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

Diplomarbeit

Titel

Tom Landvoigt, Matrikelnummer: 222115

20. Juni 2014, Berlin

betreut durch Prof. Dr. Peter Pepper

Inhaltsverzeichnis

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. Desshalb muss sich jede moderne Programmiersprache daran messen lassen wie leicht es ist mit ihr Webprojekte zu erstellen. Daher bieten Java, Scala, Ruby, PHP 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 man auch 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.

Im Rahmen eines Projekts an der TU-Berlin wurde die Sprache Simple Language (SL) entwickelt die nach JS compiliert. Das bot die Möglichkeit mit Hilfe von Compilermakros eine typsichere funktionale Abstraktion für JS in Scala bzw. seinem Webframework Play einzubetten.

Im Rahmen eines Papers[2] 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. 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
5  DATA StringOrOther a = Nothing | StringVal String | OtherVal a (4)
6
7  PUBLIC FUN getOtherOrElse : StringOrOther a -> a -> a (2)
8  DEF getString (OtherVal x) y = x
9  DEF getString x y = y
10
11 PUBLIC FUN main : DOM Void (3)
12 DEF main = Web.alert(intToString (getOtherOrElse(exampleVar, 3)))
13
14 FUN exampleVar : StringOrOther Int
15 DEF exampleVar = OtherVal 5
16
17 FUN getDocumentHight : DOM Int
18 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(...)`.

[1] siehe. Projektbericht

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 Model Sharing

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.scalatutorial.de/topic161.html\

                      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 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

[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 2.1: $translate_{type}$ 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

- 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:
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 }
```

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`

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: ?)

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].

Bei nicht primitiven Werten ist mehr Aufwand nötig. Dazu muss man zuerst verstehen

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

[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 2.4: 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`

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`

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

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
```

[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

```
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

```
1  def useTranslators
2    ( c: MacroCtxt )
3    ( input: c.universe.Type, translators: Seq[AbstractTranslator] )
4  : Option[( String,
5             Set[String],
6             c.Expr[Any => JValue],
7             c.Expr[JValue => Any] )]
8
9  def useTranslatorSLToScala
10   ( c: MacroCtxt )
11   ( input: c.universe.Type, translators: Seq[AbstractTranslator] )
12 : Option[( String,
13            Set[String],
14            c.Expr[JValue => Any] )]
15
16 def useTranslatorScalaToSL
17   ( c: MacroCtxt )
18   ( input: c.universe.Type, translators: Seq[AbstractTranslator] )
19 : Option[( String,
20            Set[String],
21            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.

Listing 2.6: Source Code von OptionTranslator

```
1  case class OptionTranslator( override val module_alias: String = "Opt" )
2  extends AbstractModulTranslator( module_alias ) {
```

```
3     val import_path = "std/option"
4
5     override def rename( module_alias: String ) = copy( module_alias );
6
7     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.asClass
18        val first_type_parameter: Type = option_class_symbol.typeParams( 0 ).asType.toType
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_class_symbol )
23
24          AbstractTranslator.useTranslators( context )( actual_type, translators ) match {
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.variabletranslation.std.
30                    OptionTranslator.scalaToJsOption( i, expr_s2j.splice )
31                  }
32                )
33                val \ac{JS}2scala = reify(
34                  {
35                    ( i: JValue ) => de.tuberlin.uebb.sl2.slmacro.variabletranslation.std.
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 ), "_var0" -> f( x ) )
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

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

liegt[9] oder ein unerwarteter Wert übergeben wird.

[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 das Hypertext Markup Language (HTML) der Webseite ein oder läd sie mit Hilfe von Ajax nach. In der aktuellen Version 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 Kapitel 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[2] können in Scala Funktionsdefinitionen[3] annotiert werden, um ihren Aufbau während der Übersetzung zu verändern. Im Rahmen dieser Diplomarbeit wurde dies genutzt um

[1] siehe Paper

[2] siehe http://docs.scala-lang.org/overviews/macros/annotations.html

[3] Es können auch Klassen, Objekte, Typparameter oder Funktionsparameter annotiert werden.

Listing 3.1: Scala Beispielfunktion

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

ENDE DER BEARBEITUNG

3.2.1 Anforderungen an eine Funktion

3.2.2 SL-Modul

3.2.3 Wrapper-Funktion

ÜBERARBEITEN

Mit Hilfe des `sl_function`-Macros können wir Funktionen markieren, die für die Verwendung in SL-Code zur Verfügung stehen sollen. Für eine markierte Funktion wird ein SL-Modul erzeugt, das dann eingebunden werden kann. Wird die Funktion in SL aufgerufen wird Ajax-Anfrage mit den entsprechenden Parametern an den Server gesenden, die Scala Funktion mit Hilfe von Reflections[4] aufgerufen und das Ergebnis als Antwort verpackt. Im Listing 3.1 sieht man beispielhaft die Benutzung des Macros und die Signaturen der generierten SL-Funktionen. Der Aufruf kann synchron oder asynchron erfolgen. Um Fehler in der Verbindung behandeln zu können, wird das Ergebnis in `Option` verpackt.

Damit eine Funktion für SL übersetzt werden kann, muss sie gewisse Vorraussetzungen erfüllen:

- sie muss statisch aufrufbar sein, also:
  * muss in einem Objekt definiert sein
  * die Signatur darf keine Typparameter enthalten
  * die Funktion darf nicht als `private` oder `protected` markiert sein
- sie muss einen Rückgabetyp definieren
- sie darf nur eine Parameterliste haben[5]

[4] siehe http://docs.scala-lang.org/overviews/reflection/overview.html

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

- der Funktionsname darf keine ungewöhnlichen Zeichen enthalten[6]

Falls diese Vorraussetzungen erfüllt sind, werden mit Hilfe der Translator-Klassen SL-Entsprechungen für die Eingangs- und Ausgangstypen gesucht. Im obigen Beispiel (Listing 3.1) wird der Typ des Parameters `i` nach `Int` übersetzt. Der Rückgabetyp wird nach `Opt.Option (Int)` übersetzt.

3.2.4 Aufruf von Scala Funktionen

3.3 Inline Macro

[6]Da sich der Name der Funktion im Name 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.

4 Erweiterungen an der DOM Monade

5 Related Works

6 Zusammenfassung

7 Anhänge

7.1 Future Works

7.2 Quellenverzeichnis

Literaturverzeichnis

[Bowie87] J. U. Bowie, R. Lüthy and D. Eisenberg. *A Method to Identify Protein Sequences That Fold into a Known Three-Dimensional Structure*. Science, 1991 (253), pp 164-170

7.3 Bilderverzeichnis

7.4 Abkürzungsverzeichnis

SL Simple Language

JS JavaScript

AST Abstract Syntax Tree

HTML Hypertext Markup Language

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 Projekt aufsetzen