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

8. Juli 2014, Berlin

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

Inhaltsverzeichnis

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 8. Juli 2014

_____

Unterschrift

0.1 Einleitung

Das World Wide Web ist ein integraler Bestandteil unseres Lebens geworden. Ein Großteil der Software mit der wir in Berührung kommen, benutzt Webseiten als Frontend. 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 **JS!** (**JS!**) zum Quasistandart 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.

Im Rahmen eines Projekts [**?**] an der TU-Berlin wurde die typsichere funktionale Sprache **SL!** (**SL!**) entwickelt die nach **JS!** compiliert. Andererseits wurde Mitte 2013 durch die Einführung von compiler makros [**?**] die Metaprogrammierung innerhalb von Scala erheblich vereinfacht.

Im Rahmen eines Papers [**?**] wurde gezeigt, das es möglich ist mit Hilfe von Compilermakros statischen **SL!** Code inline in Scala zu benutzen. Diese Einbettung sollte im Zuge dieser Diplomarbeit erweitert werden. Es ist nun möglich Scala Funktionen und Werte in einem gewissen Rahmen automatisch zu übersetzen und typsicher im **SL!** Code zu benutzen.

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

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

1 Einführung in Simple Language

Mitte 2013 wurde **SL!** als einfache funktionale Lehrsprache für den Studienbetrieb der TU-Berlin entwickelt. Im Rahmen des Compilerbauprojekts im Sommersemerster 2013 wurde **SL!** von den Studierenden um die Möglichkeit der Modularisierung erweitert [**?**]. **SL!** ist eine strikt getypte funktionale Sprache.

Ein **SL!** Programm besteht aus einer Menge von Modulen. Ein Modul ist eine Textdatei 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 JavaScript-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.

Der Syntax soll hier nur Beispielhaft beschrieben werden.

Listing 1.1: Beispielmodul

```
1  -- Kommentar
2
3  IMPORT "std/basicweb" AS Web [1.]
4  IMPORT EXTERN "foo/_bar"
5
6  DATA StringOrOther a = Nothing | StringVal String | OtherVal a [4.]
7
8  PUBLIC FUN getOtherOrElse : StringOrOther a -> a -> a [2.]
9  DEF getString (OtherVal x) y = x
10 DEF getString x y = y
11
12 PUBLIC FUN main : DOM Void [3.]
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
```

1. Mit `IMPORT "<Pfad>" AS <Bezeichner>` können Module nachgeladen werden. Typen und Funktionen die aus Fremdmodulen benutzt werden müssen mit dem `<Bezeichner>` qualifiziert werden. Ein Beispiel dafür ist `Web.alert(...)`.
   Mit `IMPORT EXTERN` können **JS!**-Quelldateien eingebunden werden. In diesem Fall würde der Inhalt der Datei `\_bar.js` im Ordner `foo` an den Anfang des Kompilats kopiert werden.
2. Die optionale Typdefinition 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>`.
3. Ein Spezialfall bildet die Funktion `main`. Sie bildet den Einstiegspunkt in ein **SL!** Programm. Sie hat den festen Typ `DOM Void`. `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.
4. 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 (2) sein.

SL! bietet noch weitere Eigenschaften wie Lambdafunktionen, benutzerdefinierte Operatoren und 'LET IN'-Ausdrücke, diese sind aber nicht für das Verständnis der Diplomarbeit relevant. Bei Interesse kann eine vollständige Beschreibung der Sprache im Report des Compilerbauprojekts [**?**] nachgelesen werden.

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 werden. Betrachten wir dazu die Scala Funktion `foo` im Listing 2.1.

Listing 2.1: Beispielfunktion 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 Funktion `sl_foo` aus Listing 2.2.

Listing 2.2: Übersetzung von foo

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

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

[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.

kurz erläutert und dann die Funktion $translate_{type}$ näher beschrieben.

2.1.1 SL Typsystem

Das Typsystem von **SL!** ist (entsprechend seines Anspruches als Lehrsprache) sehr einfach. Es gibt eine Reihe von vordefinierten Typen `Int`, `Real`, `Char`, `String`, `Bool` und `Void` sowie den Typ der **JS!**-Quoting Monade `DOM a`[2]. Mit dem Stichwort `DATA` können eigene Konstruktor-/Summentypen definiert werden [**?**, S. 123].

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 [**?**]. 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 2.1 entnommen werden.

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). Es wurden aber auch andere vordefinierte Typen wie `Seq[A]` übersetzt, deren innere Struktur sich stark von ihrem SL Äquivalent `List a` unterscheiden.

[2] Typen werden groß geschrieben, Typvariablen klein. `DOM a` steht also zum Beispiel für `DOM Void`, `DOM Int` usw.

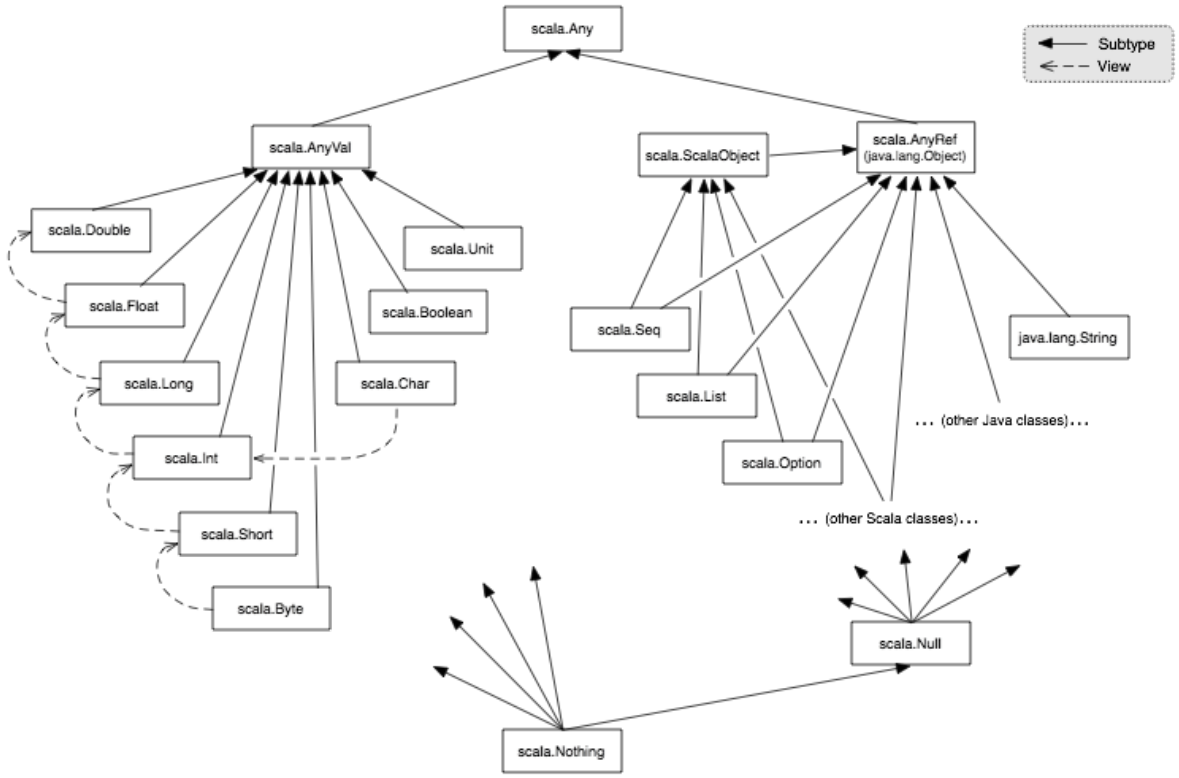

Abbildung 2.1: Vererbungshierarchie einiger Scala Klassen [**?**]

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.

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 die Fließkommaprimitiven mit `Real` und die ganzzahligen Primitiven mit `Int` assoziiert.

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

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] { ... }
```

Tabelle 2.1: Die Funktion $translate_{type}$

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

`Seq[Option[Long]]` würde zu `List Option Int` übersetzt werden. Eine vollständige Auflistung von $translate_{type}$ findet sich in Tabelle 2.1.

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.

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

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: ?)

2.2.1 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).

Bei den ganzzahligen Primitiven fällt auf, das für den Wertebereich gilt:

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

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].

[6]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.

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

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 `_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.

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`

An Hand dieses Schemas können wir nun eine Darstellungsübersetzung für den Scala Typ Option (siehe Listing 2.4) erzeugen:

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`

2.3 Erleuterung der Implementation

Die Paare der Funktion $translate_{type}$ werden durch Klassen, die von `AbstractTranslator` erben, dargestellt. Sie sind nach dem jeweiligen Scala Typen den sie übersetzen benannt[6]. Ihre Hauptfunktion 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 **AST!** (**AST!**)-Repräsentation der Wertübersetzungsfunktionen von Scala nach **SL!** und umgekehrt. Andernfalls wird `None` zurückgegeben.

[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.

Listing 2.6: Hauptfunktion in AbstractTranslator

```
def translate
  ( context: MacroCtxt )
  ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
: Option[( String,
           Set[String],
           context.Expr[Any => JValue],
           context.Expr[JValue => Any] )]
```

Weiter Parameter sind `context` und `translators`. `context` ist der Makro Kontext[8]. 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 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

```
def useTranslators
  ( context: MacroCtxt )
  ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
: Option[( String,
           Set[String],
           context.Expr[Any => JValue],
           context.Expr[JValue => Any] )]
```

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

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.

In Anhang **??** kann eine kommentierte Variante des OptionTranslators eingesehen werden.

[8] siehe Kapitel 3

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

3 Scala Compiler Macros

Wie bereits erwähnt wurde, konnte in einem Paper der Technischen Universität Berlin gezeigt, das man mit Hilfe von Compiler Macros statischen **SL!** Code in die Views von Play-Anwendungen einbetten kann [**?**]. Mit der Erweiterung von **SL!** durch ein Modul-System musste dieses Makro komplett neu geschrieben werden.

Es blieb aber ein grundsätzliches Problem erhalten. Wie kann der generierte **JS!**-Code auf das Serverumfeld wie Datenbanken, Session oder Benutzerdaten zugreifen. In herkömmlichen Anwendungen gibt es zwei Lösungen dafür: Entweder man bindet die Daten direkt in den Quellcode der einzelnen Webseite ein oder lädt sie mit Hilfe von Ajax nach. In der aktuellen Version von **SL!** kann man Scala-Werte direkt im **SL!** Code benutzen und Daten über übersetzte Scala Funktionen nachladen bzw. verändern.

3.1 Struktur des Projekts

Um Scala Funktionen für die Verwendung in **SL!**-Code zu markieren wurden 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 eingebunden wird und welchen Veränderungen gemacht werden mussten um Scala Werte und 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 Hilfsfunktionen aus `AbstractTranslator` und `AbstractModuleTranslator` genutzt.

3.2 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

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

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

Ander Anforderungen ergeben sich aus der Implementation bzw. wurden getroffen um die Implementation zu erleichtern:

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

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.

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

[3] 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.

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.

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

```
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 -- callable_functions.Examples.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
```

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.

Listing 3.3: Hilfsfunktion zur Funktion aus Listing 3.1

```
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 }
```

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!** (**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 3.1).
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 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 [**?**]. Der Aufruf solch eines Makros verhält sich wie eine Funktion, nur das das Makro die **AST!**s 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 [**?**]. 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 [**?**].

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 [**?**].
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 3.2.4 [**?**].

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[4]. Die von der Makro-API bestimmten Typen[5] der Parameter wer-

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

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

den 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['5a40c735438fd9e1fd43657bd7f8564scalaParam1'] |} : Option.Option Int`[6] 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.

Listing 3.6: Erzeugter Scala-Code zum Listing 3.5

```
1 {
2 """
3 require(...);
4 // transformed scala variables
5 sl['5a40c735438fd9e1fd43657bd7f8564scalaParam1'] = %s;
6 """.format( compact( render( scala_to_sl( Some(3) ) ) ) )
7 }
```

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.

Listing 3.7: JS-Code zum Listing 3.5

```
1 require(
2   [ "generated_inline/example.template.scala.48.sl" ],
3   function (tmp) { sl['koch.template.scala.1'] = tmp; }
4 );
5 // transformed scala variables
6 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[7], werden am Anfang des vom Makro erzeugten Scala-Codes `import`-Anweisungen eingefügt, die auf die referenzierten Scala Funktionen verweisen.

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

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

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

```
1 {
2 import example.Foo.{factorial => fun3903232409}
3 """
4 require(...);
5 ...
6 """.format( ... )
7 }
```

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

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` [**?**, 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 [**?**, 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`.

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, braucht das slci-Makro eine main-Funktion. Der Standartwert des Schalters ist `false`.

4.2 Überprüfung des Ergebnistyps von JS!-Quotings

ist für primitive Typen gefixt[1]

[1] siehe Projektbericht S.29

5 Related Works

5.1 SL in Scala

Vor- und Nachteile von SL in Scala beschreiben

5.2 Scala.js

5.3 KA wie die das nennen

6 Fazit

A Anhänge

A.1 Future Works

- Security Aspekte beim Aufrufen von Scala Funktionen
- Play PlugIn bauen
- Erzeugen eines JAR's
- Erweiterung von SL um Objekte
- Analyse des SL-Codes um bessere Typen für Scala Werte zu finden.

A.2 Quellenverzeichnis

A.3 Bilderverzeichnis

A.4 Abkürzungsverzeichnis

SL	Simple Language
JS	JavaScript
AST	Abstract Syntax Tree
URL	Uniform Resource Locator

A.5 Beschreibung der Tests und Beispielprogramme

A.5.1 Die Klasse OptionTranslator als Beispiel

Exemplarisch als Implementation für `AbstractTranslator` wird in diesem Kapitel der `OptionTranslator` genauer betrachtet.

ÜBERARBEITEN (listing kürzen)

Listing A.1: Source Code von OptionTranslator

```
case class OptionTranslator( override val module_alias: String = "Opt" )
extends AbstractModulTranslator( module_alias ) {
  val import_path = "std/option"

  override def translate
    ( context: Context )
    ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
  : Option[( String,
             Set[String],
             context.Expr[Any => JValue],
             context.Expr[JValue => Any] )] =
    {
      import context.universe._

      val option_class_symbol: ClassSymbol = typeOf[Option[_]].typeSymbol.asC
      val first_type_parameter: Type = option_class_symbol.typeParams( 0 ).as
      val option_any_type: Type = typeOf[Option[Any]]

      if ( input.<:<( option_any_type ) ) {
        val actual_type = first_type_parameter.asSeenFrom( input, option_clas

        AbstractTranslator.useTranslators( context )( actual_type, translator
          case Some( ( sl_type, imports, expr_s2j, expr_j2s ) ) =>
            {
              val scala2js = reify(
                {
                  ( i: Any ) => OptionTranslator.scalaToJsOption( i, expr_s2j
                }
              )
              val js2scala = reify( ... )
              Some(
                ( module_alias + ".Option ( " + SL_type + " )",
                imports + module_import,
                scala2js,
                js2scala )
              )
```

```
37              }
38            case None =>
39              None
40          }
41        }
42        else
43          None
44      }
45  }
46
47  object OptionTranslator {
48    def scalaToJsOption( input: Any, f: Any => JValue ): JValue =
49      {
50        import org.json4s._
51
52        input match {
53          case Some( x ) => {
54            val tmp: List[( String, JValue )] = List( "_cid" -> JInt( 0 ), "_va
55            JObject( tmp )
56          }
57          case None => JInt( 1 )
58          case _ =>
59            throw new IllegalArgumentException
60        }
61      }
62
63    def \ac{JS}ToScalaOption[T]( input: JValue, f: JValue => T ): Option[T] =
64      {
65        input match {
66          case JInt( _ ) => None: Option[T]
67          case JObject( x ) => {
68            val tmp = x.find( j => ( j._1 == "_var0" ) )
69            if ( tmp.isDefined )
70              Some( f( tmp.get._2 ) )
71            else
72              throw new IllegalArgumentException
73          }
```

```
74            case _ => throw new IllegalArgumentException
75          }
76        }
77  }
```

`OptionTranslator` erbt nicht von `AbstractTranslator` sondern von `AbstractModuleTranslator`, weil der korrespondierende **SL!** Typ `Option` in einem Modul definiert ist (Zeile 1-2). Außerdem wird ein default `module_alias` angegeben. Dies wird im Kapitel 3.2 relevant werden. In Zeile 3 wird der `import_path` des zu ladenen Moduls angegeben.

Kommen wir zur Hauptfunktion `translate`. Zunächst wird überprüft ob der übergebene Typ `input` ein Subtyp von `Option[Any]`[1] ist (Zeile 21). Falls dies der Fall ist wird die Spezialisierung von `Option` bestimmt (Zeile 22). Also handelt es sich um `Option[Int]` oder `Option[OptionTranslator]` um Beispiele zu nennen. In Zeile 24 wird versucht mit `AbstractTranslator.useTranslators` eine passende **SL!** Entsprechung für die Spezialisierung zu finden. Ist dies der Fall wird ein Ergebnis zusammengesetzt (Zeile 25-45). In jedem anderen Fall wird `None` zurückgegeben.

Die Wertübersetzungsfunktionen von Scala nach **SL!** und umgekehrt werden im companion object `OptionTranslator` definiert um sie besser testen zu können (ab Zeile 55). Sie werfen eine `IllegalArgumentException` falls der Wert auserhalb der übersetzbaren Grenzen liegt[2] oder ein unerwarteter Wert übergeben wird.

A.6 Benutzte Techniken/Bibliotheken

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

[1] Für den Scala Typ `Any` kann es keine semantisch sinnvolle Übersetzung nach **SL!** geben

[2] Das kann bei `Option` nicht passieren, aber bei anderen Übersetzungen. Siehe Tabelle 2.2.

- Simple Language

A.7 HowTo's

A.7.1 Projekt aufsetzen

A.7.2 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].

[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].

[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.

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

[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].