.6). In ihr wird zunächst mit reflection überprüft ob der übergebene Typ (`input`) ein Subtyp von `Option[Any]` ist. Durch die Definition von `Option[A]` (`sealed` und `final case` siehe Listing 2.4) können wir uns sicher sein, das übergebenen Werte nur vom Typ `Some[A]` oder `None` sind. Wieder mit reflections wird der Typ der Spezialisierung `A` bestimmt. Ist dieser Typ mit

[8] siehe Kapitel 3

[9] `translators` wird in diesem Fall auch für die Spezialisierungen von generischen Typen benutzt.

Hilfe der übergebenen Translator-Klassen (`translators`) übersetzbar, wird das Ergebnis zusammengestellt. In jedem anderen Fall wird `None` zurückgegeben.

Das Ergebnis besteht aus dem SL-Typ (zB. `Option Int`), der Import-Anweisung (zB. `IMPORT "std/option" AS Opt`) und den ASTs der beiden Wertübersetzungsfunktionen. Die Wertübersetzungsfunktionen werden im companion object definiert um sie besser mit Unit-Tests zu testen. Mit Hilfe der Funktion `reify` aus der Makro API von Scala wird aus Scala-Code der entsprechende AST generiert.

```
reify ( {
( i : Any ) = > OptionTranslator.scalaToJsOption ( i , expr_s2j )
} )
```

`expr_s2j` ist in diesem Fall die Wertübersetzungsfunktion des Typs der Spezialisierung.

```
def scalaToJsOption( input: Any, f: Any => JValue ): JValue = {
  import org.json4s._
  input match {
    case Some( x ) => {
      val tmp: List[( String, JValue )] =
        List( "_cid" -> JInt( 0 ), "_var0" -> f( x ) )
      JObject( tmp )
    }
    case None => JInt( 1 )
    case _ => throw new IllegalArgumentException
  }
}
```

Wird an die Wertübersetzungsfunktion ein unerwarteter Wert (oder bei primitiven Typen ein Wert außerhalb der zulässigen Grenzen) übergeben wird eine `IllegalArgumentException` geworfen.

Möchte man einen einen neuen Scala-Typ übersetzen bzw. zu $translate_{type}$ hinzufügen muss eine neue Translator-Klasse geschrieben werden. Eine entsprechende Anleitung findet sich im Anhang D.2.

3. Scala Compiler Macros

Im 1. Kapitel wurde SL vorgestellt. Eine strikt getypte funktionale Sprache die in JS-Code übersetzt wird.

SL brachte Höger et al. auf die Idee mit Hilfe von Scala Compiler Makros eine strikt getypte Abstraktion für JS in Scala einzubinden [HZ13]. Um ihre Ergebnisse zu demonstrieren haben sie eine Beispielwebanwendung mit Hilfe des Play-Frameworks [Pla] geschrieben.

Play ist ein Model View Controller (MVC)-Framework für Webanwendungen welches in Scala geschrieben ist. Play vereinfacht die Erstellung von strukturierten Webanwendungen. Auch innerhalb dieser Arbeit wird es genutzt, um die Einbettung von SL in Scala zu erproben.

Es konnte das Problem der dynamischen Typisierung von JS behoben werden, aber ein anderes blieb offen. Bis jetzt war es nicht möglich SL- bzw. JS-Code abhängig von der Scala-Umgebung zu generieren. Insbesondere für Webanwendung stellt dies ein Problem dar. Es sollte möglich sein den JS-Code zum Beispiel mit dem Benutzername zu personalisieren.

Mit Hilfe der im Kapitel 2 beschriebenen Einbettung von Scala-Werten/-Typen in SL ist es jetzt möglich dieses Problem zu lösen.

Um den SL-Code von der Scala-Umgebung abhängig zu machen wurden zwei Ansätze verfolgt. Zum einen ist es jetzt möglich Scala-Werte direkt im SL-Code zu nutzen. Dazu werden Platzhalter in den SL-Code eingefügt die dann durch übersetzte Scala-Werte ersetzt werden.

Auf der anderen Seite können aus Scala-Funktionen aus SL-Code heraus aufgerufen werden. Dazu wird aus einer Scala-Funktion eine SL-Funktion generiert, die die Scala-Funktion über asynchrone Kommunikation mit dem Server und Scala reflections aufruft.

Um diese Lösungen zu realisieren wurden zwei Scala Compiler Makros geschrieben. Diese werden im Laufe dieses Kapitels vorgestellt.

3.1. Makros und ihre Abhängigkeiten

Um Scala Funktionen für die Verwendung in SL-Code zu markieren wurde die macro annotation `sl_function` geschrieben, welche im Abschnitt 3.2 behandelt wird. Im darauf folgenden Abschnitt 3.3 wird beschrieben, wie statischer SL Code mit Hilfe des def macros `slci` eingebunden wird und welchen Veränderungen gemacht werden mussten um Scala Werte und annotierte Funktionen benutzen zu können. Beide Makros binden den Trait `MacroConfig` ein, in dem grundsätzliche Konfigurationen definiert sind. Zur Übersetzung der Typen und Werte werden die Translator-Klassen genutzt. Einen Überblick über das Projekt bietet die Abbildung 3.1.

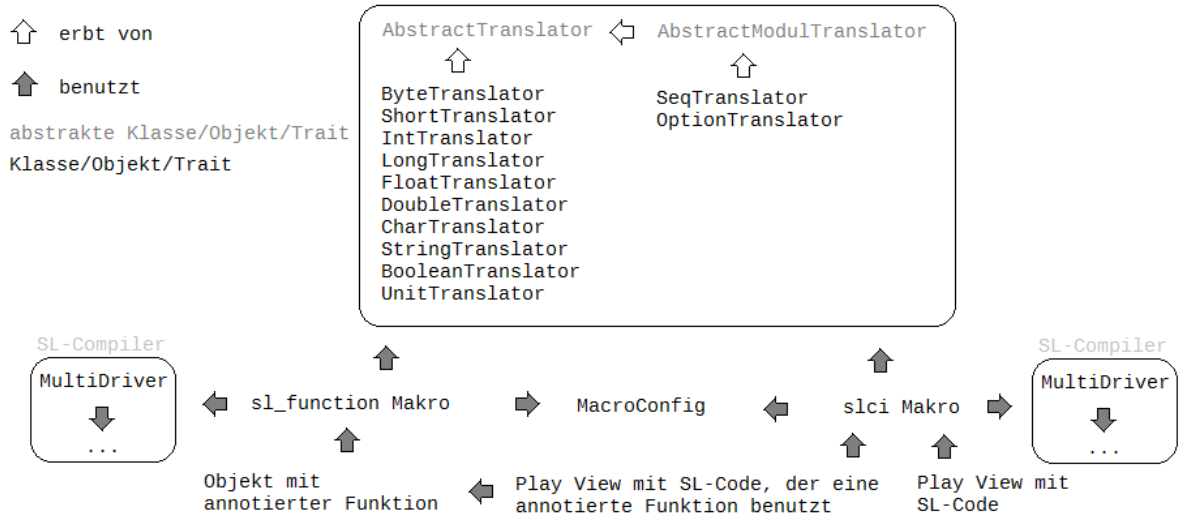

Abbildung 3.1.: Projektübersicht

3.1.1. Konfiguration der Makros

Im Trait `MacroConfig` werden die beiden hier programmierten Makros konfiguriert. In den anschließenden Abschnitten wird sich darauf bezogen. Die wichtigsten Konfigurationsparameter sind:

assets_dir Gibt an in welchem Ordner die SL-Quelldateien und ihre Kompilate liegen. Im Moment `/projekt_ordner/public/sl/`.

inline_sl_macro_folder Gibt den Unterordner von `assets_dir` an, in dem die SL-Module/-Quelldateien und ihre Kompilate liegen, die vom def marco slci generiert werden. Im Moment: `generated_inline/`.

annotation_sl_macro_folder Gibt den Unterordner von `assets_dir` an, in dem die SL-Module, die von der macro annotation sl_function generiert werden, liegen. Im Moment: `gnerated_annotation/`.

inline_sl_macro_handler_uri Gibt die Uniform Resource Locator (URL) an unter welcher der Server auf asynchrone Kommunikation lauscht um per reflection annotierte Scala-Funktionen aufzurufen. Im Moment ist dies eine relative URL: `/ajax`.

Wird in der Konfiguration etwas geändert muss das ganze Projekt neu kompiliert werden. Insbesondere müssen alle generierten SL-Module neu erstellt werden.

3.2. Macro Annotation sl_function

Mit macro annotations kann in den Übersetzungsprozess von Scala eingegriffen werden [Burb]. Es ist möglich den annotierten Code zu verändern[1]. Mit dem geschriebenen Makro können nur Funktionen annotiert werden. Für jede Funktion wird eine Hilfsfunktion und ein SL-Modul erzeugt. Die Hilfsfunktion soll den Aufruf im Rahmen von ajax requests erleichtern. Das SL-Modul ermöglicht es diesen Aufruf typsicher in SL-Programme einzubinden. Beispielhaft wird das Annotieren einer Funktion anhand der im Listing 3.1 beschriebenen Funktion `factorial` betrachtet.

Listing 3.1: Scala Beispielfunktion

```
1 -- Foo.scala
2 package example
3
4 object Foo {
5   @sl_function def factorial( i: Int ): Long = {...}
6 }
```

3.2.1. Anforderungen an eine Funktion

Die zu übersetzende Funktion muss gewisse Anforderungen erfüllen. Wenn wir sie im Rahmen von ajax requests benutzen wollen, muss sie statisch aufrufbar sein, also:

- Sie muss in einem Objekt definiert sein.
- Ihre Signatur darf keine Typparameter enthalten.

[1] Es können Funktionen, Klassen, Objekte, Typparameter oder Funktionsparameter annotiert werden.

- Die Funktion darf nicht als `private` oder `protected` markiert sein.

Andere Anforderungen ergeben sich aus der Implementation bzw. wurden aufgestellt um die Implementation zu erleichtern:

- Die Funktion muss einen Rückgabetyp definieren[2].
- Die Funktion darf nur eine Parameterliste haben[3]
- Die Ein- und Ausgangstypen müssen sich in SL-Typen übersetzen lassen.
- Der Funktionsname darf keine ungewöhnlichen Zeichen enthalten[4]

3.2.2. SL-Modul

Für jede annotierte Funktion wird ein Modul erstellt. Das Modul enthält zwei Funktionen. Jeweils für den asynchronen und synchronen Aufruf der Scala-Funktion über Ajax. Das Ergebnis wird in `Option` gekapselt, um auf Fehler in der Kommunikation mit dem Server reagieren zu können. Das Erzeugen der Ajax Anfrage und das Behandeln des Ergebnisses passiert in den JS-Funktionen `_sendRequestSync()` und `sendRequestAsync()`. Diese Funktionen sind in der JS-Bibliothek std/_scalafun.js definiert. Weiterhin enthält das Modul in Kommentaren den Namen der aufgerufenen Funktion sowie den voll qualifizierten Namen des Objektes in dem die Funktion definiert ist. Diese Informationen werden gebraucht um Abhängigkeiten zwischen der Scala Funktion und ihrer Benutzung in SL-Code aufzulösen. Genauer wird dies im Kapitel 3.3 beschrieben. Das Modul wird direkt nach dem erstellen kompiliert.

3.2.3. Hilfsfunktion

Um den Aufruf mit Ajax Anfragen zu erleichtern wird eine Hilfsfunktion definiert. Sie kapselt die eigentliche Scala Funktion. Sie erhält die Parameter als `JValue`. Die Parameter werden mit Hilfe der Funktionen aus den Translator Klassen in Scala Werte übertragen und dann auf die passenden Typ gecasted. Anschließend wird mit ihnen die eigentlich Funktion aufgerufen. Das Ergebnis wird in ein `JValue` Wert umgewandelt und zurückgegeben.

[2]Andernfalls müsste der Rückgabetyp erst über reflection bestimmt werden.

[3]In der aktuellen Implementation werden die Default-Werte eines Parameters ignoriert. Eine entsprechende warning wird erzeugt.

[4]Da sich der Name der Funktion im Name und Pfad des erzeugten Moduls widerspiegelt, sind nur die Zahlen von 0 bis 9 sowie kleine Buchstaben von a bis z erlaubt. Ähnliche Einschränkungen gelten für die übergeordneten Pakete sowie den Namen des Objekts in dem die Funktion definiert ist.

Listing 3.2: SL-Modul factorial.sl zur Funktion aus Listing 3.1

```
-- DO NOT ALTER THIS FILE! --------------------------------
-- cp: example.Foo
-- fn: factorial
-- ------------------------------------------------------
-- this file was generated by @sl_function macro ----------
-- on 20-06-2014 ------------------------------------------
IMPORT EXTERN "std/_scalafun"
IMPORT "std/option" AS Opt

-- this functions should call the scala function:
-- callable_functions.Examples.factorial
PUBLIC FUN factorialSync : Int ->  DOM ( Opt.Option (Int) )
DEF factorialSync p0 = {| _sendRequestSync( ... ) ($p0) |}
: DOM ( Opt.Option (Int) )

PUBLIC FUN factorialAsync : ( Opt.Option (Int) -> DOM Void )
-> Int -> DOM Void
DEF factorialAsync callbackFun p0 = {| _sendRequestAsync( ... )
($callbackFun, $p0) |} : DOM Void
```

Die Funktionen $scala_to_sl$ und sl_to_scala sind nur Platzhalter. Wie diese Funktionen genau definiert sind hängt von der Implementation der entsprechenden Translator-Klasse ab.

Listing 3.3: Hilfsfunktion zur Funktion aus Listing 3.1

```
-- Foo.scala
package example

object Foo {
  @sl_function def factorial( i: Int ): Long = {...}

  def factorial_sl_helper( p1: org.json4s.JValue ) : org.json4s.JValue = {
    scala_to_sl(factorial(sl_to_scala(p1)))
  }
}
```

3.2.4. Ablauf eines Aufrufs

Betrachten wir nun den Aufrufprozess einer Funktion im Ganzen am Beispiel der Funktion `factorialSync` aus dem Listing 3.2. Folgende Schritte werden durchlaufen:

1. Aufruf der Funktion `factorialSync 5` im SL-Code
2. Aufruf der JS-Funktion `(_sendRequstSync( "\ajax", "example.Foo", "factorial" )) (5)`. Es werden der URL des Ajax-Handlers, der voll qualifizierte Name des Objekts und der Funktionsname übergeben. In einem zweiten Schritt wird der eigentliche Parameter (SL-Codiert) übergeben.
3. Die SL-Parameter werden mit Hilfe der Bibliothek json.js [Cro10] in einen JSON String umgewandelt und mit Funktions- und Objektname als Anfrage an die Adresse des Ajax-Handlers geschickt (siehe Tabelle 3.1).
4. Der Ajax-Handler wandelt die Funktionsparameter (`5`) in `JValue` Werte um [Jso] und ruft dann über reflection die Hilfsfunktion `factorial_sl_helper` auf. Das Ergebnis (`120`) des Aufrufs wird als JSON String zurück an den Client gesendet.
5. Ist die Anfrage an den Server erfolgreich wird `Some(120)` zurückgegeben, andernfalls `None`.

Tabelle 3.1.: Post Parameter der Ajax Anfrage

Parametername	Inhalt
object_name	Voll qualifizierter name des Objekts
function_name	Name der Funktion
params	JSON encodierte Liste der übergebenen Parameter

3.3. Def Macro slci

Bis jetzt kann man nur Funktionen markieren. Nun soll SL benutzt werden um JS-Code zu generieren und ihn auf Benutzerseite zu verwenden. Dazu wurde das `slci` Makro neu geschrieben und erweitert. Im Laufe der nächsten Abschnitte vollziehen wir die Entwicklungsschritte des Makros nach.

Mit def macros kann während des Übersetzungsprozesses von Scala in den Code eingegriffen werden [Bura]. Der Aufruf solch eines Makros verhält sich wie eine Funktion, nur das das Makro die ASTs der Parameter übergeben bekommt und einen AST liefert der den Aufruf des Makros ersetzt. Listing 3.4 enthält einen beispielhaften Aufruf des slci-Makros.

Listing 3.4: Beispielaufruf des slci-Makros in einer Play View

```
1  -- Example.scala.html
2  ...
3  <script type="text/javascript">@{
4  Html(slci(
5  """
6  PUBLIC FUN main : DOM Void
7  DEF main = ...
8  """
9  ))}
10 </script>
11 ...
```

3.3.1. Statischen SL Code übersetzen

Mit der Entwicklung eines Modulsystems für SL musste das Einbetten von statischem Code neu geschrieben werden [BJLP13]. Die erste Version des `slci` Makros nutzte eine Version von SL die JS Code erzeugt. Im Laufe des Studentenprojekts wurde davon Abstand genommen. Das Ergebnis der Übersetzung sind JS-Dateien, die mit Hilfe von require.js in Webseiten eingebettet werden [Req].

Entsprechend wird jetzt vom `slci` Makro ein SL-Modul erzeugt. Die Datei wird entsprechend des Ortes an dem `slci` aufgerufen wird benannt:

```
<Dateiname>.<Zeilennummer>.sl
```

Wenn diese Datei übersetzt werden kann, wird sie mit require.js eingebunden, dass dann die main-Funktion des Moduls aufruft. Andernfalls wird ein Übersetzerfehler erzeugt.

Neben require.js müssen noch andere JS-Bibliotheken geladen werden. Möchte man SL-Code in einer Webseite benutzen, müssen alle Bibliotheken, die in Tabelle 3.2 aufgelistet sind, eingebunden werden.

Tabelle 3.2.: Benötigte JS-Bibliotheken

jquery-1.9.0.min.js	Erleichtert Ajax-Anfragen. Wird vom sl_function-Markro benötigt [The].
sl_init.js	Initialisiert die globale Variable `sl` und konfiguriert require.js. Muss vor require.js geladen werden.
require.js	Wird benötigt um SL-Module nach zu laden [Req].
json.js	zum Umwandeln von JS Werten in ihre JSON-Repräsentation und zurück. Siehe Abschnitt 3.2.4 [Cro10].

3.3.2. Scala Variablen in SL nutzen

Als nächstes wurde die Verwendung von Scala-Variablen in SL-Code implementiert. Anhand des Beispiels im Listing 3.5 werden die dafür nötigen Schritte erklärt.

Listing 3.5: Beispielaufruf des slci Macros mit Scala Variablen

```
1  slci(
2  """
3  IMPORT "std/option" AS Option
4  ...
5  FUN foo : Option.Option Int
6  DEF foo = $s
7  ...
8  """,
9  Some(3)
10 )
```

Die zu ersetzende Stelle wird durch einen Platzhalter (`$s`) markiert. Der $n+1$-te Parameter von `slci` wird dem n-ten Platzhalter zugeordnet. Falls die Anzahl der Parameter ungleich der Anzahl der Platzhalter ist, werden Warnings oder Errors erzeugt.

Daraufhin werden die `IMPORT`-Anweisungen analysiert und die entsprechenden Translator-Klassen geladen[5]. Die von der Makro-API bestimmten Typen[6] der Parameter werden dann mit den zur Verfügung stehenden Translator-Klassen übersetzt. Wenn alle Typen übersetzt werden konnten, werden die Platzhalter durch JS-Quotings ersetzt, die auf globale Variablen zugreifen. Im Beispiel aus Listing 3.5 würde `$s` durch `{| sl['5a40c735438fd9e1fd43657bd`
ersetzt werden. Der so erzeugte SL-Code wird dann, wie im Abschnitt 3.3.1 beschrieben, übersetzt. Listing 3.6 enthält den vom Makro erzeugt Scala-Code.

Die Parameter werden, mit den von den Translator-Klassen erzeugten Übersetzungsfunktionen, in SL-Werte übersetzt. Da sie zuerst als `JValue`-Objekte vorliegen müssen sie noch in JS-Code überführt werden. Im Listing 3.7 findet sich der nach einem Aufruf der Webseite erzeugte JS-Code.

[5] Translator-Klassen die in Standardtypen von SL übersetzen, werden immer geladen. Für `IMPORT "std/option" AS Modulalias` würde die Instanz `new OptionTranslator("Modulalias")` erzeugt werden.

[6] Manchmal muss man den Typ annotieren. Das Literal `5` hat den Typ `Int(5)` und nicht `Int`. Man schreibt also `5:Int`.

[7] Der Name der JS-Variable folgt folgendem Schema: `<Hash des Macrokontexts>scalaParam<Parameternummer`

Listing 3.6: Erzeugter Scala-Code zum Listing 3.5

```
{
"""
require(...);
// transformed scala variables
sl['5a40c735438fd9e1fd43657bd7f8564scalaParam1'] = %s;
""".format( compact( render( scala_to_sl( Some(3) ) ) ) )
}
```

Listing 3.7: JS-Code zum Listing 3.5

```
require(
  [ "generated_inline/example.template.scala.48.sl" ],
  function (tmp) { sl['koch.template.scala.1'] = tmp; }
);
// transformed scala variables
sl['5a40c735438fd9e1fd43657bd7f8564scalaParam1'] = {"_cid":0,"_var0":3};
```

3.3.3. Scala Funktionen in SL nutzen

Im Abschnitt 3.2 wurde erklärt wie Scala-Funktionen für die Verwendung in SL-Code markiert werden. Für die markierten Funktionen werden SL-Module erzeugt. Wenn ein solches Modul geladen wird[8], werden am Anfang des vom Makro erzeugten Scala-Codes `import`-Anweisungen eingefügt, die auf die referenzierten Scala Funktionen verweisen. Falls sich die Signatur der importierten Funktionen ändert, soll der Aufrufende SL-Code neu compiliert werden. Für die Funktion `factorial` aus Listing 3.1 würde der Scala-Code im Listing 3.8 erzeugt werden.

Listing 3.8: Scala `import`-Anweisung für eine annotierte Funktion

```
{
import example.Foo.{factorial => fun3903232409}
"""
require(...);
...
""".format( ... )
}
```

Die Funktion wird unter einem zufallsgenerierten Namen importiert um Namenskonflikten vorzubeugen.

[8]Der Pfad des Moduls fängt in der aktuellen Konfiguration mit `generated_annotation/` an.

4. Erweiterungen am SL-Compiler

Im Laufe der Diplomarbeit wurde der SL-Compiler an einigen Stellen erweitert oder verändert. Die Compilermakros verwenden den im Studierendenprojekt geschriebenen `MultiDriver` [BJLP13, S. 16-19].

4.1. Erweiterungen am MultiDriver

In der vorherigen Version des `MultiDrivers` wurden, wenn ein Modul eine main-Funktion enthält, neben dem Kompilat die Dateien `main.js` und `index.html` erstellt [BJLP13, S. 18-19]. Da dies unerwünscht ist, wenn der SL-Code in eine Play View eingebettet wird, wurde in der Konfiguration (`Configs.scala`) des Compilers eine neue Option eingeführt. Mit dem Schalter `generate_index_html` kann das oben genannte Verhalten unterdrückt werden. Im Normalfall ist dieser Wert auf `true` gesetzt; die Makros verwenden ihn mit dem Wert `false`.

Die übersetzten Bibliotheksmodule (zum Beispiel: option.sl.js) werden in das Zielverzeichnis der Übersetzung kopiert, wenn das zu übersetzende Modul eine main-Funktion enthält und der SL-Übersetzer in Form einer jar-Datei vorliegt. Das ist nötig, damit die entsprechenden Module nachgeladen werden können. Dies ist bis jetzt undokumentiertes Verhalten. In der aktuellen Version des SL-Übersetzers werden die Dateien auch kopiert wenn der Übersetzer nicht gepackt vorliegt.

Weiterhin wurde der Schalter `main_function_is_required` eingeführt. Wenn dieser Wert auf `true` gesetzt ist, wird sichergestellt das ein zu übersetzendes Modul eine main-Funktion enthält. Falls dies nicht der Fall ist wird die Übersetzung mit einem Fehler abgebrochen. Wie im Abschnitt 3.3.1 beschrieben, ist für das slci-Makro eine main-Funktion nötig. Der Standardwert des Schalters ist `false`.

4.2. Überprüfung des Ergebnistyps von JS-Quotings

Mit JS-Quotings kann JS-Code direkt in SL benutzt werden. Bis jetzt wurde das Ergebnis solcher Quotings zur Laufzeit nicht auf Korrektheit überprüft [BJLP13, S. 29]. Im Rahmen dieser Arbeit wurde dieses Verhalten für einige primitive Typen (`String`, `Char`, `Bool`, `Real` und `Int`) geändert. Dies gilt nur für JS-Quotings die einen entsprechenden `DOM a`-Typen haben, wie zum Beispiel im Listing 4.1.

Listing 4.1: Beispiel: JS-Quoting Monade

```
1 FUN foo : DOM Int
2 DEF foo = {| document.getElementById("canvas").width |}:DOM Int
```

Passt das Ergebnis nicht zum Typ wird die Ausführung des Programms mit einer Exception abgebrochen.

5. Related Works

Auch andere Gruppen haben JS in Scala eingebunden. In den folgenden Abschnitten wird versucht einen groben Überblick dazu zu geben und sie mit der hier vorgestellten Implementation zu vergleichen.

Allen Ansätzen ist gemein, das sie JS zu einem statischen Typsystem verhelfen. Was die Wartung großer Projekte in JS erleichtert.

5.1. Scala.js

Scala.js ist ein Compiler. Er übersetzt Scala Code in JS anstatt in JVM Bytecode [Doe13]. Der Compiler wurde als Compiler-PlugIn für den Scala Standardcompiler geschrieben und kann damit auch Eigenschaften wie Compilermakros nutzen.

Mit Scala.js kann die gesamte Sprachkern von Scala sowie einige wenige Teile des Javasprachkerns, die essenziell für Scala sind, genutzt werden. Es gibt leichte Unterscheide, da sich die primitiven Datentypen in Scala und JS unterschiedlich verhalten (siehe Abschnitt 2.2.1 und [Doe14]) und man Java runtime reflection nur sehr eingeschränkt nutzen kann. Insbesondere kann man aber Scala Code in beiden Welten also Bytecode und JS nutzen[webseite].

Mit Hilfe von implicit conversion und custom dynamic types war es möglich das JS Typsystem in das von Scala einzubetten ohne das von Scala zu verändern. Damit ist es möglich bestehende JS Bibliotheken dynamisch oder statisch getypt in Scala.js einzubinden. Was einen großen Vorteil bietet.

Ein großer Nachteil für das Entwickeln von Webseiten ist, das man innerhalb von Scala Code der nach Bytecode übersetzt wird, keinen JS Code mit Hilfe von Scala.js erzeugen kann. Also man kann im erstellten JS Code nicht leicht auf die Serverumgebung, wie die momentane Session oder Datenbanken, zugreifen.

5.2. js-scala

js-scala ist eine Scala Bibliothek um JS Code zu erzeugen [KARO12]. js-Scala benutzt dafür Lightweight Modular Staging (LMS) [RO10]. Dabei wird der in Scala geschriebene Code während der Übersetzung in eine Zwischendarstellung gebracht, die dann zur Laufzeit optimiert und in JS Code (oder auch in Scala Code) übersetzt wird. Das bringt einige Vorteile mit sich.

Es ist möglich zur Laufzeit auf die Umgebung zu reagieren. Man kann entscheiden, ob man den Code in Scala oder JS ausführen möchte. Möchte man zum Beispiel mit JS ein Bild in einem HTML-canvas malen, aber der Browser des Benutzers unterstützt dies nicht, könnte man serverseitig mit Scala ein Bild malen und dies ausliefern. Natürlich kann man auch Daten aus der Laufzeitumgebung in den erzeugten JS-Code einbinden.

Zum anderen konnte gezeigt werden, das aus einer hohen Abstraktionsebene heraus, mit Hilfe der Optimierungen, sehr effizienter JS-Code generiert werden konnte [RFBJ13].

Natürlich werden dies Vorteile durch mehr Aufwand während der Laufzeit erkauft. Das könnte man minimieren in dem man die Ergebnisse zwischenspeichert und/oder die Optimierungen einschränkt.

Wie auch in Scala.js, ist es in js-scala möglich JS-Bibliotheken in statisch oder dynamisch getypt einzubinden.

5.3. SL in Scala

Im Gegensatz zu allen anderen Ansätzen ist die Einbettung von SL in Scala sehr auf die Nutzung in Webservices beschränkt. Das liegt zum einen an der Verwendung von require.js zum Nachladen von Modulen. Dadurch ist es nicht möglich unabhängige JS Dateien zu erstellen. Zum anderen ist das Aufrufen von Scala-Funktionen über Ajax nur in einem solchen Kontext sinnvoll, aber auch besonders hilfreich.

Als Vorteil kann gesehen werden, das die beschreibende Sprache, also SL, sehr übersichtlich und ihre Grenzen klar sind. Weil die anderen Ansätze Scala als beschreibende Sprache benutzen kann dies zu Verwirrung führen.

Ein besonders schwerwiegendes Manko ist die schwierige Einbindung von bereits existierenden JS-Bibliotheken. Dabei kommt eine Eigenschaft von SL besonders zum Tragen. Das Typsystem von SL kann keine JS-Objekte darstellen, die von den meisten Bibliotheken benutzt werden. Es bleibt nur die Möglichkeit Wrapper-Module zu erstellen, die massiv von JS-Quotings Gebrauch machen und um die Verwendung von Objekten herum

arbeiten.

Zu erwähnen bleibt, das die momentane Implementation nur zeigen soll was mit SL in Scala möglich ist. Mögliche Verbesserungsvorschläge werden im Anhang A diskutiert.

5.4. Zusammenfassung

Die wichtigsten Eigenschaften der verschiedenen Ansätze werden in der Tabelle 5.1 noch einmal zusammengefasst. Dabei soll folgendes Szenario angenommen werden. Ein Webserver bietet, eine in Scala geschriebene, Rich Internet Application an. Möglich wäre das Online-Postfach eines E-Mail-Anbieters wie gmail.com.

Tabelle 5.1.: Übersicht über die verschiedenen JS-in-Scala-Projekte

	SL in Scala	Scala.js	js-scala
Optimierung des JS Codes	keine	mit cloureScript	mit Hilfe von LMS
Serveraufwand während einer Anfrage	Wertübersetzung	keinen	erzeugen und optimieren des JS Codes
Nutzen von Servervariablen während der Laufzeit	ja	nein	ja
Abstraktion von Ajax-Anfragen	ja	nein	nein
cross compiling	nein	ja	ja
Einbinden von JS Bibliotheken	schwer	leicht	leicht

Aus meiner Sicht erscheint dabei js-scala besonders viel versprechend. Es vereint die meisten Vorteile und ist besonders flexibel. Mit ähnlichen Techniken wie in dieser Arbeit vorgestellt (siehe Abschnitt 3.2) sollte es möglich sein eine Abstraktion für Ajax-Anfragen zu implementieren.

6. Fazit

A. Future Works

- Security Aspekte beim Aufrufen von Scala Funktionen
- Play PlugIn bauen
- Erzeugen eines JAR's
- Erweiterung von SL um Objekte
- Einfachere Einbindung von JS Bibliotheken. Entweder mehr Module bauen <- viel Wartung oder unterstützung von dynamischen Objekten <- weniger typsicherheit
- Einen besser an Scala angepassten Syntax für SL
- Geht das mit require.js? Optimierung des generierten JS Codes. Am Anfang vllt mit ClojureScript

B. Beschreibung der Tests und Beispielprogramme

C. Benutzte Techniken/Bibliotheken

- Scala
  - Scala v
  - SBT v
  - Play Framework v
  - Macroparadise v
  - json4s v
- JavaScript
  - JQuery v
  - require.js v
  - json.js v
- Simple Language

D. HowTo's

D.1. Projekt aufsetzen

D.2. Einen neuen Translator anlegen

Literaturverzeichnis

[BHL+13] Büchele, Andreas, Christoph Höger, Fabian Linges, Florian Lorenzen, Judith Rohloff und Martin Zuber: *The SL language and compiler*. Sprachbeschreibung, Technische Universität von Berlin, 2013.

[BJLP13] Bisping, Benjamin, Rico Jasper, Sebastian Lohmeier und Friedrich Psiorz: *Projektbericht: Erweiterung von SL um ein Modulsystem*. Projektbericht, Technische Universität von Berlin, 2013.

[Bura] Burmako, Eugene: *Def Macros*. http://docs.scala-lang.org/overviews/macros/overview.html. [Online, zuletzt besucht: 08.07.2014].

[Burb] Burmako, Eugene: *Macro Annotations*. http://docs.scala-lang.org/overviews/macros/annotations.html. [Online, zuletzt besucht: 08.07.2014].

[Bur13] Burmako, Eugene: *Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming*. In: *Proceedings of the 4th Workshop on Scala*, SCALA '13, Seiten 3:1–3:10, New York, NY, USA, 2013. ACM.

[Cro10] Crockford, Douglas: *JSON in JavaScript*. https://github.com/douglascrockford/JSON-js, Nov 2010. [Online, zuletzt besucht: 08.07.2014].

[Doe13] Doeraene, Sébastien: *Scala.js: Type-Directed Interoperability with Dynamically Typed Languages*. Technischer Bericht, 2013.

[Doe14] Doeraene, Sébastien: *Calling JavaScript from Scala.js*. http://www.scala-js.org/doc/calling-javascript.html, 2014.

[Ecm11] Ecma International: *Standard ECMA-262 ECMAScript Language Specification.* Standart, Ecma International, Jun 2011. Edition 5.1.

[HZ13] Höger, Christoph und Martin Zuber: *Towards a Tight Integration of a Functional Web Client Language into Scala.* In: *Proceedings of the 4th Workshop on Scala*, SCALA '13, Seiten 6:1–6:5, New York, NY, USA, 2013. ACM.

[Jso] Json4s: *Json4s One AST to rule them all.* http://json4s.org/. [Online, zuletzt besucht: 08.07.2014].

[KARO12] Kossakowski, Grzegorz, Nada Amin, Tiark Rompf und Martin Odersky: *JavaScript as an Embedded DSL.* In: Noble, James (Herausgeber): *ECOOP 2012 – Object-Oriented Programming*, Band 7313 der Reihe *Lecture Notes in Computer Science*, Seiten 409–434. Springer Berlin Heidelberg, 2012.

[Ode13] Odersky, Martin: *The Scala Language Specification Version 2.9.* Technischer Bericht, Programming Methods Laboratory EPF, Jun 2013.

[Ora11] Oracle America: *Integral Types and Values.* http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.1, Jul 2011. Final Release [Online, zuletzt besucht: 07.07.2014].

[Pag13] Paggen, Marcel: *Klassensystem.* http://www.scalatutorial.de/topic161.html, Feb 2013. [Online, zuletzt besucht: 07.07.2014].

[PH07] Pepper, Peter und Petra Hofstedt: *Funktionale Programmierung: Sprachdesign Und Programmiertechnik (eXamen.Press).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Pla] Play: *The High Velocity Web Framework For Java and Scala.* http://www.playframework.com/. [Online, zuletzt besucht: 08.07.2014].

[Req] RequireJS: *RequireJS.* http://requirejs.org/. [Online, zuletzt besucht: 08.07.2014].

[RFBJ13] Richard-Foy, Julien, Olivier Barais und Jean-Marc Jézéquel: *Efficient High-level Abstractions for Web Programming.* In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, Seiten 53–60, New York, NY, USA, 2013. ACM.

[RO10] Rompf, Tiark und Martin Odersky: *Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs*. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, Seiten 127–136, New York, NY, USA, 2010. ACM.

[The] The jQuery Foundation: *jQuery write less, do more.* https://jquery.com/. [Online, zuletzt besucht: 08.07.2014].

[Unb09] *A Tour of Scala: Unified Types.* http://www.scala-lang.org/old/node/128, Okt 2009. [Online, zuletzt besucht: 07.07.2014].