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

23. Juni 2014, Berlin

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

Inhaltsverzeichnis

0.1 Einleitung

[1] [2]

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. Desshalb muss sich jede moderne Programmiersprache daran messen lassen wie leicht es ist mit ihr Webprojekte zu erstellen. Daher bieten Java, Scala, Ruby und viele andere Programmiersprachen Frameworks an um schnell und einfach strukturierte Webprojekte zu erstellen. Ein gemeinsames Problem dieser Frameworks ist es, insbesondere mit dem aufkommen von Rich Internet Applications, das clientseitig Code ausgeführt werden muss. In diesem Bereich hat sich JavaScript (JS) zum 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 Bilbleotheken kann dies die Wartung und Weiterentwicklung erschweren.

Im Rahmen eines Projekts[2] an der TU-Berlin wurde die typsichere funktionale Sprache Simple Language (SL) entwickelt die nach JS compiliert. Andererseits wurde Mitte 2013 durch die Einführung von compiler makros[3] die Metaprogrammierung innerhalb von Scala erheblich vereinfacht.

Im Rahmen eines Papers[4] 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 Kentnisse im Berich funktionaler Programmierung sowie Grundlagen in den Sprachen Scala und JS vorrausgesetzt.

[1] Es gibt weitere Alternativen wie Java oder Flash, die aber Browserplugins vorraussetzen.
[2] siehe Sprachbeschreibung
[3] siehe Let Our Powers Combine
[4] siehe. Paper

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[1]. 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. 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
```

[1] siehe. Projektbericht

```
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 die Datei _bar.js im Ordner foo mit in das Modul eingebunden 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 Modules sichtbar. Darauf folgen eine oder mehrer pattern basierte[2] 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 ganzezahlige Werte und `Real` für Gleitkommazahlen. Der letzte vordefinierte Typ ist `Bool` für boolsche Werte.
4. Mit `DATA <Typname> [<Typprameter> ...] = <Konstruktor> [<Typparameter> ...] | ...` können eigene Typen definiert werden. Wie wir die so möglichen SL Typen und Werte nach Scala und zurück übersetzen wird Stoff des nächsten Kapitels sein.

SL bietet noch weitere Features wie Lambdafunktionen, benutzerdefinierte Operatoren und 'LET IN'-Ausdrücke, diese sind aber nicht für das Verständnis der Diplomarbeit relevant. Bei interesse kann die aktuelle Grammatik und lexikalische Struktur im Report des Compilerbauprojekts [] nachgelesen werden.

[2] siehe Opal

2 Modelsharing

Wenn man Scala Werte in SL Code benutzen möchte müssen diese übersetzt werden. Sowohl der Typ als auch die interne Darstellung. Möchten man zum Beispiel den Scala Wert `1.0` in SL übersetzen, so weist der Scala Compiler diesen mit dem Typ `Float` aus. Die naheliegenste Entsprechnung in SL wäre dazu `Real`. Die Übersetzung der Darstellung wäre in diesem Fall ähnlich naheliegend. Da SL nach JS compiliert würde der Wert im JS Compilat durch `1.0` representiert werden. Die Gegenrichtung, also wenn wir SL Werte in Scala benutzen wollen, funktioniert analog.

Im Zuge der Diplomarbeit reichte es immer anhand des Scala Typs alle benötigten Teile der Übersetzung zu bestimmen:

- passender SL Typ
- Funktion zum Übersetzen eines Scala Wertes in einen SL Wert
- Funktion zum Übersetzen eines SL Wertes in einen Scala Wert

(Schematische Beschreibung der übersetzung von eines wertes und einer Funktion <- erklärt warum wir immer von dem scala typ ausgehen)

Zunächst betrachten wir die Typübersetzung, darauf folgt die Darstellungsübersetzung und schließlich eine Beschreibung der Implementation.

2.1 Typübersetzung

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`[1]. Mit dem Stichwort `DATA` können eigene Konstruktor-/Summentypen definiert werden[2].

Listing 2.1: Beispiele für selbstdefinierte Datentypen in SL

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

[2] siehe Funktionale Programmierung

```
-- Summentyp
DATA Fruits = Apple | Orange | Plum

-- Konstruktortyp
DATA CycleKonst = Cycle Int Int

-- Mischung aus Konstruktor- und Summentyp mit Typvariablen
DATA Either a b = Left a | Right b
```

Im Gegensatz dazu ist das Typsystem von Scala wesentlich komplexer. Scala ist strikt Objektorientiert. Es kennt keine Vordefinierten Typen. Alle Typen sind Objekte, aber es gibt vordefinierte Objekttypen die den primitiven Datentypen von Java zugeordnet werden können (vgl.: http://www.scalatutorial.de/topic161.html#basistypen).

```
Bild: Objekttypen von Scala in ihrer Klassenhierachie [vgl.: http://www.scala

                        scala.Any

scala.AnyVal                            scala.AnyRef

scala.Byte                              java.lang.String
scala.Short                             ...
scala.Int
scala.Long
scala.Float
scala.Double
scala.Char
scala.Boolean
scala.Unit
...
```

Eigene Typen können in Scala mit Vererbung und den Schlüsselworten 'object' und 'class' definiert werden. Für die Methoden der Klassen gibt es in SL kein Äquivalent.

Für die Übersetzung der Typen definieren wir eine Funktion $translate_{type}(Type_{Scala}) = Type_{SL}$. In Abbildung () sehen man diese für die primitiven Datentypen von SL. Diese Zuordnung wurde gewählt, da sie semantisch am Sinnvollsten ist. Die Typen `Float` und `Double` wurden mit `Real` assoziert um die Bedienung zu erleichtern. Analog gilt dies

Tabelle 2.1: $translate_{type}$ für primitive Datentypen

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

für den SL Typ `Int`[3]. Wir kommen aber im Rahmen der Darstellungsübersetzung noch einmal darauf zurück. Für `DOM a` existiert kein sinnvolles Pandant in Scala.

Bei selbstdefinierten Typen muss die Übersetzung händisch passieren. Der SL Typ 'Option a' soll dafür als Beispiel dienen. Neben syntaktischen Anforderungen wie:

- gleiche Anzahl von Typparametern
- alle Werte des Typs x in Scala müssen sich in Werte des Typs `translate_type(x)` in SL darstellen lassen und umgekehrt (siehe nächstes Kapitel)
- ähnliche Unterstruktur (siehe Abblildung Übersetzung von Option)

Ist vor allem die semantische Gleichheit wichtig. Man könnte den SL Typ `String` in Scala durch 'Seq[Char]' darstellen und diese Konstruktion würde die syntaktischen Anforderungen erfüllen, wäre aber wenig sinnvoll da unintuitiv. Vor allem würden in Scala die passenden Funktionen fehlen um mit den übersetzten Werten umzugehen. Für diese Arbeit wurde Beispielhaft 'Option a' wie in der Abbildung [] beschrieben übersetzt.

Listing 2.2: Option in SL und Scala

```
1 Option in SL:
2 PUBLIC DATA Option a =
3           Some a
4         | None
5
6 Option in Scala:
```

[3]Man hätte auch keine Übersetzung für `Byte`, 'Short', `Int` bzw `Float` anbieten können. Der Benutzer müsste dann solche Werte zu 'Long' bzw. 'Double' casten. Weil dies wenig Intuitiv ist, wurde von dieser Lösung abstand genommen.

Tabelle 2.2: Übersetzung von Option[a]

Scala	SL
`Option[a]`	`Option` $translate_{type}$(`a`)
`Some(x:a)`	`Some x :` $translate_{type}$(`a`)
`None`	`None`

```
7  sealed abstract class Option[+A] ... {
8    self =>
9
10   def isEmpty: Boolean
11
12   ...
13 }
14
15 final case class Some[+A](x: A) extends Option[A] {
16   ...
17 }
18
19 case object None extends Option[Nothing] {
20   ...
21 }
```

2.2 Darstellungsübersetzung

Wie bereits in der Einführung dieses Kapitels erwähnt, wählen wir die Wertübersetzungsfunktionen anhand des Scala Typs. Da SL nach JS kompliliert muss ein Scala Wert entsprechend seines Typs in eine passende JS Darstellung übersetzt werden. Für die Gegenrichtung, also SL nach Scala gilt dies analog. Bei allen Übersetzungen haben wir das Problem der unterschiedlichen Grenzen. Man kann zwar jeden Wert des Scala Types 'Byte' in einen Wert des SL Typs `Int` übersetzen, aber nicht umgekehrt. In der Abbildung werden die Grenzen für primitive Typen aufgelistet. Die Übersetzung übernimmt bei den primitiven Datentypen die JSON Bibliothek JSon4s[4].

[4]siehe https://github.com/json4s/json4s

Tabelle 2.3: Umfang der primitven Datentypen in Scala und SL (JS)

SL	JS Darstellung	Scala
Int	Number[5] $[-2^{53}+1, 2^{53}-1]$	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[6](maximale Länge: ?)	String (maximale Länge: ?)

Tabelle 2.4: JS Darstellung des SL Typen People

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

Bei nicht primitiven Werten ist mehr Aufwand nötig. Dazu muss man zuerst verstehen wie die Darstellung von SL Werte für selbstdefinierte Type JS aussieht.

(Beschreibung der übersetzung fehlt)

Listing 2.3: Beispiel eines selbstedefinierten Typs

```
1 DATA People a b c = Alice | Bob a | Cesar b c | Dieter
```

(Zwischensatz) - Hier können vielfältige Probleme bei den Übersetzungen auftreten zb.: Können Seq[Double] beliebiger Länge in List[Real] übersetzt werden

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

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

Tabelle 2.5: Übersetzung von Option Werten

SL	JS Darstellung	Scala
Option a		Option[a]

Some(val)	`{ "_cid" => 0`
	`, "_var0" => val`
	`}`
		`Some(sl_to_scala(val))`

None	`1`	`None`

2.3 Erleuterung der Implementation

Die Übersetzung ist in Klassen organisiert. Eine Klasse erbt von 'AbstractTranslator' und bildet dabei die Verbindung von einem Scala Typen mit einem SL Typen ab. Da sie wir immer vom Scala Typen ausgehen sind sie nach diesen benannt. Die Hauptfunktion ist 'translate'. Ihr wird ein Scala Typ übergeben. Wenn der übergebene Scala Typ der Klasse entspricht erhält man als Rückgabewert den entsprechenden SL Typen, die Import Statements um die entsprechenden SL Module zu laden[7] sowie die Abstract Syntax Tree (AST)-Representation der Wertübersetzungsfunktionen von Scala nach SL und umgekehrt. Andernfalls wird `None` zurückgegeben.

Listing 2.4: 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] )]
```

Weiter Parameter sind 'context' und 'translators'. 'context' ist der Compiler Macro Kontext. Mit 'translators' werden alle Translatorklassen übergeben mit denen Spezialisierungen übersetzt werden können.

Möchte man einen Scala Typ nicht nur gegen eine Klasse prüfen kann man die Hilfsfunktionen 'useTranslators', 'useTranslatorSLToScala' oder 'useTranslatorScalaToSL' aus dem companion object von 'AbstractTranslator' nutzen.

Listing 2.5: Hilfsfunktionen

[7] Bei primitven 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 werde

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

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

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

'translators' gibt hier an welchen Teil der Funktion $translate_{type}$ man nutzen möchte.

2.3.1 Die Klasse OptionTranslator als Beispiel

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

ÜBERARBEITEN (listing kürzen)

Listing 2.6: Source Code von OptionTranslator

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

  override def rename( module_alias: String ) = copy( module_alias );

  override def translate
```

```
8       ( context: Context )
9       ( input: context.universe.Type, translators: Seq[AbstractTranslator] )
10    : Option[( String,
11               Set[String],
12               context.Expr[Any => JValue],
13               context.Expr[JValue => Any] )] =
14      {
15        import context.universe._
16
17        val option_class_symbol: ClassSymbol = typeOf[Option[_]].typeSymbol.asC
18        val first_type_parameter: Type = option_class_symbol.typeParams( 0 ).as
19        val option_any_type: Type = typeOf[Option[Any]]
20
21        if ( input.<:<( option_any_type ) ) {
22          val actual_type = first_type_parameter.asSeenFrom( input, option_clas
23
24          AbstractTranslator.useTranslators( context )( actual_type, translator
25            case Some( ( \ac{SL}_type, imports, expr_s2j, expr_j2s ) ) =>
26              {
27                val scala2js = reify(
28                  {
29                    ( i: Any ) => de.tuberlin.uebb.sl2.slmacro.variabletranslat
30                    OptionTranslator.scalaToJsOption( i, expr_s2j.splice )
31                  }
32                )
33                val \ac{JS}2scala = reify(
34                  {
35                    ( i: JValue ) => de.tuberlin.uebb.sl2.slmacro.variabletrans
36                    OptionTranslator.jsToScalaOption( i, expr_j2s.splice )
37                  }
38                )
39                Some(
40                  ( module_alias + ".Option ( " + \ac{SL}_type + " )",
41                  imports + module_import,
42                  scala2js,
43                  \ac{JS}2scala )
44                )
```

```
45              }
46            case None =>
47              None
48          }
49        }
50        else
51          None
52      }
53  }
54
55  object OptionTranslator {
56    def scalaToJsOption( input: Any, f: Any => JValue ): JValue =
57      {
58        import org.json4s._
59
60        input match {
61          case Some( x ) => {
62            val tmp: List[( String, JValue )] = List( "_cid" -> JInt( 0 ), "_va
63            JObject( tmp )
64          }
65          case None => JInt( 1 )
66          case _ =>
67            throw new IllegalArgumentException
68        }
69      }
70
71    def \ac{JS}ToScalaOption[T]( input: JValue, f: JValue => T ): Option[T] =
72      {
73        input match {
74          case JInt( _ ) => None: Option[T]
75          case JObject( x ) => {
76            val tmp = x.find( j => ( j._1 == "_var0" ) )
77            if ( tmp.isDefined )
78              Some( f( tmp.get._2 ) )
79            else
80              throw new IllegalArgumentException
81          }
```

```
82            case _ => throw new IllegalArgumentException
83          }
84        }
85    }
```

`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]`[8] 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[9] oder ein unerwarteter Wert übergeben wird.

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

[9] Das kann bei `Option` nicht passieren, aber bei anderen Übersetzungen. Siehe Tabelle 2.3.

3 Scala Compiler Macros

Wie bereits erwähnt wurde, konnte in einem Paper[1] 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 Macro 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 läd 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 Macros 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[2] Code zu verändern. Mit dem geschriebenen Macro kön-

[1] siehe Paper

[2] Es können auch Funktionen, Klassen, Objekte, Typparameter oder Funktionsparameter annotiert werden. siehe http://docs.scala-lang.org/overviews/macros/annotations.html

nen 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[3]
- Die Ein- und Ausgangstypen müssen sich in SL-Typen übersetzen lassen.
- Der Funktionsname darf keine ungewöhnlichen Zeichen enthalten[4]

3.2.2 SL-Modul

Für jede annotierte Funktion wird ein Modul erstellt. Das Modul enthält zwei Funktionen. Jeweils für den asynchronen und synchronen Aufruf der Scala-Funktion über Ajax.

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

[4] Da sich der Name der Funktion im Name und Pfad des erzeugten Moduls wiederspiegelt 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 Ergebniss 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-Bibleothek 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 Ergebniss 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 Aufrufsprozess 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 Uniform Resource Locator (URL) des Ajax-Handlers, der voll qualifizierte Name des Objects und der Funktionsname übergeben. In einem zweiten Schritt wird der eigentliche Parameter (SL-Codiert) übergeben.
3. Die SL-Parameter werden in einen JSON String umgewandelt[5] und mit Funktions- und Objektname als Anfrage an die Addresse des Ajax-Handlers geschickt (siehe Tabelle 3.1).
4. Der Ajax-Handler wandelt die Funktionsparameter (`5`) in `JValue` Werte um[6] und ruft dann über reflection die Hilfsfunktion `factorial_sl_helper` auf. Das Ergebins (`120`) des Aufrufs wird zurück an den Client gesendet.
5. Ist die Anfrage an den Server erfolgreich wird `Some(120)` zurückgegeben, andernfalls `None`.

3.3 Def Macro slci

Bis jetzt kann man nur Funktionen markieren um sie in SL-Code zu benutzen. Nun soll SL benutzt werden um JS-Code zu generieren und ihn auf Benutzerseite zu verwenden.

[5]Für die Umwandlung wird die Bibleothek json.js benutzt (http://www.json.org/js.html).
[6]Die Umwandlung geschieht mit der Bibliothek json4s (https://github.com/json4s/json4s).

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

Dazu wurde das `slci` Makro neu geschrieben und erweitert. Im Laufe der nächsten Abschnitter vollziehen wir die Entwicklung des Macros nach.

Mit def macros[7] kann auch wärend des Übersetzungsprozesses von Scala in den Code eingegriffen werden. Der Aufruf solch eines Macros verhält sich wie eine Funktion, nur das das Makro die ASTs der Parameter übergeben bekommt und einen AST liefert der den Aufruf des Makros ersetzt.

3.3.1 Statischen SL Code übersetzen

Mit der Entwicklung eines Modulsystems[8] 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[9] in Webseiten eingebettet werden[10].

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. Andernfalls wird ein Compilerfehler erzeugt.

3.3.2 Scala Variablen in SL nutzen

3.3.3 Scala Funktionen in SL nutzen

[7] siehe http://docs.scala-lang.org/overviews/macros/overview.html
[8] siehe Projektbericht
[9] siehe Projektbericht require.js
[10] Der erzeugte JS Code kann auch mit node.js (siehe node.js) ausgeführt werden.

4 Erweiterungen am SL-Compiler

5 Related Works

5.1 Scala.js

5.2

6 Zusammenfassung

7 Anhänge

7.1 Future Works

7.2 Quellenverzeichnis

7.3 Bilderverzeichnis

7.4 Abkürzungsverzeichnis

SL Simple Language

JS JavaScript

AST Abstract Syntax Tree

URL Uniform Resource Locator

7.5 Beschreibung der Tests und Beispielprogramme

7.6 Benutzte Techniken/Bibliotheken

- Scala
  - Scala v
  - SBT v
  - Play Framework v
  - Macroparadise v
  - json4s v
- JavaScript
  - JQuery v

  - require.js v
  - json.js v
- Simple Language

7.7 HowTo's

7.7.1 Projekt aufsetzen

7.7.2 Einen neuen Translator anlegen

Literaturverzeichnis

[1] Eugene Burmako. 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, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[2] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. Javascript as an embedded dsl. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434. Springer Berlin Heidelberg, 2012.