Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Ernst-Reuter-Platz 7
10587 Berlin

Diplomarbeit

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

Tom Landvoigt, Matrikelnummer: 222115

23. Juli 2014, Berlin

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

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 23. Juli 2014

_____

Unterschrift

Danksagung

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Diplomarbeit unterstützt und motiviert haben.

Besonders möchte ich mich bei meinem Betreuer Martin Zuber für viele nützliche Tipps und Anregungen bedanken.

Daneben gilt mein Dank Hanka und Franzi, die mir bei meiner schrecklichen Rechtschreibung beigestanden haben.

Meinen Eltern möchte ich dafür danken, dass sie mich während des Studiums so herzlich unterstützt und dabei so viel Geduld bewiesen haben.

Zusammenfassung

Im Rahmen dieser Arbeit wurde die Einbettung von SimpleLanguage in Scala verbessert. Es ist jetzt möglich Scala-Werte und -Funktionen in dem eingebetteten SL-Code zu benutzten. Sowohl die dafür nötige Übersetzung von Scala-Typen und -Werten in SL sowie die Implementation der dafür nötigen Scala-Compilermakros wird hier beschrieben. Ein Überblick über verwandte Projekte sowie mögliche zukünftige Arbeiten schließen diese Diplomarbeit ab.

1 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, dass clientseitig Code ausgeführt werden muss. In diesem Bereich hat sich **JS!** (**JS!**) zum Quasi-Standard entwickelt[1]. Dadurch ist man beim Schreiben von browserseitigen 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 **TUB!** (**TUB!**) einen Compiler entwickelt, der die typsichere, funktionale Sprache **SL!** (**SL!**) nach JS übersetzt [**?**]. Andererseits wurde durch die Einführung von compiler makros [**?**] die Metaprogrammierung innerhalb von Scala erheblich vereinfacht.

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

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 eine Scala-Funktion wird eine SL-Funktion erzeugt, die dann die entsprechenden Scala-Funktion aufruft, falls sich die Ein- und Rückgabetypen in SL-Typen übersetzen lassen.

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

Dazu wurde die Möglichkeit geschaffen Scala-Typen 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 **??**. Kapitel wird die Sprache SL kurz vorgestellt. Daraufhin wird beschrieben wie Scala-Typen und -Werte in SL übersetzt werden (Kapitel **??**). Dies wird im **??**. Kapitel benutzt, um mit Hilfe der dort beschriebenen Makros SL in Scala einzubetten. Daraufhin 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 **??**).

2 Einführung in SimpleLanguage

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 modularen Compiler, der es ermöglicht leicht neue Konzepte auszuprobieren. In den folgenden Abschnitten werden die für diese Arbeit relevanten Eigenschaften erklärt.

2.1 Struktur eines SL-Programms

Im Rahmen des Compilerbauprojekts im Sommersemester 2013 wurde SL von den Studierenden um die Möglichkeit der Modularisierung erweitert [**?**]. Seitdem 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.

2.2 Syntax von SL

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

2.2.1 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 2.1: 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.

2.2.2 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-quoting-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.

2.2.3 Funktionsdefinitionen

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 Funktionsdefinitionen der Form `DEF <Funktionsname> = <Funktionsrumpf>`.

2.2.4 Programmeinstiegspunkt

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

2.2.5 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 nächsten Kapitels sein.

3 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 ineinander überführt werden. Dies wird im allgemeinen unter dem Begriff Model Sharing zusammengefasst.

Betrachten wir dazu beispielhaft die Scala-Funktion `scala_foo` im Listing **??**.

Listing 3.1: 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, dass 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 **??** behandelt.

Haben wir einen passenden Typen gefunden, müssen auch die Werte ineinander überführt werden. Dies sollte eine bijektive Abbildung sein. Dass dies nicht immer möglich ist, wird in Abschnitt **??** 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 **??**.

Listing 3.2: 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] Dass dies keine besonders große Einschränkung ist, wird später gezeigt. SL definiert wesentlich weniger Typen in seiner Standardbibliothek als Scala.

3.1 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.

3.1.1 SL-Typsystem

Das Typsystem von SL 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 Typen definiert werden. Dabei handelt es sich um eine Mischung aus Summen- und Tupeltypen [**?**, S.119f u. S. 123]. Ein Typ besteht aus $\mathbb{N}^+$ Konstruktoren mit jeweils $\mathbb{N}$ Attributen.

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 `Option a` wären zum Beispiel `Option Int` oder `Option Void`.

Listing 3.3: Beispiele für selbst definierte Datentypen in SL

```
1 -- Typ mit einem Konstruktor ohne Attribute
2 DATA Void = Void
3
4 -- Typ mit einem Konstruktor mit zwei Attributen
5 DATA CycleKonst = Cycle Int Int
6
7 -- polymorpher Typ mit zwei Konstruktoren
8 DATA Option a = Some a | None
```

3.1.2 Scala-Typsystem

Das Scala-Typsystem in Gänze zu erklären würde den Rahmen dieser Arbeit bei weitem sprengen [**?**]. 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 [**?**]. 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 **??** entnommen werden.

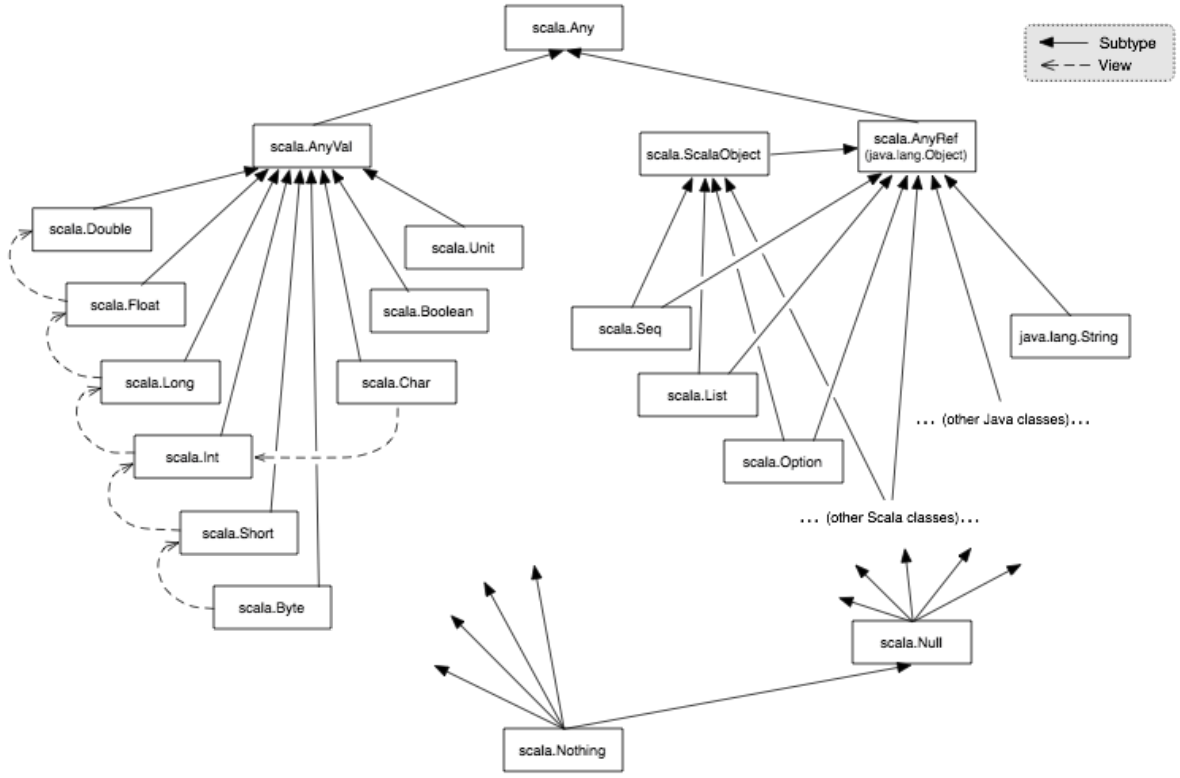

Abbildung 3.1: Vererbungshierarchie einiger Scala-Klassen [**?**]

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

3.1.3 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 3.4: 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 **??** sehen werden, wird die zweite Bedingung für einige primitive 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 3.1: Die Funktion $translate_{type}$

Primitive Datentypen:

Scala-Typ	Byte Short Int Long	Float Double	Char	String	Boolean	Unit
SL-Typ	Int	Real	Char	String	Bool	Void

Andere:

Scala-Typ	Seq[A]	Tuple2[A,B]	Option[A]	Either[A,B]
SL-Typ	List a	Pair a b	Option a	Either a b

Bei generischen Datentypen wie `Seq[A]` folgt aus den oben genannten Bedingungen, dass 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 **??**.

3.1.4 Formalisierung von $translate_{type}$

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

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, $TypeName$ 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:

$$translate_{type}(Model) \hat{=}$$

$$\textbf{DATA}\ t_{Name}(BaseName)\, t_{Param}(Param) = t_{Case}(Sub_1)\ |\ \ldots\ |\ t_{Case}(Sub_n)$$

mit:

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

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

und

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

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. [**?**] inspiriert. Mit Hilfe dieser Schemata können einige der Scala-Typen in SL-Typen übersetzt werden. Wie man im Listing **??** 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 Standardbibliotheken von Scala und SL miteinander 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.

3.2 Darstellungsübersetzung

In diesem Abschnitt wird beschrieben wie Scala-Werte in SL-Werte bzw. zurück übersetzt werden.

Grundsätzlich gelten drei Annahmen:

1. Die Wertübersetzungsfunktionen werden anhand des Scala-Typs gewählt.
2. Werte werden von Scala nach SL und umgekehrt übersetzt.
3. Die Wertübersetzungsfunktionen sind in Scala implementiert. Das heißt wir können die Darstellung der Scala-Werte im JRE ignorieren. Müssen aber die JS-Darstellung der SL-Werte verstehen.

Zunächst wird die Übersetzung von primitiven Werten behandelt. Später werden komplexere Werte übersetzt.

3.2.1 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 **??** werden die Wertebereiche für primitive Typen aufgelistet. Kann ein Wert von einer Darstellungsform nicht

[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 operieren.

[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 3.2: Wertebereich der primitiven Datentypen in Scala und SL [**?**, S. 28-30] [**?**]

SL-Typ	JS-Darstellung und Wertebereich	Scala-Typ und Wertebereich
Int	Number $[-2^{53}+1, 2^{53}-1]$[3]	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: ?)

in die andere Darstellungsform umgewandelt werden, muss dieser Fehler behandelt werden (siehe Abschnitt **??**). Insbesondere bei den ganzzahligen Primitiven ist das Problem unumgänglich. Für ihre Wertebereiche gilt:

$$|\texttt{Scala Int}| < |\texttt{Number bzw. SL Int}| < |\texttt{Scala Long}|$$

3.2.2 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 3.5: Beispiel eines selbst definierten 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 **??** findet sich in der Tabelle **??**.

Mit Hilfe dieser Informationen und dem Schema aus Absatz **??** können jetzt Option-Werte zwischen SL und Scala ausgetauscht werden. In der Tabelle **??** wurden beispielhaft

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

Tabelle 3.3: 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`

einige Werte vom Typ `Option[Int]` übersetzt. In Abschnitt **??** wird auf die Implementation der Übersetzung von `Option[A]` noch einmal genauer eingegangen.

Tabelle 3.4: Übersetzung von `Option` Werten

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

Wie bereits im Abschnitt **??** erwähnt ist es auch möglich ohne das dort beschriebene Schema Werte zu übersetzen. Betrachten wir dies am Beispiel `Seq[A]`. `Seq[A]` ist ein Scala-Trait, der ein Interface zum Arbeiten auf verkettete Listen darstellt. Er kann mit `val a: Seq[Int] = Seq(1,2,3)` instanziiert werden. In den meisten Fällen wird `Seq[A]` intern als `List[A]` dargestellt, muss es aber nicht. Die innere Struktur ist unbekannt, aber man kann mit Hilfe von Methode oder pattern matching auf die Elemente der Liste zugreifen.

Das SL-Äquivalent `List a` hat folgende Typdefinition:

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

Mit pattern matching, rekursiven Funktionen und dem Wissen über die JS-Darstellung von `List a` ist es möglich Werte zu übersetzen. In der Tabelle **??** wurden einige Werte beispielhaft übersetzt.

Die Übersetzung von anonymen Funktionen, also Werten die eine Funktion darstellen, ist im Moment nicht möglich, da kein adäquater Weg gefunden wurde, um ihre Darstellung zwischen Scala und JS auszutauschen.

3.3 Erläuterung 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

Tabelle 3.5: Übersetzung von `Seq[Int]`-Werten

Scala	JS-Darstellung	SL
`Seq()`	`0`	`Nil`
`Seq(1)`	`{ "_cid" => 1, "_var0" => 1, "_var1" => 0 }`	`Cons 1 Nil`

`Seq(3,4)`	`{ "_cid" => 1`
	`, "_var0" => 3`
	`, "_var1" =>`
	`{ "_cid" => 1`
	`, "_var0" => 4`
	`, "_var1" => 0`
	`}`
	`}`
		`Cons 3 (Cons 4 Nil)`

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. Zum Beispiel stellt die Klasse `SeqTranslator` die Verbindung zwischen dem Scala-Typ `Seq[A]` und dem SL-Typ `List a` her.

Die Hauptmethode von `AbstractTranslator` ist `translate` (siehe Listing **??**). 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[6] sowie die **AST!** (**AST!**)-Repräsentation der Wertübersetzungsfunktionen von Scala nach SL und umgekehrt. Andernfalls wird `None` zurückgegeben.

Listing 3.6: Hauptfunktion in AbstractTranslator

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

Weitere Parameter sind `context` und `translators`. `context` ist der Makro Kontext[7]. Er wird benötigt um **AST!**s aufzubauen und den übergebenen Typen zu prüfen. Mit `translators` wird der Teil von $translate_{type}$ übergeben mit denen Spezialisierungen eines

[6] 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.

[7] siehe Kapitel **??**

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 3.7: 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[8].

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

3.3.1 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 **??**). 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 **??**) können wir uns sicher sein, dass übergebene Werte nur vom Typ `Some[A]` oder `None` sind. Wieder mit reflections wird der Typ der Spezialisierung `A` bestimmt. Ist dieser Typ mit 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 (zum Beispiel `Option Int`), der Import-Anweisung (mindestens `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. Grundsätzlich könnte

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

der AST, der die Wertübersetzung repräsentiert, eine anonyme Funktion und kein Funktionsaufruf sein. In späteren Kapiteln heißen die Funktionen zum Übersetzen deshalb immer `scala_to_js` und `js_to_scala`.

```
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` erzeugt.

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

4 Scala Compiler Makros

Im **??**. 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 [**?**]. Um ihre Ergebnisse zu demonstrieren, haben sie eine Beispielwebanwendung mit Hilfe des Play-Frameworks [**?**] geschrieben.

Play ist ein **MVC!** (**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 **??** 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.

4.1 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 **??** behandelt wird. Im darauf folgenden Abschnitt **??** wird beschrieben, wie statischer **SL!** Code mit Hilfe des def macros `slci` eingebunden wird und welche 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 **??**.

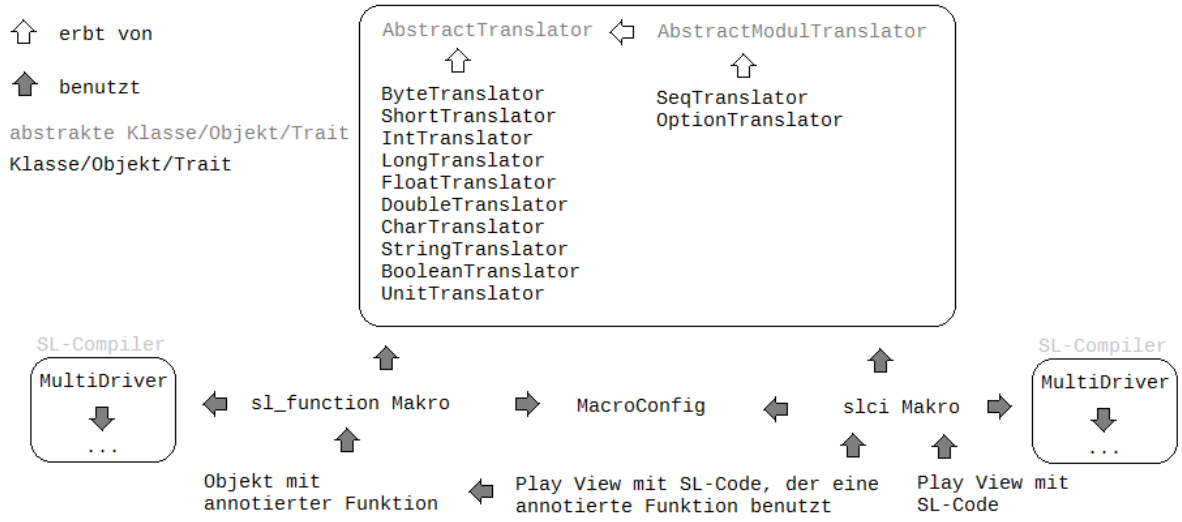

Abbildung 4.1: Projektübersicht

4.1.1 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: `generated_annotation/`.

inline_sl_macro_handler_uri Gibt die **URL!** (**URL!**) an, unter welcher der Server auf asynchrone Kommunikation lauscht um per reflection annotierte Scala-Funktionen aufzurufen. Im Moment ist dies die 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.

4.2 Macro Annotation sl_function

Mit macro annotations kann in den Übersetzungsprozess von Scala eingegriffen werden [?]. 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-Anfragen erleichtern. Das **SL!**-Modul ermöglicht es diesen Aufruf typsicher in **SL!**-Programme einzubinden. Beispielhaft wird das Annotieren einer Funktion anhand der im Listing **??** beschriebenen Funktion `factorial` betrachtet.

Listing 4.1: Scala Beispielfunktion

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

4.2.1 Anforderungen an eine Funktion

Die zu übersetzende Funktion muss gewisse Anforderungen erfüllen. Wenn wir sie im Rahmen von Ajax-Anfragen 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 Ein- und Ausgangstypen müssen sich in SL-Typen übersetzen lassen. Gilt dies nicht, kann keine passende SL-Funktion erzeugt werden.

Die Funktion muss einen Rückgabetyp definieren. Andernfalls müsste der Rückgabetyp erst über reflection bestimmt werden. Der Scala Compiler kann den Rückgabetyp wahrscheinlich bestimmen, dies wurde aber nicht in Angriff genommen.

Die Funktion darf nur eine Parameterliste haben. Mehrere Parameterlisten könnten in eine zusammengefasst werden. Da diese Einschränkung aber nur minimal ist und es den Aufruf der Funktion über reflection erschwert hätte, wurde davon Abstand genommen.

Standardwerte von Parametern werden ignoriert. In der aktuellen Implementation werden die Standardwerte eines Parameters ignoriert. Eine entsprechende Compilerwarnung wird erzeugt.

Der Funktionsname darf keine ungewöhnlichen Zeichen enthalten 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.

4.2.2 SL-Modul

Für jede annotierte Funktion wird ein Modul erstellt. Das heißt eine SL-Quelldatei erstellt, die den entsprechenden SL-Code enthält.

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. Im Fehlerfall wird `None` zurückgegeben. 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 Funkti-

on und ihrer Benutzung in **SL!**-Code aufzulösen. Genauer wird dies im Abschnitt **??** beschrieben. Das Modul wird direkt nach dem erstellen kompiliert.

Zu erwähnen sind noch die Import-Anweisungen. Die Quelldateien `std/option` und `std/_scalafun` werden immer importiert, da sie offensichtlich immer gebraucht werden. Enthält aber die Funktionsdefinition der annotierten Funktion einen Scala-Typ, dessen SL-Pendant in einem externen Modul definiert ist (zum Beispiel `List a`), muss dieses auch importiert werden. Eine entsprechende Import-Anweisung wird dann hinzugefügt.

Listing 4.2: SL-Modul factorial.sl zur Funktion aus Listing **??**

```
1  -- DO NOT ALTER THIS FILE! --------------------------------
2  -- cp: example.Foo
3  -- fn: factorial
4  -- ---------------------------------------------------------
5  -- this file was generated by @sl_function macro ----------
6  -- on 20-06-2014 ------------------------------------------
7  IMPORT EXTERN "std/_scalafun"
8  IMPORT "std/option" AS Opt
9
10 -- this functions should call the scala function:
11 -- example.Foo.factorial
12 PUBLIC FUN factorialSync : Int ->  DOM ( Opt.Option (Int) )
13 DEF factorialSync p0 = {| _sendRequestSync( ... ) ($p0) |}
   : DOM ( Opt.Option (Int) )
14
15 PUBLIC FUN factorialAsync : ( Opt.Option (Int) -> DOM Void )
   -> Int -> DOM Void
16 DEF factorialAsync callbackFun p0 = {| _sendRequestAsync( ... )
   ($callbackFun, $p0) |} : DOM Void
```

4.2.3 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.

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 (siehe Abschnitt **??**).

Listing 4.3: Hilfsfunktion zur Funktion aus Listing **??**

```
1  -- Foo.scala
2  package example
3
4  object Foo {
5    @sl_function def factorial( i: Int ): Long = {...}
6
7    def factorial_sl_helper( p1: org.json4s.JValue ) : org.json4s.JValue = {
8      scala_to_sl(factorial(sl_to_scala(p1)))
9    }
10 }
```

4.2.4 Ablauf eines sl_function Aufrufs

Wie bereits erwähnt wird das Makro während der Übersetzung des Scala Programms aufgerufen. Der Aufruf des `sl_function` folgt den letzten drei Abschnitten. Zunächst wird überprüft, ob die annotierte Funktion den Anforderungen genügt. Dann wird das Modul im Ordner `annotation_sl_macro_folder` (siehe Abschnitt **??**) erzeugt und übersetzt.

Daraufhin wird der **AST!** der Hilfsfunktion erzeugt und diese hinter der annotierten Funktion in das Objekt eingefügt.

Sollte es in einem der Schritte zu einem Fehler kommen, wird ein Compilerfehler erzeugt und die Übersetzung des Scala-Programms abgebrochen.

4.2.5 Aufruf einer annotierten Funktion

Betrachten wir nun den Aufrufprozess einer Funktion im Ganzen am Beispiel der Funktion `factorialSync` aus dem Listing **??**. 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 [**?**] in einen JSON String umgewandelt und mit Funktions- und Objektname als Anfrage an die Adresse des Ajax-Handlers geschickt (siehe Tabelle **??**).

4. Der Ajax-Handler wandelt die Funktionsparameter (`5`) in `JValue` Werte um [**?**] 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 4.1: 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

4.3 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 [**?**]. Der Aufruf solch eines Makros verhält sich wie eine Funktion, nur dass das Makro die **AST!**s der Parameter übergeben bekommt und einen **AST!** liefert, der den Aufruf des Makros ersetzt. Listing **??** enthält einen beispielhaften Aufruf des slci-Makros.

Listing 4.4: 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 ...
```

4.3.1 Statischen SL-Code übersetzen

Mit der Entwicklung eines Modulsystems für **SL!** musste das Einbetten von statischem Code neu geschrieben werden [?]. 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[2] in Webseiten eingebettet werden [?].

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 der Aufruf des slci-Makros durch einen String ersetzt. Der String repräsentiert ein JS-Snippet, das mit Hilfe von require.js das übersetzte Modul einbindet und seine `main`-Funktion aufruft.

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

Tabelle 4.2: Benötigte JS-Bibliotheken

jquery-1.9.0.min.js	Erleichtert Ajax-Anfragen. Wird vom sl_function-Markro benötigt [?].
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 [?].
json.js	zum Umwandeln von JS-Werten in ihre JSON-Repräsentation und zurück. Siehe Abschnitt **??** [?].

4.3.2 Scala Variablen in SL nutzen

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

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 Compiler-Warnungen erzeugt oder die Übersetzung mit einem Fehler abgebrochen.

[2] require.js ist eine JS-Bibliothek mit der JS-Dateien und -Module geladen werden können. Es ist besonders für die Benutzung in Browsern optimiert.

Listing 4.5: Beispielaufruf des slci-Macros mit Scala Variablen

```
slci(
"""
IMPORT "std/option" AS Option
...
FUN foo : Option.Option Int
DEF foo = #s
...
""",
Some(3)
)
```

Daraufhin werden die `IMPORT`-Anweisungen analysiert und die entsprechenden Translator-Klassen geladen[3]. Die von der Makro-API bestimmten Typen[4] 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 **??** würde `#s` durch `{| sl['5a40c735438fd9e1fd43657bd7f8564scalaParam1'] |} : Option.Option Int`[5] ersetzt werden. Der so erzeugte SL-Code wird dann, wie im Abschnitt **??** beschrieben, übersetzt. Listing **??** enthält den vom Makro erzeugt Scala-Code.

Listing 4.6: Erzeugter Scala-Code zum Listing **??**

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

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 **??** findet sich der nach einem Aufruf

[3] 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.

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

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

der Webseite erzeugte **JS!**-Code.

Listing 4.7: JS-Code zum Listing **??**

```
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};
```

4.3.3 Scala Funktionen in SL nutzen

Im Abschnitt **??** 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[6], 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 kompiliert werden. Für die Funktion `factorial` aus Listing **??** würde der Scala-Code im Listing **??** erzeugt werden.

Listing 4.8: Scala `import`-Anweisung für eine annotierte Funktion

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

Die Funktion wird unter einem zufälligen Namen importiert, um Namenskonflikten vorzubeugen.

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

5 SL-Compiler

In diesem Kapitel wird erläutert, wie der SL-Compiler von den hier vorgestellten Compilermakros genutzt wird. Grundsätzlich mussten nur sehr wenige unbedeutende Veränderungen am SL-Compiler gemacht werden. Die Makros benutzen den SL-Compiler, der im Rahmen des Compilerbauprojekts im Sommersemester 2013 an der **TUB!** entwickelt wurde [**?**].

Zunächst wird die Benutzung des SL-Compilers beschrieben. Es folgen die Beschreibungen der Veränderungen, die im Rahmen dieser Diplomarbeit am SL-Compiler gemacht wurden.

5.1 Einbettung des SL-Compilers

Im Grunde sind die im Rahmen dieser Arbeit geschriebenen Compilermakros neue Benutzerschnittstellen für den SL-Compiler. Sie erzeugen SL-Quelldateien, die mit Hilfe des SL-Compilers übersetzt werden. Nur an einer Stelle wurde eine Hilfsfunktionen aus dem SL-Compiler genutzt, um Import-Anweisungen aus dem SL-Code zu extrahieren.

Der aktuelle SL-Compiler wird über die Konsole des **SBT!** (**SBT!**) aufgerufen (Eingaben in eckigen Klammern sind optional) [**?**] [**?**, S. 15f]:

```
1 > run-main de.tuberlin.uebb.sl2.impl.Main [-d <output directory>]
2 [-cp <classpath directory>] -sourcepath <source directory>
3 <module files>
```

Die Eingaben werden analysiert und in eine Konfiguration (`Configs` Trait) umgewandelt. Die Konfiguration enthält folgende Einträge:

sourcepath wo liegen die Quelldateien

sources welche von den Quelldateien sollen übersetzt werden

classpath in welchem Ordner soll nach übersetzten Versionen von importierten SL-Modulen gesucht werden

destination in welchem Ordner soll das Ergebnis der Übersetzung gespeichert werden

Die Konfiguration enthält noch zwei Einträge, die später vom SL-Compiler gesetzt werden. Die so entstandene Konfiguration wird an die Methode `run` aus dem Trait `MultiDriver` [**?**, S. 16-19] übergeben.

```
val result = run( config )
```

`result` enthält die Information, ob die Übersetzung erfolgreich war oder nicht

Die für diese Arbeit geschriebenen Makros benutzen das gleiche Schema. Sie generieren mit Hilfe der im Abschnitt **??** vorgestellten `MacroConfig` eine Instanz des `Configs` Traits. Dabei werden `sourcepath`, `classpath` und `destination` auf `assets_dir` gesetzt. `sources` enthält den zu `assets_dir` relativen Pfad des generierten Moduls. Diese Konfiguration wird dann an `run` übergeben.

Im Rahmen dieser Arbeit war es kaum nötig in den Build-Prozess [**?**, S. 16-19] des SL-Compilers einzugreifen. Die notwendigen Änderungen werden im nächsten Kapitel beschrieben. Insbesondere musste nicht in die Typprüfung eingegriffen werden.

5.2 Erweiterungen an der Konfiguration und 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 [**?**, 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 mit require.js 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 (in `sources` enthalten) eine main-Funktion enthält. Falls dies nicht der Fall ist, wird die Übersetzung mit einem Fehler abgebrochen. Wie im Abschnitt **??** beschrieben, ist nur für das slci-Makro eine main-Funktion nötig. Der Standardwert des Schalters ist `false`.

5.3 Überprüfung des Ergebnistyps von JS-quotings

Um den SL-Compiler besser kennen zu lernen, wurde zu Beginn der Arbeit ein neues Feature für den SL-Compiler umgesetzt.

Wie bereits in der Einführung zu SL (siehe Kapitel **??**) erwähnt, kann mit der JS-quoting-Monade JS-Code direkt in SL benutzt werden. Bis jetzt wurde das Ergebnis solcher quotings zur Laufzeit nicht auf Korrektheit überprüft [**?**, 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 **??**.

Listing 5.1: 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.

6 Ähnliche Projekte

In diesem Kapitel sollen Projekte vorgestellt werden, die JS als Zielsprache haben und JS zu einem statischen Typsystem verhelfen. Einen Überblick über Projekte, die JS als Zielsprache haben, hat das CoffeScript-Projekt zusammengestellt[1].

Für dieses Problem gibt es mehrere Lösungsmöglichkeiten. Es ist möglich einen neue statisch getypte Sprache zu entwickeln, die nach JS übersetzt werden. Ein Beispiel dafür ist das in dieser Arbeit vorgestellte SL. Andere wären TypeScript und JSX[2]. Eine andere Möglichkeit wäre eine bereits existierende statisch getypte Sprache nach JS zu übersetzten, also einen neuen Compiler für eine existierende Sprache zu schreiben. Beispiele dafür wären gopherjs (Go), ghcjs (Haskell), haste (Haskell), fay (Haskell) oder sharpkit (C#). Oder man baut die virtuelle Maschine einer Sprache in JS nach. Wie dies im doppio-Projekt (jvm) gemacht wurde. Ein anderen Ansatz verfolgt das js_of_ocaml Projekt. Hier wird der Ocaml-Bytecode in JS übersetzt. Natürlich kann man auch eine Programmbibliothek schreiben, die JS-Code erzeugt und das Typsystem der Host-Sprache nutzt. Ein Beispiel dafür ist die in Haskell geschriebene sunroof-Bibliothek.

Im Rahmen dieser Arbeit möchte ich mich aber auf Projekte beschränken, die JS in Scala einbetten und diese dann mit der hier vorgestellten Lösung vergleichen. Dazu wird Scala.js und js-scala betrachtet. Auch jscala erzeugt JS-Code innerhalb von Scala, verhilft ihm aber nicht zu einem statischen Typsystem.

Das Szenario soll ein nach dem MVC-Schema geschriebenes Webprojekt sein, das die hier vorgestellten Projekte benutzt, um den clientseitigen JS-Code zu erzeugen. 'Zur Laufzeit' soll in dieser Betrachtung bedeuten: während einer Anfrage an den in Scala geschriebenen Webserver. Es werden folgende Aspekte betrachtet:

Ist es möglich Code client- und serverseitig zu benutzen? Also kann der Code zum generieren von JS auch in Scala genutzt werden?

Können Scala-Werte zur Laufzeit in den JS-Code eingebunden werden? Kann der

[1] siehe https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

[2] siehe webseiten

generierte JS-Code auf die Serverumgebung wie Session oder Datenbanken zugreifen?

Wie gut lassen sich JS-Bibliotheken in den generierten JS-Code einbinden? Welche Mechanismen stellen die einzelnen Projekte dafür zur Verfügung?

Welchen Aufwand erzeugt das generieren des JS-Codes zur Laufzeit?

In den folgenden Abschnitten werden die Ansätze kurz vorgestellt und später verglichen.

6.1 Scala.js

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

Mit Scala.js kann der gesamte Sprachkern von Scala sowie einige wenige Teile des Java-Sprachkerns, die essenziell für Scala sind, genutzt werden. Es gibt leichte Unterscheide, da sich die primitiven Datentypen in Scala und JS unterschiedlich verhalten (siehe Abschnitt **??** und [**?**]) und man Java runtime reflection nur sehr eingeschränkt nutzen kann. Insbesondere kann man aber Scala-Code in beiden Welten, also Bytecode und JS, nutzen[3].

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.

Zur Laufzeit liegen nur die übersetzten Dateien vor. Das bedeutet einerseits, dass der generierte JS-Code nicht auf Scala-Werte zugreifen kann, aber auch das nur ein Mindestmaß an Rechenzeit verloren geht.

6.2 js-scala

js-scala ist eine Scala Bibliothek um JS Code zu erzeugen [**?**]. js-Scala benutzt dafür Lightweight Modular Staging (LMS) [**?**]. Dabei wird der in Scala geschriebene Code während der Übersetzung in eine Zwischendarstellung gebracht, die dann zur Laufzeit

[3] [webseite]

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, dass aus einer hohen Abstraktionsebene heraus, mit Hilfe der Optimierungen, sehr effizienter JS-Code generiert werden konnte [**?**].

Natürlich werden diese Vorteile durch mehr Aufwand während der Laufzeit erkauft. Das könnte man minimieren, indem man die Ergebnisse zwischenspeichert und/oder die Optimierungen einschränkt.

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

6.3 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, dass 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. Zwar ist es möglich, mit Hilfe JS-quoting-Monade, aus Sicht einer funktionalen Sprache, typsicher auf JS zuzugreifen, aber durch das Typsystem von SL ist man trotzdem sehr eingeschränkt. SL kennt keine Abstraktion für JS-Objekte. Für Bibliotheken wie jQuery die JS-Objekte benutzen, müssen dadurch aufwändig wrapper Module gebaut und gewartet werden.

In SL geschriebener Code kann natürlich nicht in Scala wiederverwendet werden, aber es ist möglich zur Laufzeit auf die Serverumgebung zu reagieren. Insbesondere die Einbindung von annotierten Scala-Funktionen über Ajax ist einzigartig. Auch der Aufwand zur Laufzeit hält sich in Grenzen. Nur die Übersetzung der verwendeten Scala-Werte

passiert zur Laufzeit.

Zu erwähnen bleibt, dass die momentane Implementation nur zeigen soll, was mit SL in Scala möglich ist. Mögliche Verbesserungsvorschläge werden im Anhang **??** vorgestellt.

6.4 Zusammenfassung

Die wichtigsten Eigenschaften der verschiedenen Ansätze werden in der Tabelle **??** noch einmal zusammengefasst.

Tabelle 6.1: 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
Code server- und clientseitig nutzen	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 **??**) sollte es möglich sein eine Abstraktion für Ajax-Anfragen zu implementieren.

7 Ausblick

Bis jetzt ist die Einbettung von SL in Scala eher ein Prototyp. Um dies in einen Zustand für den produktiven Einsatz weiter zu entwickeln, sollte noch einige Probleme angegangen werden. Die folgende Liste erhebt keinen Anspruch auf Vollständigkeit.

Security Insbesondere der Aufruf von annotierten Funktionen stellt ein großes Problem dar. Im Moment können alle Hilfsfunktionen über den Ajax-Handler aufgerufen werden (siehe Abschnitt **??**). In Zukunft sollte der Zugriff auf die annotierten Funktionen sinnvoll begrenzt werden.

Bessere Fehlermeldungen Ein Problem sind die immer noch schwer verständlichen Fehlermeldungen des SL-Compilers. Während des Studentenprojekts wurden einige Schritte in die richtige Richtung gemacht. Die Fehlermeldungen bleiben trotzdem meist kryptisch. Insbesondere beim slci-Makro sind auch die Zeilen- und Spaltenangaben der Fehler falsch. Sie beziehen sich auf das generierte Modul und nicht auf die Datei in der das slci-Makro aufgerufen wird.

Typen für Objekte Das SL-Typsystem sollte Objekte unterstützen. Zum einen würde dies die Einbettung in Scala wesentlich vereinfachen. Auf der anderen Seite würde die Einbindung von JS-Bibliotheken vereinfacht werden. Sie setzen häufig auf die Benutzung von JS-Objekten.

Automatische Übersetzung von Scala-Typen Wie im Abschnitt **??** vorgestellt, können bestimmte Scala-Konstrukte automatisch übersetzt werden. Dies sollte implementiert werden und es sollte erforscht werden für welche Konstrukte das noch gilt. Insbesondere würde auch die Einführung von Objekt-Typen in SL dies vereinfachen.

Bessere Einbindung von JS-Bibliotheken Im Moment ist die Einbindung von JS-Bibliotheken relativ schwierig. Für jede Bibliothek muss ein Modul gebaut und gewartet werden. Alle Funktionalität einer Bibliothek muss entsprechend des SL-Typsystems in neue Funktionen gegossen werden. Insbesondere wird dies dadurch erschwert das SL keine Objekte kennt.

Optimierung des erzeugten JS-Codes Bis jetzt erzeugt der SL-Compiler unoptimierten Code. Möglicherweise ist es möglich dieses Problem mit Hilfe des Closer Compilers [] anzugehen.

Testumgebung/-bibliothek für SL Es sollte die Möglichkeit geschaffen werden innerhalb von SBT SL-Bibliotheken zu testen. Ein Problem das mit der Einbettung von SL in Scala gelöst werden sollte, ist die Wartung von großen JS-Bibliotheken. Dazu gehört aber auch diese zu testen.

Play PlugIn Das annotieren von Funktionen und ihr Aufruf über Ajax ergibt nur im Rahmen von Webprojekten Sinn. Play ist ein verbreitetes MVC-Framework für Scala. Es würde die Bedienung erleichtern wenn es ein entsprechendes PlugIn für Play gäbe.

Anpassen des Syntax an Scala Der Syntax von SL orientiert sich eher an Haskell und Opal als an Scala. Um die Bedienung zu erleichtern wäre es sinnvoll den Syntax etwas anzupassen.

8 Fazit

Die Einbettung von SL wurde im Rahmen dieser Arbeit erfolgreich verbessert. Es ist jetzt möglich Scala-Werte und -Funktionen im eingebetteten SL-Code zu benutzen. Insbesondere das Benutzen von Scala-Funktionen über Ajax-Anfragen macht den hier vorgestellten Ansatz einzigartig und erleichtert die Entwicklung von Webprojekten erheblich.

Trotzdem muss noch einiges getan werden um die hier vorgestellte Lösung für den produktiven Einsatz vorzubereiten. Besonders sollte darauf geachtete werden bestehende JS-Bibliotheken bessere einzubinden und bestimmte Scala-Typen auch automatisch zu übersetzen.

A Anleitungen

In diesem Anhang werden einige Anleitungen zur Verfügung gestellt. Der erste Abschnitt beschreibt das Einrichten des Projekts. Also was muss getan werden um die Anwendung lokal zu testen bzw. Tic Tac Toe zu spielen.

Der Abschnitt **??** wird genau beschrieben was getan werden muss um eine neue Translator-Klasse zu schreiben.

A.1 Projekt aufsetzen

Es werden die Programme Git und SBT benötigt. Eine Anleitung wie man Git installiert, findet man hier:
`http://git-scm.com/book/en/Getting-Started-Installing-Git`
Für SBT sollte man der Dokumentation auf
`http://www.scala-sbt.org/`
folgen.

Zunächst müssen ein paar Werte definiert werden:

<root path> Ist der Ordner in des das Projekt liegen soll.
<project path> Ist der Pfad zum Wurzelverzeichnis des Projekts. Das ist `<root path>/sl2-demo`.

Jetzt kann es losgehen. Zunächst wird der Quelltext kopiert.

```
1 cd <root path>
2 git clone https://github.com/illupoed/sl2-demo.git
3 cd sl2-demo
4 git submodule update --init --recursive
```

Als nächstes muss die Konfiguration angepasst werden. Dazu öffnet man in einem Editor die Datei:
`<project path>/sl2/src/main/scala/de/tuberlin/uebb/sl2/slmacro/MacroConfig.scala`
Der Wert von `val assets_dir` muss auf `<project path>/public/sl/` geändert werden.

Wenn man SBT richtig eingerichtet hat, kann man jetzt im Ordner `<project path>` mit dem Befehl `sbt` die SBT-Konsole starten. In der SBT-Konsole kann jetzt mit dem Befehl `run` der Webserver gestartet werden. Jetzt sollte unter der Adresse `http://localhost:9000/koch` das Beispielprogramm aus Abschnitt **??** aufrufbar sein.

Optional kann in der SBT-Konsole mit dem Befehl `eclipse`, aus dem Quelltext ein Eclipse-Projekt generiert werden. Auf der Webseite[1] des sbteclipse Projekts findet man dazu mehr Informationen.

A.2 Neuen Translator anlegen

In diesem Abschnitt wird kurz beschrieben wie eine neue Translator-Klasse angelegt wird. Zunächst muss unterschieden werden ob Ziel der Übersetzung ein SL-Typ sein soll der in einem Modul oder in prelude.sl definiert ist.

Ist der SL-Typ in einem Modul definiert, erbt die neue Klasse von `AbstractModulTranslator`. Andernfalls erbt sie von `AbstractTranslator`. Für die Implementation kann man sich dann an den bereits geschriebenen Translator-Klassen orientieren. Erbt die Klasse von `AbstractModulTranslator` muss darauf geachtet werden das sie einen eindeutigen default `module_alias` hat. Es darf keine zwei Klassen geben, die von `AbstractModulTranslator` erben und den gleichen default `module_alias` haben. Andernfalls könnte das `sl_function`-Makro zwei SL-Bibliotheken unter dem gleichen Namen einbinden, was zu einem Übersetzungsfehler führt.

Ist die neue Klasse programmiert, muss sie den Makros bekannt gemacht werden. Dazu wird eine Klasse die von `AbstractModulTranslator` in die Funktion `AbstractModulTranslator.allModulTranslators` eingetragen. Erbt die Klasse von `AbstractTranslator`, muss sie in die Funktion `AbstractTranslator.preludeTranslators` eingetragen werden.

Daraufhin sollten Unit-Tests und ein check-Programm geschrieben werden. Dies wird in Abschnitt **??** etwas genauer erklärt. Dabei kann man sich an den bereits geschriebenen Tests orientieren.

[1] `https://github.com/typesafehub/sbteclipse`

B Beschreibung der Tests und Beispielprogramme

Im Moment lassen sich Makros noch schwer testen. Insbesondere das Erzeugen von Compiler-Fehlern konnte nicht getestet werden. Deshalb sollte für das Testen der Makros auf die Beispielprogramme zurückgegriffen werden. Vor allem wurde auf das Testen der Translator-Klassen wert gelegt. Im nächsten Abschnitt wird dies genauer beschrieben.

B.1 Test der Translator-Klassen

Das Testen der Translator-Klassen (siehe Abbildung **??**) erfolgt auf zwei Wegen. Zum einen können Unit-Tests aufgerufen werden. Zum anderen wurden Check-Anwendungen geschrieben.

Die Unit-Tests können in der SBT-Konsole des sl2-Subprojekts mit

```
> testOnly de.tuberlin.uebb.sl2.slmacro.TranslationTest
```

aufgerufen werden.

Sie überprüfen ob die Wertübersetzungsfunktionen bijektiv sind und die Wertebereiche der einzelnen Datentypen beachten. Um dies zu erleichtern wurden die Wertübersetzungsfunktionen jeweils im companion object der Translator-Klasse definiert. Die Bijektivität wird mit Tests der Form

```
1 val tmp: Int = 0
2 IntTranslator.jsToScalaInt( IntTranslator.scalaToJsInt( tmp ) ) == tmp
```

und

```
1 val tmp = JInt( 1 )
2 IntTranslator.scalaToJsInt( IntTranslator.jsToScalaInt( tmp ) ) == tmp
```

überprüft.

Das beim Überschreiten des Wertebereichs oder bei unerwarteten Eingaben eine `IllegalArgumentException` erzeugt wird, überprüfen Tests der Art:

```
1 val tmp = JInt( Int.MaxValue.toLong + 1 )
2 // es wird eine IllegalArgumentException erwartet
3 intercept[IllegalArgumentException] {
4   IntTranslator.jsToScalaInt( tmp )
5 }
```

Bis jetzt wurde nur getestet das die Wertübersetzungsfunktionen bijektiv sind, aber nicht ob sie für SL ein sinnvolles Ergebnis liefern. Also das der SL-Wert `Some 2` gleich dem Wert `scala_to_sl(Some(2):Option[Int])` ist. Dazu wurden Testanwendungen geschrieben. Mit Hilfe des Play-HTTP-Servers können diese unter folgenden URL's aufgerufen werden:

http://localhost:9000/checkPrim Hier wird die Übersetzung aller primitiven Scala-Werte überprüft

http://localhost:9000/checkOption Testet den OptionTranslator

http://localhost:9000/checkSeq Testet den SeqTranslator

http://localhost:9000/checkTuple2 Testet den Tuple2Translator

http://localhost:9000/checkEither Testet den EitherTranslator

http://localhost:9000/checkFunCall Sehr einfacher Test ob das Benutzen von annotierten Funktionen funktioniert

B.2 Bespielprogramme

Es wurden zwei Beispielprogramme geschrieben, die die neuen Fähigkeiten der SL-Einbettung verdeutlichen sollen.

B.2.1 Tic Tac Toe

Die Anwendung Tic Tac Toe kann unter der Adresse `http://localhost:9000/tictactoe` aufgerufen werden. Man spielt gegen den Computer das bekannte Spiel. Das Programm soll die Benutzung von annotierten Funktionen testen und verdeutlichen. Dazu wurde die 'KI' als Scala-Funktion geschrieben, die über Ajax aufgerufen wird.

B.2.2 Koch-Kurve

Unter der Adresse `http://localhost:9000/koch` werden Koch-Kurven in einem HTML-canvas gezeichnet. Diese Anwendung wurde bereits von Höger et al. in SL implementiert

[**?**]. Der SL-Code wurde an die aktuelle Version des SL-Compilers angepasst und jetzt kann auf der Webseite ausgewählt werden, wie viel Kurven gezeichnet werden sollen. Damit soll die Benutzung von Scala-Variablen in eingebetteten SL-Code verdeutlicht werden.

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].

[Typ] TypeSave: *sbt*. http://www.scala-sbt.org/. [Online, zuletzt besucht: 18.07.2014].

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