

SL2: Die Simple Language mit Modulsystem

Benjamin Bisping, Rico Jasper,
Sebastian Lohmeier und Friedrich Psiorz

Compilerbauprojekt SoSe 2013
Technische Universität Berlin
20.09.2013

Einführung

Syntax und Parser

Semantische Analyse

Codegenerierung und Signaturen

Fehlermeldungen

Prelude und Bibliotheken

Beispielprogramme und Tests

Fazit

Einführung

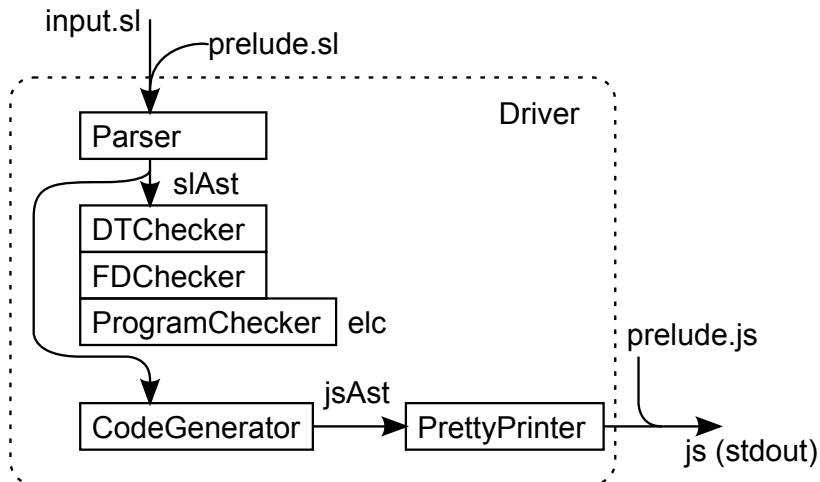
SL: typischer und funktional im Browser (JavaScript)



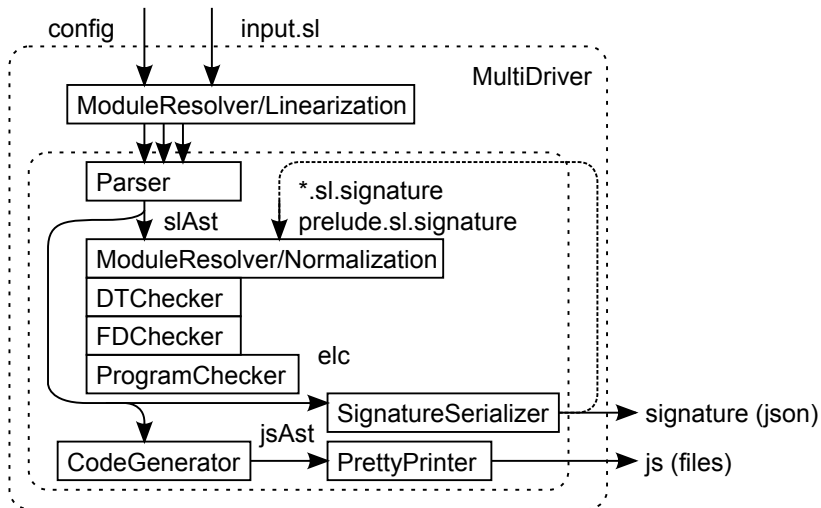
SL2: unabhängig kompilierbare Module

- ▶ Moduldefinition und -import (auch für das Prelude)
- ▶ Export und einfache Qualifizierung von Funktionen und Datentypen
- ▶ Einbindung von Funktionen und Datentypen aus JavaScript
- ▶ Anpassungen der Syntax und Semantik
- ▶ Fehlermeldungen verbessert
- ▶ Compilierung ins Dateisystem
- ▶ Bibliotheken, Beispielprogramme und Tests

Altes Framework



Neues Framework



Syntax – Ausgangspunkt

Typisch funktionale Syntax, ähnlich Haskell und Opal.
Besonderheiten:

- ▶ JavaScript-Blöcke:

```
{ | /* JS-Code */ | } : DOM Void
```

- ▶ Fest eingebaute Funktionen und Operatoren:

- ▶ Standard-Operatoren für Ganzzahl-Arithmetik und Vergleiche
- ▶ `+s`, `+r`, `*r`, etc. für Zeichenketten- und Gleitkomma-Operationen
- ▶ Unäres Minus auf Ganzzahlen (teilweise auch Gleitkomma)
 - Einziger unärer Operator, einzige überladene Funktion
- ▶ `&` sowie `&=` für Bind-Operation auf DOM-Monade

- ▶ Eigene Funktionen und (binäre) Operatoren definierbar

Syntax – Zielsetzung

- ▶ Unterscheidung zwischen eingebauten und selbst definierten Operatoren aufheben
- ▶ Modulsystem – Syntax für Import, Export und Zugriff auf importierte Bezeichner
- ▶ „Weniger Magie, mehr Bibliotheken“ – Auch Basis-Operatoren und -Funktionen sollten in der Prelude und selbst geschriebenen Bibliotheken definiert werden können

Anpassungen der Operatoren

In SL2 werde alle Operatoren, abgesehen von der Präzedenz, gleich behandelt:

- ▶ Dürfen keine alphanumerischen Zeichen enthalten
- ▶ Werden nicht überladen
- ▶ Unäres Minus fällt weg, stattdessen Zahlen-Literale mit negativem Vorzeichen
 - ▶ Erlaubt: -2 , $-.34e-13$
 - ▶ Nicht erlaubt: $-x$, -2 , $-(2)$
 - ▶ Unintuitiv: $x-2$ ist Applikation x (-2)

⇒ Nicht schön, aber konsistenter als SL1

Syntax für das Modulsystem

Qualifizierter Import eines Moduls:

```
IMPORT "path/to/module" AS MyModule
```

Zugriff auf importierte Bezeichner:

```
MyModule.function
```

```
MyModule.Type
```

```
MyModule.Constructor
```

Export von Funktionen und Konstruktoren:

```
PUBLIC DATA MyType = Cons1 | Cons2 | Cons3
```

```
PUBLIC FUN myFun : MyType -> Int
```

Low-Level-Funktionalitäten

Mit dem Keyword `EXTERN` kann direkt auf die JavaScript-Ebene zugegriffen werden.

- ▶ Definition von Funktionen in JavaScript, ohne DOM-Monade:

```
DEF EXTERN function = { | js-code | }
```

- ▶ Direktes Einfügen von JavaScript-Code in die Ausgabe:

```
IMPORT EXTERN "path/to/js-file"
```

- ▶ Definition von Typen ohne Konstruktoren:

```
DATA EXTERN TypeName
```

Beispiel

Auszug aus *std/prelude*:

```
-- Einfuegen der Datei _prelude.js
IMPORT EXTERN "std/_prelude"

-- Integer Datentyp ohne Konstruktoren
DATA EXTERN Int

-- Funktion _add definiert in _prelude.js
PUBLIC FUN + : Int -> Int -> Int
DEF EXTERN + = {| _add |}
```

Parser

Zwei Parser-Implementierungen:

- ▶ *Parboiled-Parser* war Standard-Parser in SL1.
- ▶ *Combinator-Parser* hatte zu Beginn noch nicht alle Features von SL1, ist jetzt unser Standard-Parser wegen besserer Lokalisierung von Knoten im AST.

Beide Parser parsen SL2 korrekt.

Semantische Analyse

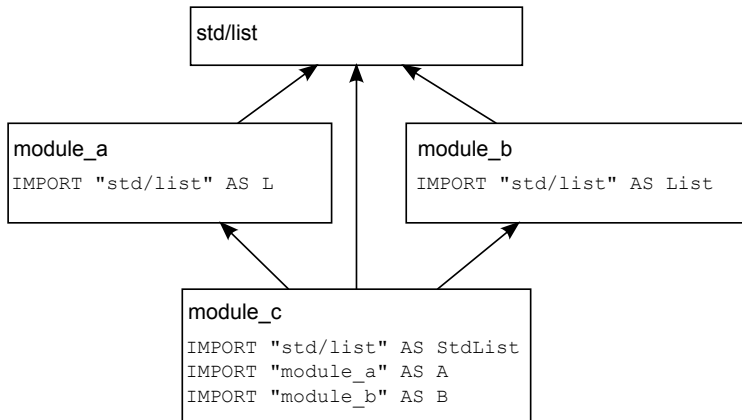
1. Auflösung von Importen
2. Modulnormalisierung
3. Datentypen und Funktionen überprüfen
4. Type-Checking

Import-Überprüfung

- ▶ Import-Anweisung: Paar aus Pfad und Bezeichner
`IMPORT "path/to/module" AS MyModule`
- ▶ eindeutige Modul-Bezeichner-Zuordnung
- ▶ Annahme: genau ein Pfad identifiziert ein Modul
- ▶ erlaubte Pfade:
 - ▶ Kleinbuchstaben
 - ▶ Zahlen
 - ▶ Minus (-) und Unterstrich (_)
 - ▶ relative Pfade

Modulnormalisierung

- ▶ keine modulübergreifende Modul-Bezeichner-Zuordnung
- ▶ Normalisierung erforderlich



- ▶ Substitution von `L` bzw. `List` durch `StdList`

Kontextprüfung

- ▶ Berücksichtigung von importierten Datentypen und Funktionen
- ▶ initialer Kontext um Modulkontext erweitert
- ▶ Type-Checker weitestgehend unverändert

Codegenerierung und Signaturen

1. Modulsignatur
2. Compileraufruf und Pfade
3. Abhängigkeitsanalyse
4. require.js
5. Code-Generierung

Modulsignatur

- ▶ Signatur für semantische Analyse erforderlich
- ▶ Inhalt:
 - ▶ Importliste
 - ▶ Datendefinitionen
 - ▶ Funktionssignaturen
- ▶ Mögliche Signaturformate:
 - ▶ native Serialisierung
 - ▶ Simple Language
 - ▶ JavaScript Object Notation

Modulsignatur – JSON

```
IMPORT "some/module" AS M
```

JSON:

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

Compileraufruf und Pfade

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

Abhängigkeitsanalyse I

Ein Modul ist zu kompilieren, wenn

1. Quell-Datei in `<module files>`, oder
2. importiert
und Quell-Datei im `<source directory>`
keine Signatur-Datei im `<classpath directory>`, oder
3. importiert
und Quell-Datei im `<source directory>`
und Signatur-Datei im `<classpath directory>`
und Quell-Datei jünger als Signatur-Datei.

Abhängigkeitsanalyse II

*A.sl → B.sl
A.sl.signature B.sl.signature

A.sl → *B.sl
A.sl.signature B.sl.signature

A.sl → B.sl → *C.sl
A.sl.signature B.sl.signature C.sl.signature

require.js

require.js statt Common.js

Installation in node.js (u.U. relativ zum akt. Verzeichnis)

```
> npm install requirejs
```

Code-Generierung I

```
> run-main de.tuberlin.uebb.sl2.impl.Main -sourcepath  
src/main/sl/examples/ boxsort.sl
```

```
boxsort.sl.signature
```

```
boxsort.sl.js
```

```
main.js
```

```
require.js
```

```
index.html
```


Code-Generierung II

```
IMPORT "std/debuglog" AS Dbg
```

```
...
```

```
PUBLIC FUN main : DOM Void
```

```
DEF main =
```

```
    Web.document &= \ doc .
```

```
    ...
```

```
DEF getNode (NodeWithNumber n1 i1) = n1
```

```
...
```

Code-Generierung III: boxsort.sl.js

```
define(function(require, exports, module) {  
  var $$std$prelude = require("std/prelude.sl");  
  var Dbg = require("std/debuglog.sl");  
  ...  
  function $getNode(_arg0) { ... };  
  ...  
  var $main = function () { ... }();  
  exports.$main = $main  
});
```

Code-Generierung IV: main.js

```
if (typeof window === 'undefined') {  
  /* in node.js */  
  var requirejs = require('requirejs');  
  
  requirejs.config({  
    //Pass the top-level main.js/index.js require  
    //function to requirejs so that node modules  
    //are loaded relative to the top-level JS file.  
    nodeRequire: require,  
    paths: {std : "C:/Users/monochromata/git/sl2/target/  
      scala-2.10/classes/lib" }  
  });  
  
  requirejs(["boxsort.sl"], function($$$boxsort) {  
    $$$boxsort.$main()  
  });  
...  
...
```

Code-Generierung V: main.js

```
...  
} else {  
  require.config({  
    paths: {std : "file:/C:/Users/monochromata/git/sl2/  
      target/scala-2.10/classes/lib/" }  
  });  
  
  /* in browsers*/  
  require(["boxsort.sl"], function($$$boxsort) {  
    $$$boxsort.$main()  
  });  
}
```

Fehlermeldungen – Ausgangspunkt

Die bisherige Fehlerbehandlung in SL war unzureichend

- ▶ Parser parst nur bis zum ersten Syntaxfehler, gibt aber bis dahin gültige Teile des Programms einfach weiter
- ▶ Statt Fehlermeldungen werden Scala-Objekte ausgegeben
- ▶ Teilweise unbehandelte Exceptions; der Compiler beendet sich mit Stacktrace

⇒ Absolut unzureichend für jedes Projekt, das mehr als ein paar Zeilen Code umfasst.

Fehlermeldungen – Format

```
/path/to/file.sl:5:1-23: Use of undefined  
type(s) in 'Foo': 'Foo.Bar'
```

Das allgemeine Format ist folgendes:

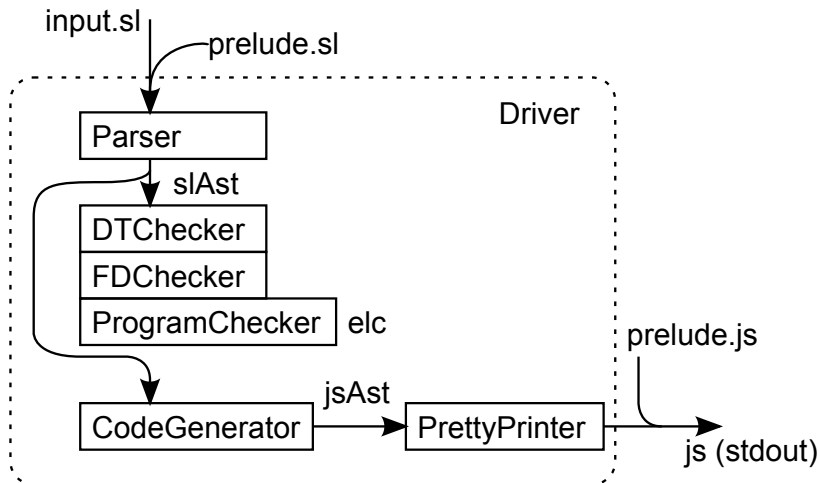
Dateiname : Zeile(n) [: Spalte(n)] : Fehlermeldung

Lokalisierung mit Zeilen- und Spaltennummer nur mit
Combinator-Parser

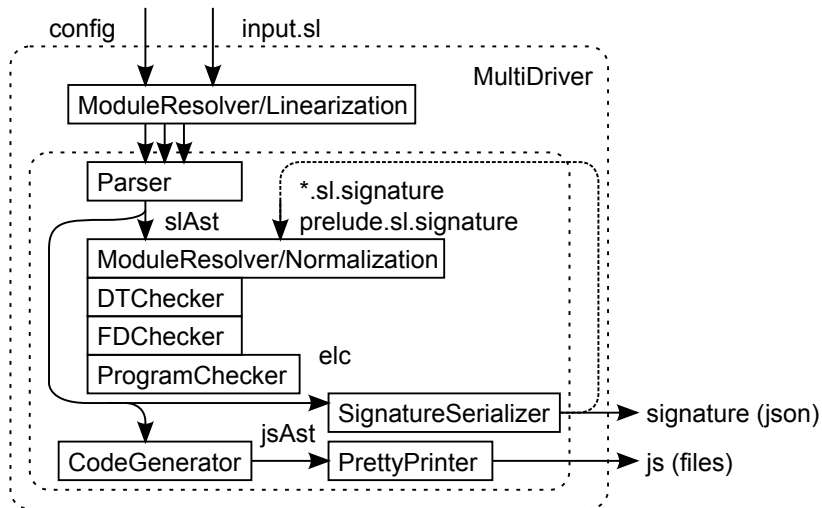
Fehlerarten

- ▶ Syntaktische Fehler
 - ▶ Häufige Fehler haben eigene Produktionen im Parser
 - ▶ Teilweise nur kryptische Fehlermeldungen der Parsec-Bibliothek
 - ▶ Abbruch nach erstem syntaktischen Fehler
- ▶ Semantische Fehler
 - ▶ Typfehler werden erkannt, Ort teils unintuitiv
 - ▶ Doppelte Deklarationen, fehlende Definitionen, falsche Aritäten, etc. werden mit Ort(en) zurückgegeben
- ▶ Importfehler
 - ▶ Moduldateien nicht vorhanden: Ort des Import-Statements sowie Suchpfad für Datei werden ausgegeben
 - ▶ Zyklische Importe, Qualifizierter Import der Prelude
- ▶ Laufzeitfehler werden vom JavaScript-Interpreter behandelt

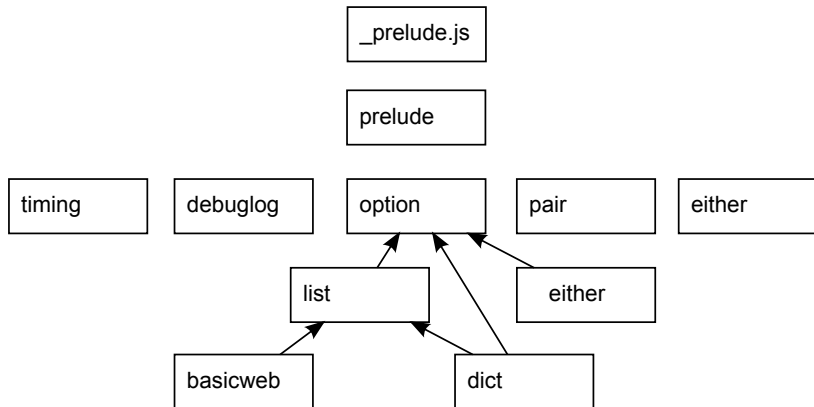
Prelude im alten Framework



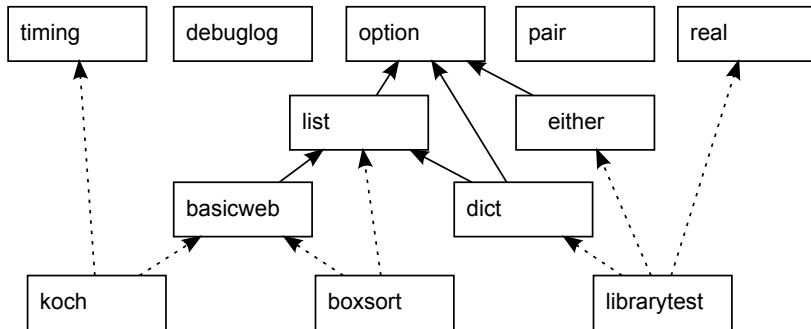
Prelude im neuen Framework



Bibliotheken



Beispielprogramme



Fazit I

- ▶ Modulare typsichere Webanwendungen im Browser und node.js möglich
- ▶ Modulimporte, qualifizierte Bezeichner, Exporte
- ▶ Fehlermeldungen verbessert
- ▶ Prelude in Module überführt
- ▶ initiale Standard-Bibliothek erstellt

→ Pflichtenheft erfüllt

Fazit II

Mögliche Erweiterungen

- ▶ Flexiblerer Import
- ▶ Statische zyklische Abhängigkeiten
- ▶ Konfiguration der Codegenerierung für require.js
- ▶ Verbesserte Typchecker-Fehlermeldungen
- ▶ Erweiterte Bibliotheken