

Projektbericht: Erweiterung von SL um ein Modulsystem

Benjamin Bisping, Rico Jasper, Sebastian Lohmeier, Friedrich Moritz Psiorz

Compilerbauprojekt SoSe 2013, Technische Universität Berlin

1 Einleitung

TODO (* Hällü Wörlö *)

2 Überblick

TODO (* Alle/Ben: Kurz die neuen Features bzw. den Ausgangspunkt beschreiben *)

3 Syntax und Parser

TODO (* Fritz: Syntaxanpassungen, Schwierigkeiten, Designentscheidungen *)

3.1 Qualifizierte Bezeichner

3.2 Grammatik

4 Semantische Analyse

TODO (* Rico: Methode, Schwierigkeiten, Designentscheidungen *)

4.1 Auflösung von Importen

4.2 Type-Checking

5 Codegenerierung

TODO (* Sebastian: Beispiele, Schwierigkeiten, Designentscheidungen *)

TODO (* Aufruf des Compilers, aus binary und in sbt, durchgehendes Beispiel *)

Die Ausführung des generierten JavaScript-Codes wird in node.js¹, Firefox², Chrome³ und Internet Explorer⁴ unterstützt.

Bei Aufruf des Compilers mit

```
$> <PROGRAMM-NAME> -d <outputDirectory> -cp <classpathDirectory>  
[ -sourcepath <sourceDirectory> ] <moduleFile>
```

werden aus dem <classpathDirectory> die Signatur-Dateien bereits kompilierter Module geladen, sowie das angegebene <moduleFile> sowie alle von diesem transitiv verwendeten Module, die noch nicht kompiliert im <outputDirectory> vorhanden sind bzw. deren **TODO (* keine Metonymie! *)**

Modifikationsdatum im <outputDirectory> vor dem Modifikationsdatum der SL-Moduldatei im <sourceDirectory> liegt, kompiliert. Dabei werden Signaturen (siehe Abschnitt 5.1), sowie JavaScript-Dateien (siehe Abschnitt 5.3) für alle kompilierten Module erstellt, wobei require.js (siehe Abschnitt 5.2) verwendet wird, um die JavaScript-Dateien der Module zu laden. Sofern das beim Aufruf des Compilers angegebene <moduleFile> eine Funktion namens main deklariert, werden für dieses noch eine main.js-Datei und eine index.html-Datei erstellt, die den Aufruf der main-Funktion in node.js und im Browser erlauben (siehe Abschnitt 5.3).

TODO (* Ausführung des Codes in (unterstütztem) Browser und node.js, Voraussetzungen dafür *)

5.1 Signaturen

TODO (* Das ist wohl eher für Rico... *)

5.2 require.js

Um die Module zur Laufzeit in JavaScript zu laden, wurde require.js⁵ statt CommonJS⁶ ausgewählt, da es im Gegensatz zum Modulsystem von node.js auch im Browser verfügbar ist, jedoch auch in node.js genutzt werden kann **TODO (* naja, das stimmt noch nicht ganz *)**

¹ <http://nodejs.org/> - getestet mit Version 0.10.10 (**TODO (* aktualisieren auf 0.10.13 *)**)

² **TODO (* URL, zum Testen benutzte Version und OS *)**

³ **TODO (* URL, zum Testen benutzte Version und OS *)**

⁴ **TODO (* URL, zum Testen benutzte Version und OS *)**

⁵ <http://requirejs.org/> v. 2.1.6 **TODO (* updaten auf 2.1.8 *)**

⁶ <http://www.commonjs.org/>

In node.js stehen zwei Wege zur Verfügung, um Abhängigkeiten zwischen Modulen zu deklarieren und zur Laufzeit aufzulösen, siehe Listings ?? und ??.
TODO (* AMD besprechen? *)

Die Moduldefinition mit einem Array von Abhängigkeiten (siehe Beispiel im Listing ??) erlaubt den Zugriff auf verwendete Module, können jedoch keine zirkulären statischen Abhängigkeiten auflösen, da für die Erstellung der gegenseitig abhängigen Module jeweils das andere Modul-Objekt als Parameter bei Erstellung des Moduls übergeben werden muss. Dieses Problem wird in require.js mittels Export-Objekten gelöst, die beim Erstellen eines Moduls übergeben und zur Laufzeit verwendet werden (siehe Beispiel im Listing ??). Die Moduldefinition mit Export-Objekten wurde in SL2 gewählt, um später statische zirkuläre Abhängigkeiten auflösen zu können, auch wenn die bisherige Implementierung des Compilers dies nicht erlaubt.

```
define(["modules/B"], function(b) {
    return {
        "a" : function() { return "A.a"; },
        "b" : function() { return b.b(); }
    };
});

define(function(require, exports, module) {
    var b = require("modules/B");
    exports.a = function() { return "A.a"; };
    exports.b = function() { return b.b(); };
});
```

TODO (* Kompilierung der Module *)

TODO (* Kompilierung der main-Funktion *)

TODO (* Designentscheidung für require.js-Verwendung, die theoretisch auch statisch zirkuläre Abhängigkeiten auflösen kann *)

TODO (* require.js wird mitgeliefert, sodass es für Ausführung im Browser nicht installiert werden muss *)

TODO (* Installation von require.js in node.js – im lokalen Verzeichnis oder global? in Systemvoraussetzungen für SL2 beschreiben *)

5.3 Build-Prozess

TODO (* implizit unqualifiziert importiertes prelude aus dem resources-Verzeichnis der SL2-Distribution, Zugriffe darauf werden nach dem Typcheck qualifiziert mit /lib/prelude – bzw. mit /lib/prelude *)

TODO (* Übersetzung der / (oder aller nicht-zugelassenen Zeichen) zu \$ in JavaScript? *)

TODO (* Ort, an dem die Templates, prelude, und require.js (im Distributable) gespeichert sind *)

5.4 Externe Definitionen

TODO (* Das ist wohl eher für Ben... *)

6 Prelude und Bibliotheken

Einerseits zur Erweiterung des ursprünglichen Funktionsumfangs, andererseits vor allem zum Testen des neuen Modulsystems, haben wir eine Reihe grundlegender Bibliotheken für SL entwickelt. Im Folgenden wollen wir Ausschnitte aus den Bibliothekssignaturen vorstellen, ihre Funktionen angerissen und Besonderheiten bei ihrer Verwendung des Modulsystems und neuer Sprachfeatures ansprechen. Die vollständigen Module inklusive Implementierung finden sich in `/src/main/resources/lib/`.

6.1 Prelude und Libraries

Fast alle vormalig fest in den Compiler eingebauten Funktionen und Konstruktoren werden jetzt durch ein eigenes, umfangreicheres Prelude-Modul definiert. Dieses wird implizit durch jedes SL-Programm unqualifiziert importiert.

Im Prelude werden unter anderem alle Basistypen deklariert. Zugleich sind diese allerdings noch in den Compiler integriert, damit die Literale einen Typ erhalten können unabhängig vom Prelude-Import. Die meisten dieser Datentypen kommen ohne Konstruktorendefinition daher, sind deshalb aber noch lange nicht leer, was wir durch `DATA EXTERN` anzeigen.

```
DATA EXTERN Int
DATA EXTERN Real
DATA EXTERN Char
DATA EXTERN String
```

```
PUBLIC DATA Void = Void
DATA EXTERN DOM a
```

Stärker als andere Module bildet das Prelude Funktionen auf handgeschriebenen JavaScript-Code ab. Diese Abbildung wurde bisher durch eine hardcodierte Umwandlung im SL-Compiler realisiert. Dank `IMPORT EXTERN` und `DEF EXTERN` **VERWEIS EINBAUEN** kann das Prelude selbst spezifizieren, dass `+` auf das JavaScript-Objekt `_add` aus `_prelude.js` abgebildet werden soll.

```
IMPORT EXTERN "_prelude"
[...]
PUBLIC FUN + : Int -> Int -> Int
DEF EXTERN + = {| _add |}
```

So sind weite Teile der Preludes umgesetzt. Andere grundlegende Aspekte sind hingegen völlig in SL definiert, zum Beispiel der Datentyp `BOOL`.

```
PUBLIC DATA Bool = True | False
```

```
PUBLIC FUN not : Bool -> Bool
DEF not True = False
DEF not False = True
```

Es sind auch einige neue Funktionen hinzugekommen, zum Beispiel # für Funktionskomposition⁷ und id als Identitätsfunktion.

```
PUBLIC FUN # : (b -> c) -> (a -> b) -> (a -> c)
DEF f # g = \ x . f (g x)
```

```
PUBLIC FUN id : a -> a
DEF id a = a
```

Eine spannende neue Funktion im Prelude ist **error**. Diese hat einen beliebigen Rückgabety, kann also an beliebigen Stellen in den Code geschrieben werden. Allerdings wird **error** niemals einen Wert zurückgeben, sondern schlicht das Programm mit einer Fehlermeldung enden lassen.⁸ Man kann sich das **error** auch als eine Möglichkeit vorstellen, in der Abwesenheit von Subtyping, eine Art Bottom-Type einzuführen. Vor allem ist es aber praktisch: Häufig möchte man im Implementierungsprozess schon teile Testen, aber noch nicht überall sinnvollen Code eintragen. Manchmal lässt sich für einen Fall auch einfach kein sinnvolles Programmverhalten angeben.

```
-- The representation of the undefined.
PUBLIC FUN error : String -> a
DEF EXTERN error = {| function(msg){throw msg} |}
```

6.2 List, Option, Either

Unsere mitgelieferten Module enthalten die klassischen algebraischen, generischen Datentypen **List** (aka Sequence), **Option** (aka Maybe), **Either** (aka Union) und **Pair** (aka Product2). Bis auf **List.fromString** sind diese Module komplett in SL geschrieben ohne Rückgriff auf JavaScript. Wir haben auch ein paar der grundlegenden Funktionen wie **map** und **reduce** implementiert. Vorrangig ging es uns aber darum, komplexere importierte Konstruktoren beim Pattern Matching anhand dieser Typen auszuprobieren.

```
PUBLIC DATA List a      = Nil | Cons a (List a)
PUBLIC DATA Option a   = None | Some a
PUBLIC DATA Either a b = Left a | Right b
PUBLIC DATA Pair a b   = Pair a b
```

⁷ Das ungewöhnliche Zeichen rührt daher, dass „o“ in SL kein Operator sein kann und „.“ für die Lambda-Abstraktion und Namensqualifizierung reserviert ist.

⁸ Diese Funktion ist also keine echte, wohldefinierte Funktion, sondern hat dasselbe „Ergebnis“ wie eine Endlosrekursion.

6.3 Reeale Zahlen — `real.sl`

Am Anfang des Projekts hatten wir reele Zahlen in SL integriert. Diese und noch mehr Funktionen auf Reals werden jetzt in `real.sl` definiert durch Abbildung auf entsprechende Funktionen auf JavaScripts `num`. Bei der ursprünglichen Umsetzung erwies sich als ausgesprochen unhandlich, dass die Operatoren wie `+` und `/` schon durch ihre Verwendung für Integer belegt waren. `real.sl` überschreibt für sich die Operatoren. Zum Beispiel enthält es folgende Definitionen:

```
PUBLIC FUN + : Real -> Real -> Real
PUBLIC FUN / : Real -> Real -> Real
PUBLIC FUN == : Real -> Real -> Bool
PUBLIC FUN round : Real -> Int
PUBLIC FUN fromInt : Int -> Real
```

In einem anderen Modul kann somit also `(R.fromInt x) R.* 0.333` geschrieben werden. `real.sl` ist also für uns auch eine gute Möglichkeit, um das Zusammenspiel von aus dem Prelude importierten unqualifizierten Bezeichnern und Modulinternen deklamationen auszutesten.

6.4 Dictionaries — `dict.sl`

Anders als zum Beispiel `List` ist der abstrakte Datentyp `Dict` komplett ohne SLs algebraische Datentypen umgesetzt. Stattdessen arbeiten die Implementierungen der einzelnen Funktionen ausschließlich mit JavaScripts `Object`, also den in JavaScript grundlegenden Wörterbuchobjekten.

```
DATA EXTERN Dict a
PUBLIC FUN empty : Dict a
PUBLIC FUN put : Dict a -> String -> a -> Dict a
PUBLIC FUN has : Dict a -> String -> Bool
PUBLIC FUN get : Dict a -> String -> a
PUBLIC FUN getOpt : Dict a -> String -> Opt.Option a
PUBLIC FUN fromList : (String -> a) -> List.List String -> Dict a
```

`dict.sl` zeigt, wie man auch außerhalb des durch den SL-Compiler vorgesehenen besonderen Fleckchens `prelude.sl`, sinnvoll Strukturen durch Rückgriff auf JavaScript definieren kann, die auch mit rein SL-definierten Strukturen wie `List` und `Option` interagieren können.

6.5 `println-Debugging` — `debuglog.sl`

Das neue Modul `debuglog` erlaubt, normale Programme mit Konsolenausgaben zu versehen, die neben der Programmausführung ausgegeben werden.

```
PUBLIC FUN print : String -> DOM Void
PUBLIC FUN andPrint : a -> (a -> String) -> a
PUBLIC FUN andPrintMessage : a -> String -> a
```

Im Hintergrund bilden die Funktionen auf `console.log` ab, das unter `node.js` sowie neueren Versionen von Firefox (bzw. Firebug), Internet Explorer (ab IE8, Developer Tools) unauffällige Programmausgaben ermöglicht.

Allerdings bewegen sich `andPrint` sowie `andPrintMessage` und die Hilfsfunktion `logAvailable : Bool` am Rand des funktionalen Paradigmas.

```
I0.andPrint (L.Cons 1 (L.Cons 2 L.Nil)) (L.toString intToStr)
```

Dieser Ausdruck hat als Rückgabewert die Liste $\langle 1, 2 \rangle$, während als (fürs Programm hoffentlich unsichtbarer) Seiteneffekt, noch " $\langle 1, 2 \rangle$ " auf die Konsole geschrieben wird. Semantisch sollten `andPrint` sowie `andPrintMessage` äquivalent zur Identitätsfunktion mit ein paar unnötigen Parametern sein. Solange man es wie `Debug.Trace.trace` in Haskell nur vorsichtig für Debugging-Zwecke einsetzt, sollte alles klar gehen.

6.6 Browseranbindung — `basicweb.sl`

Wir schrieben auch eine kleine Bibliothek `basicweb`, die einige der Input/Output-Möglichkeiten von Websites bereitstellt. Diese Bibliothek ergibt natürlich nur sinn, wenn das mit SL erzeugte JS-Script im Browser ausgeführt wird.

```
DATA EXTERN Node
DATA EXTERN Document

PUBLIC FUN document : DOM Document
PUBLIC FUN getBody : Document -> DOM Node

PUBLIC FUN appendChild : Node -> Node -> DOM Void
PUBLIC FUN removeChild : Node -> Node -> DOM Void
PUBLIC FUN getChildNodes : Node -> DOM (List.List Node)

PUBLIC FUN setOnClick : Node -> DOM Void -> DOM Void
PUBLIC FUN getValue : Node -> DOM String
PUBLIC FUN setValue : Node -> String -> DOM Void

PUBLIC FUN createElement : Document -> String -> DOM Node
PUBLIC FUN createButton : Document -> String -> DOM Void -> DOM Node
PUBLIC FUN createInput : Document -> String -> DOM Void -> DOM Node

PUBLIC FUN alert : String -> DOM Void
PUBLIC FUN prompt : String -> String -> DOM String
```

Wir haben nur einen sehr kleinen Teil der Standard-JavaScript-Befehle abgebildet. Mit diesem Teil lässt sich schon eine überschaubare Webanwendung wie in `boxsort.sl` gezeigt umsetzen, die in gängigen modernen Browsern läuft.

6.7 Zusammenfassung

Die entwickelten Bibliotheken sind weit davon entfernt, durchdacht und ausgewachsen zu sein. Sie zeigen jedoch schon gut, wie unsere neuen Features es erlauben, verschiedene Funktionen in Modulen zu sammeln und diese Module aufeinander aufbauen zu lassen.

Es wird deutlich, dass die vorgeschlagenen **EXTERN**-Konstrukte es erlauben, auch funktionale Bibliotheken wie `dict.sl` ohne eingriffe in den Compiler zu entwickeln. Die monadischen JavaScript-Literale sind mächtig genug, um Aspekte wie die Interaktion mit dem Browser in Modulen wie `basicweb.sl` zusammenzufassen.

Das Prelude als echtes Modul umzusetzen, gestaltet auch den Compiler übersichtlicher. Die Prelude-Funktionen sind jetzt gleichberechtigte Funktionen innerhalb der Sprache und führen kein Eigenleben in Checks und Codegenerierung mehr.

7 Beispielprogramme

8 Fehlermeldungen

TODO (* Fritz *)

9 Zusammenfassung

TODO (* ... *)