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 Wörld *)

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.js[1], Firefox[2], Chrome[3] und Internet Explorer[4] 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 Modifikationdatum 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 (unterstützem) 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.js[5] statt CommonJS[6] 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 requirejs in node.js – im lokalen Verzeichnis oder global? in Systemvoraussetzungen für SL2 beschreiben *)

5.3 Build-Prozess

TODO (* implizit unqualifiziert importieres 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

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

6.1 Prelude

Alle vormals 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 auch alle Basistypen deklariert. (Allerdings sind diese gleichzeitig noch in den Compiler integriert, damit die Literale einen Typ unabhängig vom Prelude-Import erhalten können.)

Unter anderem sind `#` für Funktionskomposition und `id` als Identitätsfunktion im Prelude hinzugekommen.

```
PUBLIC DATA Bool = True | False
PUBLIC FUN not : Bool -> Bool

-- Function composition
PUBLIC FUN # : (b -> c) -> (a -> b) -> (a -> c)
PUBLIC FUN id : a -> a

-- String functions

PUBLIC FUN ++ : (String -> String -> String)

------------------------------------------

PUBLIC FUN intToStr : Int -> String
PUBLIC FUN strToInt : String -> Int
```

6.2 Error

```
PUBLIC FUN error : String -> a
DEF EXTERN error = {| function(msg){throw msg} |}
```

Eine spannende neue Funktion im Prelude ist `error`. Diese hat einen beliebigen Rückgabetyp, 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.[7] Das ist sehr praktisch, wenn man im Implementierungsprozess schon teile Testen möchte, aber noch nicht überall sinnvollen Code eintragen kann.

[7] Diese Funktion ist also keine echte, wohldefinierte Funktion, sondern hat dasselbe "Ergebnis" wie eine Endlosrekursion.

6.3 println-Debugging

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

Mit einem ähnlichen Trick arbeitet das neue Modul `basicio`. Dieses erlaubt, normale Programme mit Konsolenausgaben zu versehen, die neben der Programmausführung ausgegeben werden.

```
IO.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 `"<1,2>"` auf die Konsole geschrieben wird. Semantisch sollten `andPrint` sowie `andPrintDbg` äquivalent zur Identitätsfunktion mit ein paar unnötigen Parametern sein.

6.4 List, Option, Either

```
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
```

Unsere mitgelieferten Module enthalten die klassischen algebraischen, generischen Datentypen `List` (aka Sequence), `Option` (aka Maybe) und `Either` (aka Union). 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.

6.5 Dictionarys

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

```
DATA Dict a
PUBLIC FUN empty : Dict a
PUBLIC FUN put : Dict a -> String -> a -> Dict a
PUBLIC FUN get : Dict a -> String -> a
PUBLIC FUN has : Dict a -> String -> Bool
```

6.6 Browseranbindung

6.7 Floats

7 Fehlermeldungen

TODO (* Fritz *)

8 Zusammenfassung

TODO (* ... *)