Projektbericht: Erweiterung von SL um ein Modulsystem

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

Compilerbauprojekt SoSe 2013
Technische Universität Berlin

Zusammenfassung. Worum geht es hier?

Inhaltsverzeichnis

1 Einleitung

TODO (* Hällü Wörld *)

2 Überblick

Ziel des Projektes war es, SL um ein Modulsystem zu erweitertern. Dazu reicherten wir den Sprachumfang von SL mit einer Reihe von Konstrukten an. Für diese besprechen wir die Syntax genauer in Abschnitt 3, die semantische Analyse in Abschnitt 4 und die Codegenerierung in Abschnitt 5. Grob umfassen die neuen Features:

Module Jede SL-Datei wird jetzt als eine Modul-Definition aufgefasst.

Export von Bezeichnern Durch das `PUBLIC`-Keyword können Funktionsdeklarationen für andere Module importierbar gemacht werden.
`PUBLIC FUN f : Int -> Int` fügt `f : Int -> Int` der öffentlichen Signatur eines Moduls hinzu.
`PUBLIC DATA T = C` exportiert den Konstruktor `C : T`. Während der Übersetzung eine Moduls `code.sl` wird nicht nur eine `code.js`, sondern auch eine `code.signature` mit den Signaturinformationen angelegt.

Import von Modulen Module können importiert mittels `IMPORT` werden.
`IMPORT "code" AS M` macht die exportierten Definitionen aus `code.sl` unter dem lokalen Modulbezeichner `M` verfügbar.

Qualifizierte Bezeichner In Ausdrücken, Pattern und Typausdrücken können importierte Bezeichner vorkommen, beispielsweise so:

```
FUN g : M.T -> Int
DEF g M.C = M.f 23
```

Externe Definitionen Mithilfe von `EXTERN` lassen sich Funktionsdefinitionen als JavaScript angeben.

```
FUN myCast : String -> Int
DEF EXTERN myCast = {| parseInt |}
```

In diesem Code wird `myCast` durch JavaScripts `parseInt` definiert. Allerdings: `EXTERN` ist nicht zur Verwendung in normalen Modulen vorgesehen, sondern soll nur in der Definition von Bibliotheken zum Einsatz kommen. Aus einer `EXTERN`-Definition sollte auch nicht auf andere Definitionen des Moduls zugegriffen werden.

Externe Datentypen Durch die externen Definitionen werden häufig auch „externe“ Datentypen notwendig.
`DATA EXTERN Node` definiert einen Typ ohne Konstruktoren. Die Programmiererin hat dafür Sorge zu tragen, dass dadurch nicht versehentlich leere Typen entstehen.

Import von JavaScript-Code Zum Zusammenspiel mit `DEF EXTERN` gibt es auch noch die Möglichkeit, JavaScript-Code in ein Modul einzubinden.
`IMPORT EXTERN "_code"` zum Beispiel bindet die Datei `_code.js` direkt mit in das Compilat mit ein.

Darüber hinaus widmeten wir uns auch noch weiteren Themen:

Syntaxanpassungen Einige Details wie Inkonsistenzen zwischen den bestehenden Parsern und die implizite Klammerung bei Typausdrücken Zum Beispiel `Dat a -> b` wird jetzt als `(Dat a) -> b` und nicht mehr als `Dat (a->b)` gelesen. (Siehe Abschnitt 3.4)

Compilationdriver Das Ergebnis des SL-Compilers wird nicht mehr nach Stdout geschrieben, sondern in Dateien abgelegt. Dabei werden auch transitive Abhängigkeiten aufgelöst und notwendige zusätzliche Dateien mit angelegt. (Siehe Abschnitt 5.4)

Fehlermeldungen Fehlermeldungen von syntaktischer und semantischer Analyse sowie Modulauflösung enthalten jetzt für gewöhnlich auch Angaben zum Ort des Fehlers. Viele Fehlermeldungen sind etwas präziser geworden, auch wenn sie immer noch mäßig hilfreich beim produktiven Einsatz ohne tiefere Compilerkenntnisse sein dürften. (Siehe Abschnitt 6)

Prelude Die vormals hart in den SL-Übersetzer eingebauten grundlegenden Funktionen werden jetzt aus einem Modul importiert. (Siehe Abschnitt 7.1) Prelude wird durch jedes SL-Programm implizit unqualifiziert importiert. Unqualifizierte Imports, die nicht das Prelude betreffen, haben wir nicht vorgesehen und aktuell würden sie auch Probleme bei der Typprüfung verursachen.

Bibliotheken Wir haben einige simple Bibliotheken für `List`, `Option`, `Either`, `Real`, `Dict`, println-Debugging und Webentwicklung geschrieben. Diese stehen in mannigfaltiger Abhängigkeit voneinander und versuchen vielseitig Gebrauch von den neuen Features zu machen. (Siehe Abschnitt 7)

Beispielprogramme Zusätzlich zu den Bibliotheken programmierten wir einige ausführbare SL-Programme, die auf den Bibliotheken aufbauen. (Siehe Abschnitt 8)

Tests Für viele der neuen Features schrieben wir auch Unit-Tests. Der Großteil unserer Arbeit an dieser Front floss jedoch darein, die alten Unit-Tests an die neuen Features anzupassen. (Siehe Abschnitt 8)

3 Syntax und Parser

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

3.1 Grammatik

Die Grammatik von SL ist in der Abbildung 1, die lexikalische Struktur ist in Abbildung 2. Nicht erwähnt darin sind die Kommentare. SL folgt bei den Kommentaren der Tradition von Haskell; es gibt also einerseits Zeilenkommentare, die mit `--` eingeleitet werden, andererseits auch Blockkommentare, die von `{|` und `|}` eingeschlossen werden.

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{Toplevel}^{+} \\
\mathit{Toplevel} & ::= & \mathit{Import} \mid \mathit{Signature} \mid \mathit{FunDef} \mid \mathit{DataDef} \\
\mathit{Import} & ::= & \texttt{IMPORT EXTERN}\ \mathit{string} \\
 & \mid & \texttt{IMPORT}\ \mathit{string}\ \texttt{AS}\ \mathit{module} \\
\mathit{Signature} & ::= & [\texttt{PUBLIC}]\ \texttt{FUN}\ \mathit{var}\ \texttt{:}\ \mathit{Type} \\
\mathit{Type} & ::= & \mathit{BaseType}\ (\texttt{->}\ \mathit{BaseType})^{*} \\
\mathit{BaseType} & ::= & \mathit{typevar} \\
 & \mid & \mathit{TypeExpr} \\
 & \mid & \texttt{(}\ \mathit{Type}\ \texttt{)} \\
\mathit{TypeExpr} & ::= & \mathit{TypeCon}\ \mathit{BaseType}^{*} \\
\mathit{DataDef} & ::= & [\texttt{PUBLIC}]\ \texttt{DATA EXTERN}\ \mathit{string} \\
 & \mid & [\texttt{PUBLIC}]\ \texttt{DATA}\ \mathit{typecon}\ \mathit{typevar}^{*}\ \texttt{=}\ \mathit{DataBody} \\
\mathit{DataBody} & ::= & \mathit{con}\ \mathit{TypeArg}^{*}\ \left(\texttt{|}\ \mathit{con}\ \mathit{TypeArg}^{*}\right)^{*} \\
\mathit{TypeArg} & ::= & \mathit{typevar} \\
 & \mid & \mathit{TypeCon} \\
 & \mid & \texttt{(}\ \mathit{Type}\ \texttt{)} \\
\mathit{FunDef} & ::= & \texttt{DEF EXTERN}\ \mathit{var}\ \texttt{=}\ \mathit{JsQuote} \\
 & \mid & \texttt{DEF EXTERN}\ \mathit{op}\ \texttt{=}\ \mathit{JsQuote} \\
 & \mid & \texttt{DEF}\ \mathit{var}\ \mathit{Pat}^{*}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{DEF}\ \mathit{Pat}\ \mathit{op}\ \mathit{Pat}\ \texttt{=}\ \mathit{Expr} \\
\mathit{Pat} & ::= & \mathit{var} \\
 & \mid & \mathit{Con} \\
 & \mid & \texttt{(}\ \mathit{Con}\ \mathit{Pat}^{*}\ \texttt{)} \\
\mathit{Expr} & ::= & \texttt{IF}\ \mathit{Expr}\ \texttt{THEN}\ \mathit{Expr}\ \texttt{ELSE}\ \mathit{Expr} \\
 & \mid & \texttt{\textbackslash}\ \mathit{Pat}^{+}\ \texttt{.}\ \mathit{Expr} \\
 & \mid & \texttt{CASE}\ \mathit{Expr}\ \mathit{Alt}^{+} \\
 & \mid & \texttt{LET}\ \mathit{LocalDef}^{+}\ \texttt{IN}\ \mathit{Expr} \\
 & \mid & \mathit{Expr}\ \mathit{Expr} \\
 & \mid & \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
 & \mid & \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \\
 & \mid & \mathit{JsQuote}\ [\texttt{:}\ \mathit{Type}] \\
 & \mid & \mathit{Var} \mid \mathit{Con} \mid \mathit{int} \mid \mathit{real} \mid \mathit{char} \mid \mathit{string} \\
\mathit{Alt} & ::= & \texttt{OF}\ \mathit{PPat}\ \texttt{THEN}\ \mathit{Expr} \\
\mathit{PPat} & ::= & \mathit{var} \mid \mathit{con}\ \mathit{Pat}^{*} \\
\mathit{LocalDef} & ::= & \mathit{var}\ \texttt{=}\ \mathit{Expr} \\
\mathit{JsQuote} & ::= & \texttt{\{|}\ \mathit{string}\ \texttt{|\}} \\
\mathit{Var} & ::= & [\mathit{module}\ .]\ \mathit{var} \\
\mathit{Con} & ::= & [\mathit{module}\ .]\ \mathit{con} \\
\mathit{TypeCon} & ::= & [\mathit{module}\ .]\ \mathit{typecon}
\end{array}
$$

Grammar 1: Grammatik von SL

var	::=	*lowercase alphanum**
con	::=	*uppercase alphanum**
typevar	::=	*lowercase alphanum**
typecon	::=	*uppercase alphanum**
module	::=	*uppercase alphanum**
op	::=	*singleop* | *multiop*$^+$
singleop	::=	`!` | `§` | `%` | `&` | `/` | `?`
		| `+` | `*` | `#` | `-` | `<` | `>`
multiop	::=	*singleop* | `=`
lowercase	::=	`a` | $\cdots$ | `z`
uppercase	::=	`A` | $\cdots$ | `Z`
digit	::=	`0` | $\cdots$ | `9`
alphanum	::=	*lowercase* | *uppercase* | *digit*
int	::=	[`-`] *digit*$^+$
real	::=	[`-`] *realbody* [*exp*]
realbody	::=	*digit*$^+$ `.` *digit** | `.` *digit*$^+$
exp	::=	`e` *int* | `E` *int*

Grammar 2: Lexikalische Struktur von SL

3.2 Syntaxanpassungen

Operatoren In der ursprünglichen Version von SL wurde unterschieden zwischen eingebauten und selbst definierten Operatoren. Einige der vorgegebenen definierten Operatoren hatten überdies Namen, die für selbst definierte Operatoren nicht erlaubt wären, nämlich `+s` für die String-Konkatenation sowie die Gleitkommaoperatoren `+r`, `-r`, `*r` und `/r`. Außerdem war ein unäres Minus sowohl auf Gleitkomma- als auch auf Ganzzahlen definiert, was in der Sprache dahingehend einzigartig war, dass es ansonsten weder unäre Operatoren gibt, noch überladene Funktionen/Operatoren, noch Bezeichner, die je nach Position (präfix oder infix) eine unterschidliche Funktion bezeichnen, wie in diesem Fall das Minuszeichen einerseits die Negation auf Gleitkomma- und Ganzzahlen und andererseits die Subtraktion von Ganzzahlen bezeichnete.

Was die ungewöhnlich benannten Operatoren für Zeichenketten- und Gleitokmmaoperationen betrifft, so haben wir uns dafür entschieden, sie aus der Sprache zu entfernen. Dies verhindert auch Parserprobleme, wenn Bezeichner direkt hinter Operatoren geschrieben werden; so wurde etwa `1+sum` bisher als `1 +s um` geparst, was zu Verwirrung führen kann. Der Operator für die String-Konkatenation heißt nun `++`, die Gleitkommaoperationen sind nun in einem eigenen Modul untergebracht und müssen in Programmen entsprechend qualifiziert verwendet werden, z.B. schreibt man nun `1.0 R.+ 2.5` statt `1.0 +r 2.5`, falls das Modul *std/real* als `R` importiert wurde.

Das unäre Minus haben wir komplett aus der Sprache entfernt. Stattdessen gibt es jetzt in der Prelude eine Funktion `neg` für die Negation von Ganzzahlen;

außerdem können nun Gleitkomma- und Ganzzahlliterale ein Minus als Vorzeichen enthalten, wenn dieses nicht durch Leerschritte oder Klammern von der ersten Ziffer bzw. dem Dezimalpunkt getrennt ist. So sind beispielsweise die Schreibweisen `5 * (-1)` und `5 * -1` immer noch zulässig, nicht aber `5 * -(1)`, `5 * - 1` oder auch `5 * (-x)`. Problematisch bei dieser Lösung ist allerdings, dass sie beim Parsen auch zu uninutitiven Ergebnissen führt. Beispielsweise würde man erwarten, dass der Ausdruck `x-2` (ohne Leerschritte) als Subtraktion geparst wird, also gleichbedeutend mit `x - 2`. Tatsächlich wird der Ausdruck aber als Applikation von x mit dem Argument -2 geparst, gleichbedeutend mit als `x (-2)`.

Import-Statements Um das Modulsystem nutzen zu können, muss es eine Möglichkeit geben, andere Module zu importieren. Dazu haben wir ein *Import*-Statement eingebaut, mit der folgenden Syntax:

`IMPORT` *Modulpfad* `AS` *Modulbezeichner*

Dabei ist der *Modulbezeichner* ein großgeschriebener SL-Bezeichner, der zur Qualifikation von importierten Bezeichnern verwendet wird. Der *Modulpfad* der Pfad der Moduldateien, ohne Dateiendung. Wenn das Modul etwa aus der Datei *testmodule.sl* im aktuellen Verzeichnis kompiliert wurde, so heißt der *Modulpfad* einfach `"testmodule"`. Beginnt der Modulpfad jedoch mit dem Präfix *std/*, so wird das Modul in der Standardbibliothek gesucht. Der *Modulpfad* für das Listenmodul in der Standardbibliothek ist also `"std/list"`.

Die Sprache stellt nur Syntax für solche qualifizierten Importe zur Verfügung. Der einzige unqualifizierte Import eines Moduls ist der implizite Import von *std/prelude*, der automatisch bei jeder Kompilierung vorgenommen wird.

Wir haben uns für das Design mit den qualifizierten Imports entschieden, da es bei komplexeren Projekten beim Verständnis des Codes stark hilft, wenn leicht nachvollzogen werden kann, aus welchem Modul eine bestimmte Funktion oder ein Typ stammt. Eine alternative Möglichkeit, dieses Ziel zu erreichen, wäre beim Import alle zu importierenden Bezeichner einzeln aufzulisten. Dadurch wären zwar Ausdrücke sauberer lesbar, allerdings müsste dann bei jeder Verwendung einer neuen Funktion diese mühsam in der Importzeile hinzugefügt werden. Außerdem lässt sich auch mit unserer Lösung ein ähnliches Ergebnis erreichen, wie das folgende Beispiel zeigt:

```
IMPORT "std/list" AS List
DEF length = List.length
```

Dies funktioniert allerdings nur für Funktionen, nicht für Typen oder Konstruktoren. Eine dahingehende Erweiterung, vielleicht auch eine entsprechende Kurzschreibweise, wäre für die Zukunft sinnvoll. Ebenfalls wünschenswert wäre die Möglichkeit, das Modul *std/prelude* qualifiziert zu importieren und dadurch den unqualifizierten Import zu überschreiben. Dies wäre etwa sinnvoll für Anwendungen, die viel mit Gleitkommazahlen arbeiten und darum lieber die Operatoren `+`,

`-`, `*`, `/` aus *std/real* unqualifiziert verwenden würden. Es ist zwar jetzt schon möglich, so etwas zu schreiben wie `DEF a + b = a Real.+ b`, aber dadurch wird es unmöglich, die Addition aus der Prelude im gleichen Modul zu verwenden.

Externe Definitionen, externe Imports und externe Datentypen Bei der Implementation von Modulen, insbesondere des Moduls *std/prelude*, das Funktionen definiert, die vormals direkt in den Sprachkern eingebaut waren, wurden auch bestimmte Spracherweiterungen nötig. Für all diese Erweiterungen verwenden wir das gleiche Keyword `EXTERN`. Dies soll signalisieren, dass es sich dabei um Features handelt, die nur nötig sind, wenn man mit SL-„externem“, d.h. *JavaScript*-Code arbeitet.

Externe Definitionen erlauben den Aufruf von *JavaScript*-Code ohne Verwendung der DOM-Monade. Dadurch können Funktionen aus *JavaScript* verwendet werden, um die Primitiven von SL zu implementieren. Der Programmierer hat bei Verwendung von externen Definitionen selbst darauf zu achten, dass die Funktion keine Nebeneffekte hat und dass die Funktion wirklich den in der Signatur angegebenen Typ hat, ohne dass der Compiler dies prüfen kann. Die Syntax hierfür ist:

`DEF EXTERN` *Funtkions-/Operatorname* `= {|` *JavaScript-Code* `|}`

Dabei ist zu beachten, dass im Gegensatz zu normalen Funktions- bzw. Operatordefinitionen keine Argumente angegeben werden dürfen.

Externe Imports ermöglichen es, dem Compiler mitzuteilen, dass er den Inhalt einer beliebigen *JavaScript*-Datei bei der Codegenerierung vor die Ausgabe kopieren soll. Dadurch kann auf die in dieser Datei in *JavaScript*-Blöcken zugegriffen werden. Dies ist besonders hilfreich in Kombination mit externen Definitionen, wobei der tatsächliche Inhalt der *JavaScript*-Funktionen in der importen Datei liegt und in der externen Definition lediglich deren Name angegeben ist. Das Modul *std/prelude* ist etwa auf diese Weise geschrieben. Die Syntax für externe Importe ist:

`IMPORT EXTERN` *Pfad*

Dabei ist *Pfad* wieder ein Zeichenkettenliteral; die Endung *.js* entfällt dabei.

Externe Datentypen sind Datentypen ohne Konstruktoren. Dadurch können etwa verschiedene Typen von *JavaScript*-Objekten dargestellt werden. Die Basistypen `Int` und `String` gehören etwa in diese Kategorie. Die Syntax lautet:

`DATA EXTERN` *Typname*

Sichtbarkeit Funktionssignaturen und Datentypendefinitionen kann jetzt das Keyword `PUBLIC` vorangestellt werden. Dadurch werden die entsprechenden Funktionen bzw. Konstruktoren exportiert, sind also in anderen Modulen sichtbar. Nicht als `PUBLIC` markierte Datentypen werden ebenfalls exportiert, allerdings ohne ihre Konstruktoren.

3.3 Qualifizierte Bezeichner

Da wir Module im Allgemeinen qualifiziert importieren, muss auf importierte Bezeichner auch qualifiziert zugegriffen werden. Die entsprechende Syntax lautet:

Modulbezeichner.Bezeichner

Der *Modulbezeichner* ist ein großgeschriebener Bezeichner, wie er auch für Typen und Konstruktoren verwendet wird. Der *Bezeichner* ist ein gewöhnlicher SL-Typ-, Konstruktor- oder Funktionsbezeichner.

3.4 Weitere Anpassungen der bestehenden Grammatik

4 Semantische Analyse

Aufgabe der semantischen Analyse ist es, den Kontext des vom Parser eingelesenen Syntaxbaums zu überprüfen. Durch die Erweiterung der Sprache SL um ein Modulsystem musste die Analyse angepasst und ausgebaut werden. Ohne das Modulsystem war der Kontext auf eine Quelldatei sowie fest einprogrammierte Konstrukte (z.B. Operatoren für ganze Zahlen) beschränkt.

Da nun ein Modul auch andere Module importieren kann, erweitert sich der zu analysierende Kontext. Zum einen wurde die Grammatik um die `IMPORT`-Anweisung ergänzt. Einem Modul ist es beispielsweise nicht erlaubt ein anderes Modul mehrfach zu importieren. Zum anderen können Datentypen und Funktionen aus dem importierten Modul verwendet werden. Für das Type-Checking muss daher die sogenannte Signatur des Imports bekannt sein. Diese umfasst Datendefinitionen und Funktionssignaturen.

4.1 Auflösung von Importen

Nachdem die abstrakte Syntax vom Parser eingelesen wurde, müssen die Importe aufgelöst werden. Andernfalls ist Type-Checking nicht möglich, welches vor der Spracherweiterung direkt im Anschluss des Parsings stattfand. Die Auflösung von Importen bezeichnet das Suchen und Laden von Signaturen von externen Modulen. Zuvor müssen die Import-Anweisungen allerdings selbst auf Korrektheit überprüft werden.

Import-Überprüfung TODO (* Modulbezeichner richtiger Begriff? *)

Die Import-Anweisung ist im Grunde ein Paar aus Modulpfad und Modulbezeichner:

```
IMPORT "my/path/to/module-file" AS ModuleIde
```

Der Pfad gibt dabei an, wo das Modul zu finden ist. Der Modulbezeichner ermöglicht die Verwendung von Datentypen und Funktionen des Importierten Moduls.

Wir möchten verbieten, dass ein Modul mehrfach vom lokalen Modul importiert wird. Wir gehen dabei davon aus, dass ein Modul eineindeutig einem Pfad zugeordnet ist. Also überprüfen wir, ob jeder Pfad nur genau einmal vorkommt. Ebenso möchten wir nicht zulassen, dass ein Modulbezeichner für mehrere verschiedene Module verwendet wird.

Zum Beispiel ist folgende Importliste nicht erlaubt, da hier zwei Mal derselbe Modulbezeichner „`Duplicate`“ verwendet wird:

```
IMPORT "my/path/module-a"    AS Duplicate
IMPORT "other/path/module-b" AS Duplicate
IMPORT "my/path/module-v"    AS Innocent
```

Unsere Annahme, dass Module eindeutig über den Pfad identifiziert werden, kann in einigen Fällen jedoch unzureichend sein. Dies betrifft die Groß- und Kleinschreibung auf Windowssystemen sowie die Verwendung von „.“ und „..“. Daher verbieten wir die Verwendung von Punkten und Großbuchstaben.

Neben dem regulären Import existiert auch noch der unqualifizierte Import. Dieser ist für den Programmierer unzugänglich. Er dient zum Einbinden des Preludes, sodass Funktionen wie Addition auch ohne Modulbezeichner verwendet werden können. Der unqualifizierte Import ist von der Prüfung von doppelten Modulbezeichnern ausgeschlossen, jedoch nicht von Tests auf Pfade

Laut unserer Grammatik TODO (* tolle grammatik oder verweis zur grammatik *)

wird der Pfad zu einem externen Modul als String definiert. Der Parser akzeptiert daher jede Art von gültigen Strings. Deshalb muss an dieser Stelle die Pfadsyntax überprüft werden. Grammatik 3 zeigt die Produktionsregeln für Pfade. Erlaubt sind relative Pfade bestehend aus beliebig vielen Verzeichnissen und dem Moduldateinamen am Ende. Als Trennungssymbol dient das Schrägstrichsymbol ‚/‘. Verzeichnisse und Module dürfen Kleinbuchstaben, Zahlen, Minussymbole und Unterstriche enthalten. Der Modulname entspricht dem Dateinamen der Quelldatei ohne Endung „.sl“.

$$
\begin{array}{lll}
\mathit{Path} & ::= & (\mathit{Dir}\ \texttt{/})^*\ \mathit{Module} \\
\mathit{Dir} & ::= & \mathit{char}^+ \\
\mathit{Module} & ::= & \mathit{char}^+ \\
\mathit{char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \\
 & & \mid \texttt{0} \mid \cdots \mid \texttt{9} \\
 & & \mid \texttt{-} \mid \texttt{\_}
\end{array}
$$

Grammar 3: Gültige Importpfade

Dies sind Beispiele für korrekte Pfade:

```
IMPORT "module-a"     AS A
```

```
IMPORT "dir/module-b" AS B
IMPORT "123/module-c" AS C
IMPORT "module_4"     AS D
```

Inkorrekt sind dagegen die folgenden:

```
IMPORT "MoDuLe"                   AS A
IMPORT "module-b.sl"              AS B
IMPORT "dir/."                    AS C
IMPORT "/absolute/path/module-d" AS D
IMPORT "m.o.d.u.l.e"              AS E
IMPORT "./module-f"               AS F
IMPORT "d.i.r/module-g"           AS G
```

Laden der Signatur Falls die Importe korrekt sind, werden die dazugehörigen Moduldateien geladen. Ein Modul liegt dabei immer in zwei Dateien vor: Die Signatur („*.signatur“) und die in JavaScript übersetzte Implementierung („*.js“). Können nicht alle benötigten Dateien gefunden werden, so kann der Kompiliervorgang nicht fortgesetzt werden. Die Suche der Dateien erfolgt relativ zum Klassenpfad, Zielverzeichnis, TODO (* mainUnit? *)

und aktuellen Verzeichnis.

Die Signatur-Datei enthält einen Teil des abstrakten Syntaxbaums des zu importierenden Moduls. Datentypdefinitionen und Funktionssignaturen sind hier in einer serialisierten Form abgespeichert. Das Format dieser Datei wird in Abschnitt 5.2 beschrieben.

4.2 Type-Checking

Durch das Auflösen besteht nun Zugriff auf die Signaturen der Importe. Diese können in ihrer ursprünglichen Form aber noch nicht dem Type-Checker übergeben werden. Vorher erfolgt eine Normalisierungsphase.

Modulnormalisierung Jedes Modul kann wiederum Module importieren. Demzufolge ist es auch möglich, dass zwei importierte Module A und B von einem dritten Modul „List“ Gebrauch machen. Im folgenden Beispiel importiert das Modul C diese Module.

```
-- Module A --
IMPORT "std/list" AS List
PUPBLIC FUN foo : List.List -> List.List

-- Module B --
IMPORT "std/list" AS L
PUBLIC DATA MyType = Ctor L.List
```

```
-- Modul C --
IMPORT "a" AS A
IMPORT "b" AS B
IMPORT "std/list" AS StdList

FUN bar : B.MyType -> StdList.List
DEF bar x = CASE x OF B.Ctor l THEN A.foo l
```

Hier würde der Type-Checker nicht wissen, dass der Typ `List.List` aus Modul A derselbe ist wie `L.List` in Modul C. Deshalb müssen die aufgelösten Importe normalisiert werden. Die Idee dahinter ist, dass jedem Modul genau ein Modulbezeichner zugeordnet wird. Dieser Bezeichner wird vom lokalen Modul, hier C, bestimmt. Das bedeutet, dass der Modulbezeichner für das Modul List durch `StdList` ersetzt wird.

Nach der Normalisierung sieht die abstrakte Syntax der Importierten Module also so aus:

```
-- Module A --
IMPORT "std/list" AS StdList

PUBLIC FUN foo : StdList.List -> StdList.List

-- Module B --
IMPORT "std/list" AS StdList

PUBLIC DATA MyType = Ctor StdList.List
```

Diese Ersetzung ist möglich, da auch C das Modul List importiert. Anhand des Pfades `"std/list"` kann dieses Modul auch in den anderen Modulen erkannt und der Bezeichner substituiert werden.

Es kann jedoch auch der Fall eintreten, dass C das List-Modul unbekannt ist. In diesem Fall wird ein neuer Bezeichner generiert. Dieser hat die Form `#<Nummer>` wobei `<Nummer>` durch eine fortlaufende Nummer ersetzt wird. Durch dieses Vorgehen kann auch das folgende Beispielprogramm typgeprüft werden:

```
-- Module A --
IMPORT "std/list" AS L
IMPORT "std/option" AS O

PUBLIC FUN foo : O.Option -> Int

-- Module B --
IMPORT "std/list" AS L
IMPORT "std/option" AS O

PUBLIC FUN bar : List.List -> O.Option
```

```
-- Modul C --
IMPORT "a" AS A
IMPORT "b" AS B
IMPORT "std/list" AS L

FUN baz : L.List -> Int
DEF baz l = B.foo(A.bar(l))
```

Das Modul Option ist C unbekannt. Es kann also keine Funktionen oder Typen aus diesem Modul direkt verwenden. Dennoch ist es möglich die Funktionen `foo` und `bar` wie in Modul C aufzurufen. Während der Normalisierung wird der Modulbezeichner `0` in A und B durch `#1` ersetzt.

Modulkontext Vor dem eigentlichen Type-Checking werden die Datentyp- und Funktionsdefinitionen überprüft. Einige der Tests mussten erweitert werden, um auch importierte Definitionen berücksichtigen zu können.

Im Falle der Datentypdefinitionen gibt es die zwei Tests, `checkNoUndefinedTypeCons` und `checkTypeConsApp`, welche auch den Kontext von Importen beachten müssen. `checkNoUndefinedTypeCons` überprüft ob die im Programm verwendeten Konstruktoren existieren. Bisher war dies auf den lokalen Kontext beschränkt. Da jedoch auch Konstruktoren aus anderen Modulen verwendet werden können sollen, werden jene zu der Menge der bekannten Konstruktoren hinzugefügt. Aus demselben Grund musste auch `checkTypeConsApp` erweitert werden. Diese Funktion testet, ob genügend Parameter für einen Konstruktor angegeben wurden.

Da Datendefinitionen von unqualifizierten Importen Probleme verursachen können, müssen diese in weiteren Tests betrachtet werden. Typbezeichner und Konstruktoren dürfen nicht mit lokalen Definitionen in Konflikt geraten. Deshalb werden jene importierten Datendefinitionen in den Tests `checkTypeConsDisjoint` und `unqualifiedImportedDataDefs` ebenfalls betrachtet

Für den Type-Check ist es notwendig den initialen Kontext um den Modulkontext zu erweitern. Dazu muss der Modulkontext zunächst gebildet werden. Dies geschieht auf ähnliche Weise, wie auch der Kontext des lokalen Moduls aufgebaut wird. Alle Konstruktoren der Module erhalten ein Typschema und werden zusammen mit den Funktionen dem Kontext hinzugefügt.

Die lokalen Funktionssignaturen und -definitionen sowie die Signaturen aus den externen Modulen werden genutzt um das Programm in ELC[1] zu übersetzen. Die Übersetzung wird dann zusammen mit dem initialen Kontext inklusive Modulkontext an den Type-Checker übergeben, der weitestgehend unangerührt blieb.

[1] Enriched Lambda Calculus

5 Codegenerierung

Der SL2-Compiler generiert Code, der unter node.js[2] und im Browser[3] ausgeführt werden kann. Bei einem Compileraufruf können jeweils mehrere Dateien angebenen werden (siehe Abschnitt 5.1). Während der Compilierung werden aus dem `<classpathDirectory>` die Signatur-Dateien (siehe Abschnitt 5.2) bereits kompilierter Module geladen. Anschließend werden die angegebenen `<module files>` sowie von diesem verwendete Module, kompiliert (Details siehe Abschnitt 5.4). Während der Codegenerierung werden Signaturen (siehe Abschnitt 5.2), sowie JavaScript-Dateien (siehe Abschnitt 5.4) für alle kompilierten Module erstellt, wobei require.js (siehe Abschnitt 5.3) verwendet wird, um die JavaScript-Dateien der Module zu laden. Sofern das beim Aufruf des Compilers angegebene `<module files>` 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 5.4).

TODO (* Ausführung des Codes in (unterstützem) Browser und node.js, Voraussetzungen dafür (require.js muss ggf. im node.js installiert werden) *)

TODO (* Re-Kompilierung der Standardbibliothek: erst prelude.sl, dann buildstd.sl *)

TODO (* Anpassung der Pfade zur Standardbibliothek in main.js *)

5.1 Compileraufruf und Pfadangaben

Bei Aufruf des Compilers mit

```
$> <PROGRAMM-NAME> [-d <output directory>] [-cp <classpath directory>]
-sourcepath <source directory> <module files>
```

TODO (* PROGRAMM-NAME *)

bzw. im Scala Build Tool:

```
run-main de.tuberlin.uebb.sl2.impl.Main  [-d <output directory>]
[-cp <classpath directory>] -sourcepath <source directory>
<module files>
```

TODO (* Syntax für optionale Argumente *)

TODO (* die Verzeichnisse erklären *)

TODO (* Standard-Bibliotheken *)

TODO (* Überleitung zu den Signaturen *)

5.2 Signaturen

Wie bereits in Abschnitt 4 erläutert, ist es notwendig Signaturen für Module zu erzeugen. Dies ist ein neues Merkmal, welches vor Einführung des Modulsystems

[2] http://nodejs.org/ - getestet mit Version 0.10.10

[3] getestet mit Firefox, Chrome und Internet Explorer

nicht erforderlich war. Die JavaScript-Datei, welche den übersetzten SL-Code enthält ist unzureichend um die Signatur eines Moduls auszulesen. Die Signatur umfasst die Funktionssignaturen, Datendefinitionen und eine Liste von Importen des Moduls. Mit diesen Informationen ist es später möglich eine Typüberprüfung durchzuführen und transitive Importe zu erfassen.

Die Signatur wird als Datei abgespeichert. Dazu muss sie zunächst jedoch serialisiert werden. Es gab mehrere Alternativen, wie diese Serialisierung umgesetzt werden kann. Wir haben drei Möglichkeiten betrachtet:

1. Die Signatur als SL-Code ausgeben und zum Deserialisieren erneut parsen.
2. Die in Java/Scala eingebaute Serialisierungsfunktion von Objekten verwenden.
3. Sie in ein JSON-Objekt umwandeln.

Die erste Möglichkeit hat den Vorteil einfach lesbar und editierbar zu sein. Allerdings sind wir dabei auf die Ausdrucksmöglichkeiten von SL beschränkt. Dies könnte spätere Erweiterungen erschweren, wenn zum Beispiel Metadaten abgespeichert werden sollen.

Die Möglichkeit in Java nahezu beliebige Objekte serialisieren und später wieder laden zu können wäre eine einfache und schnelle Möglichkeit gewesen einen Teil des Syntaxbaums als Datei abzuspeichern. Allerdings ist diese schwer zu lesen und von Hand zu editieren.

Aufgrund der beiden Schwächen der ersten beiden Möglichkeiten haben wir uns für die dritte entschieden. Sie ermöglicht die erwünschten Freiheiten und bleibt dennoch von Hand editierbar. Die Implementierung war ebenfalls unproblematisch. Prinzipiell hätte man auch ein anderes bekanntes Format wie z.B. XML verwenden können. Der Vorteil von JSON ist jedoch, dass es Teil der JavaScript-Syntax ist und damit nativ von JavaScript eingelesen werden kann. Zwar haben wir dies in unserem Projekt noch in keiner Form ausgenutzt, kann später jedoch von Vorteil sein.

Da die Signatur selbst ein Teil der abstrakten Syntax ist, haben wir uns deren Struktur als Beispiel für die JSON-Objektstruktur genommen. Diese Struktur ist am besten anhand eines Beispiels zu erläutern. Die Signatur des folgenden Moduls soll zu JSON serialisiert werden:

```
IMPORT "some/module" AS M

PUBLIC DATA MyType a = Ctor1 a | Ctor2

PUBLIC FUN foo : MyType Int -> Int
DEF foo x = ...
```

Das Wurzelobjekt ist ein JSON-Hash der stets die drei Elemente `imports`, `signatures` und `dataDefs` enthält. Dies sind auch die Bezeichner der korrespondierenden Felder aus der Klasse `Program`, welches den abstrakten Syntaxbaum speichert.

`imports` ist ein Array, welches alle Importfelder aus Modulpfad und -bezeichner beinhaltet:

```
"imports" : [
  {
    "name" : "M",
    "path" : "some\/module"
  }
]
```

Dagegen ist `signatures` ein Hash, wie das Vorbild aus `Program` welches eine `Map[VarName, FunctionSig]` ist. In diesem Hash werden Bezeichner und Typ der Funktionen zugeordnet.

```
"signatures" : {
  "foo" : [
    {
      "type" : ".MyType",
      "params" : [{"type" : ".Int", "params" : []}]
    },
    {
      "type" : ".Int",
      "params" : []
    }
  ]
}
```

In diesem Beispiel sehen wir auch, wie der Typ der Funktion `foo` serialisiert wird. Es gibt drei Sorten von Typen:

Typvariablen die durch einen beliebigen Typen ersetzt werden können.
Funktionstypen welche prinzipiell eine Auflistung von Typen ist. Der Funktionstyp `A -> B -> C` wird als Liste der Typen A, B und C dargestellt.
Typausdruck ist ein konkreter singulärer Typ, welcher denselben Bezeichner wie in seiner `DATA`-Definition beinhaltet. Außerdem können diese Parametertypen entgegennehmen, welche wiederum eine Liste von Typen ist.

Typvariablen werden in JSON als einfache Zeichenkette dargestellt. Funktionstypen sind Arrays, welche weitere Typen beinhalten. Der Typausdruck ist ein zwei-elementiger Hash mit den Werten `type` für den Bezeichner und `params` welcher eine Liste von Typen speichert. Da für jede Art von Typ ein anderer JSON-Objekttyp verwendet wird, sind die Typarten auch in JSON einfach voneinander zu unterscheiden.

Zuletzt müssen noch die Datentypdefinitionen serialisiert werden. Diese werden wie Importe in einem Array aufgezählt. Eine einzelne Definition ist wiederum ein Hash, der die relevanten Felder der `DataDef`-Klasse enthält. Dies sind `ide` für den Typbezeichner, eine Liste `tvars` aus Typvariablen und eine Liste von Konstruktoren `constructors`. Ein Konstruktor besteht wiederum aus einem Bezeichner `constructor` und eine Liste aus Typvariablen `types`.

```
"dataDefs" : [
  {
    "ide" : "MyType",
    "tvars" : ["a"],
    "constructors" : [
      {
        "constructor" : "Ctor1",
        "types" : ["a"]
      },
      {
        "constructor" : "Ctor2",
        "types" : []
      }
    ]
  }
]
```

5.3 require.js

Um die Module zur Laufzeit in JavaScript zu laden, wurde require.js[4] statt CommonJS[5] ausgewählt, da es im Gegensatz zum Modulsystem von node.js auch ohne Umwege sowohl im Browser als auch in node.js verfügbar ist.

In require.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(); }
  };
});
```

[4] http://requirejs.org/ v. 2.1.6
[5] http://www.commonjs.org/

```
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 *)

5.4 Build-Prozess

TODO (* MultiDriver, ModuleLinearization, zyklische Abhängigkeiten *)
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 *)
TODO (* wenn mehr als ein Modul eine main-Methode bereitstellt, kann nur die zuletzt ausgeführte verwendet werden *)
TODO (* Hierarchie der generierten Dateien *)

5.5 Externe Definitionen

Besonders beim Schreiben von Funktionsbibliotheken muss man häufig auf JavaScript-Funktionen zugreifen, ohne dass man dabei in eine `DOM`-Monade geraten will. Das ursprüngliche SL erlaubte nur JS-Code-Literale vom Typ `DOM x`. Da es kein `return` für `DOM` gibt und auch keins geben soll, ist es damit nicht möglich, neue Funktionen in SL zu schreiben, die einen anderen Rückgabetyp als `DOM` besitzen.

Um dennoch das Prelude von SL und weitere Bibliotheken (Abschnitt 7) in SL verfassen zu können, haben wir durch `DEF EXTERN` einen sehr definierten Platz geschaffen, an dem ein JavaScript-Literal `{|someJsCode()|}` ausgepackt werden darf. Die Übersetzung dazu ist vergleichsweise simpel:

`DEF EXTERN bla = {| js |}` $\rightsquigarrow$ `var bla = js;`

Um auf auf der rechten Seite dieser Definition bequem selbstdefinierte JavaScript-Funktionen anzugeben, bietete es sich an, diese in einem weiteren Modul zusammenzufassen und sie dann per `IMPORT EXTERN` einzubinden. Die Übersetzung zu `IMPORT EXTERN "<datei>"` ist dabei schlicht, dass an den Anfang des Kompilats der Inhalt der Datei `"datei.js"` gesetzt wird.

6 Fehlermeldungen

TODO (* Fritz *)

7 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/`.

7.1 Prelude

Fast 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 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 Funktionskomposition[6] 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 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] 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} |}
```

7.2 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
```

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

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

7.3 Reele 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 Verwendeung 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 deklarationen auszutesten.

7.4 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.

7.5 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 Expolorer (ab IE8, Developer Tools) unauffällige Programmausgaben ermöglicht.

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

```
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 `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.

7.6 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.

7.7 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-Funkionen sind jetzt gleichberechtigte Funktionen innerhalb der Sprache und führen kein Eigenleben in Checks und Codegenerierung mehr.

8 Beispielprogramme und Tests

8.1 Beispielprogramme

Wir haben eine Reihe kleinerer Testprogramme geschrieben.

`hello.sl` Das minimale „Hello World“-Programm. Verwendet nur `debuglog`.
`helloworld.sl` Spielt mit diversen Grundlagen aus `list`, `option` und `dict`.
`transitiveimports.sl` Verwendet `option`, ohne es direkt zu importieren. Stattdessen werden `dict` und `list` benutzt. (Das ist ein wichtiger Testfall!)
`similarimports.sl` Importiert eine neudefinierte `Option` und zeigt, dass sie nicht mit `Option` aus `std/option` clasht.
`librarytest.sl` Testet das Zusammenspiel einiger Funktionen aus `list` und `dict` sowie `real`. Macht außerdem vom lokalen Überschreiben von Prelude-Bezeichnern Gebrauch.
`boxsort.sl` Größeres Beispiel, das mittels `basicweb` eine interaktive Website erzeugt. Macht starken Gebrauch von allen möglichen Features aus den `std`-Librarys.
`koch.sl` Modifizierte Version des ursprünglichen Kochkurvenbeispiels. Verwendet Browseranzeige und `timing`, um eine Animation ausgehend von der Kochkurve zu zeichnen.
`sub/hello.sl` Gibt "Hello Moon!äus und wird von `moonWorld.sl` verwendet, um das Einbinden verschieden qualifizierter Module mit identischem einfachen Namen zu testen.
`moonWorld.sl` Verwendet `hello.sl` und `sub/hello.sl`, um zu prüfen, ob die beiden Module getrennt kompiliert und eingebunden werden.
`cycleA.sl` und `cycleB.sl` enhalten eine direkte zyklische Abhängigkeit, um zu testen, ob diese erkannt wird.

9 Zusammenfassung

TODO (* ... *)