The SL language and compiler

Andreas Büchele Christoph Höger Fabian Linges
Florian Lorenzen Judith Rohloff Martin Zuber

April 16, 2013

1 Introduction

SL (Simple Language) is very simple, purely functional programming language, mainly developed for teaching purposes at TU Berlin. The current implementation of the compiler translates SL programs to JavaScript.

This document briefly describes the language, its standard prelude, and the compiler. We assume familiarity with the basic concepts of functional programming languages.

1.1 Symbols used in syntactic descriptions

We use standard EBNF descriptions for the languages' syntax. Symbols in `monospaced` font represent terminals, symbols in *italics* represent nonterminals. We furthermore use the following operators:

α^*	zero ore more repetitions of α
α^+	one ore more repetitions of α
$\alpha\otimes$`T`	zero ore more repetitions of α separated by `T`
$\alpha\oplus$`T`	one ore more repetitions of α separated by `T`

2 The language SL

The grammatical structure of SL is shown in Fig. 1 with the lexical definitions in Fig. 2. We illustrate the different phrases by example.

TODO

Lexic anpassen (Operatoren)

2.1 Predefined functions and types

SL has five predefined types: `Int`, `Char`, and `String`, integers, characters, and strings respectively, as well as the type of the built-in JavaScript-quoting monad `DOM` and the unit type `Void`.

```
Program     ::=  Def+
Def         ::=  FUN var : Type
              |  DEF var PPat* = Expr
              |  DATA type var* = (cons Type*)⊕|
PPat        ::=  var | cons | ( cons PPat* )
Type        ::=  var | type | ( type Type* )
              |  Type -> Type⊕-> | (Type -> Type)
Expr        ::=  IF Expr THEN Expr ELSE Expr
              |  \ PPat+ . Expr
              |  CASE Expr Alt+
              |  LET LocalDef+ IN Expr
              |  Expr Expr
              |  ( Expr )
              |  Expr binop Expr
              |  JSQuote
              |  var | cons | [-] num | char | string
JSQuote     ::=  {| string |} [: Type]
LocalDef    ::=  var = Expr
Alt         ::=  OF Pat THEN Expr
Pat         ::=  var | cons | cons PPat+
```

Grammar 1: The grammar of SL.

```
binop       ::=  + | - | * | /
              |  < | <= | == | /= | >= | >
var         ::=  lowercase alphanum*
cons        ::=  uppercase alphanum*
type        ::=  uppercase alphanum*
lowercase   ::=  a | ··· | z
uppercase   ::=  A | ··· | Z
digit       ::=  0 | ··· | 9
alphanum    ::=  lowercase | uppercase | digit
num         ::=  digit+
char        ::=  ' character '
string      ::=  " character* "
```

Grammar 2: Lexical structure of SL.

To manipulate predefined types SL has the following predefined functions:

```
+   : Int -> Int -> Int
-   : Int -> Int -> Int
*   : Int -> Int -> Int
/   : Int -> Int -> Int
<   : Int -> Int -> Bool
<=  : Int -> Int -> Bool
==  : Int -> Int -> Bool
/=  : Int -> Int -> Bool
>=  : Int -> Int -> Bool
ord : Char -> Int
chr : Int -> Char
```

JavaScript quotes run inside the `DOM` monad and can be combined using the standard bind operators. `yield` lifts SL values into the `DOM` monad.

```
yield : a -> DOM a
&=    : DOM a -> (a -> DOM b) -> DOM b
&     : DOM a -> DOM b -> DOM b
```

In contrast to built-in types, `Bool` is not predefined but belongs to the standard prelude.

For the two function `ord` and `chr` that convert among integer and characters the following identities holds:

```
ord (chr n) = n
chr (ord c) = c
```

2.2 Top-level definitions

An SL program consists of a sequence of data type definitions, function signatures, and function definitions that may appear in any order.

2.2.1 Data type definitions.

A data type definition introduces a type name and one or more data constructors. Data types may be parametric, the type parameters have to be listed after the type name. We can define the type of polymorphic lists with the following piece of code:

```
DATA List a = Cons a (List a) | Nil
```

Note that data constructors and type names always start with a capital letter whereas type parameters start with small letters.

To demonstrate the use of more than one type parameter, we define pairs and triples:

```
DATA P2 a b = P2 a b

DATA P3 a b c = P3 a b c
```

There are the following context restrictions on data type definitions:

- All names of types in a program must be disjoint.
- All type parameters in a data type definition must be disjoint. nn
- All data constructors in a program must be disjoint (not just all data constructors of a particular type).
- No undeclared type parameters may be used on the righ-hand side of a definition.
- All uses of type names on the right-hand side of a definition must be applied to the correct number of type arguments. For example, given the previous definition of lists, the following n-ary tree is invalid:

  ```
  DATA Tree a = Tree a List
  ```

  Instead, `List` must be applied to a type argument:

  ```
  DATA Tree a = Tree a (List (Tree a))
  ```

2.2.2 Function definitions

Top-level function definitions are pattern based and consist of one or more consequent clauses. The clauses of a function f must not be interrupted by clauses of a different function g nor may any function have the name of a predefined function of Sec. 2.1.

The clauses of a function are tried in top-down order until the first matching pattern is found (first-fit pattern matching).

In patterns only variables or constructors introduced in data type definition are allowed. Furthermore, variables and function names have to start with a small letter to easily distinguish them from data constructors.

As an example, we define the well-known map- and foldr1-function on lists (using the list data type from the previous section):

```
DEF map f Nil         = Nil
DEF map f (Cons x xs) = Cons (f x) (map f xs)

DEF foldr1 op (Cons x Nil) = x
DEF foldr1 op (Cons x xs)  = op x (foldr1 op xs)
```

2.2.3 User-defined operators

In addition to regular function definitions the programmer can define custom, binary operators. An operator definition is a regular function definition where the operator name is stated infix. Thus a custom operator `%` for a modulo function might be defined the following way:

```
DEF a % n = a - (n * (a / n))
```

2.2.4 Function signatures

Note that we do not have to write down types for functions, the SL front-end is able to infer the most general type for each function — if it is type correct at all. The type of the `map` definitions defined previously is

```
map : (a -> b) -> List a -> List b
```

Nevertheless, the programmer can still provide a type signature for each top-level function definition, i. e., to specialize a function according to her own choice. Given the `foldr1` function defined earlier, the programer can explicitly give a typing for this function by providing a corresponding signature for the function definitions:

```
FUN foldr1 : (a -> a -> a) -> List a -> a
```

2.3 Expressions

A function body, i. e., the right hand side of a function definition, consists of an expression, which may use the variables introduced on the left-hand side of the definition and all names of top-level function definitions.

Function application Function application of a function f to an argument a is written as juxtaposition of f and a without parentheses: $f\ a$. This makes the partial application of functions particularly comfortable.

For example, we can write the increment function like this:

```
DEF add x y = x + y
DEF inc = add 1
```

Note that the grammar of SL does not allow partial application of predefined infix functions, hence the definition of `add`.

Conditionals SL has to kinds of conditionals: if- and case-expressions.

The condition in an if-expression must be of type `Bool` (defined in the prelude). The mutually recursive pair of `even` and `odd` uses an if-expression:

```
-- n must be non-negative.
DEF even n = IF n == 0 THEN True ELSE odd (n-1)
DEF odd n  = IF n == 1 THEN True ELSE even (n-1)
```

Case-expressions perform pattern-matching for a single expression, i. e., we can rewrite the `map` function into a single clause using a case-expression:

```
DEF map f l = CASE l
                OF Nil       THEN Nil
                OF Cons x xs THEN Cons (f x) (map f xs)
```

Similar to pattern-based top-level definitions, pattern matching uses a top-down first-fit strategy.

Using the prelude definition of `Bool`

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

an expression `IF c THEN t ELSE e` is equivalent to

```
CASE c
  OF True  THEN t
  OF False THEN e
```

λ-abstractions A λ-abstraction introduces an anonymous function. Like in top-level definitions, pattern matching is used for the arguments. Note, however, that program evaluation will fail if a pattern in a λ-abstraction does not match, since there are no alternatives to try.

An anonymous function taking a two element list of integers returning the maximum of the two can be written as:

```
\ (Cons x (Cons y Nil)) . IF x >= y THEN x ELSE y
```

The application of the above λ-abstraction to a one or three element list will fail of course.

Local definitions An expression may contain local definitions using a let-expression, e. g.:

```
LET gt5 = \ x . x > 5
IN map gt5 list
```

It is allowed to introduce more than one local definition, provided the names introduced in the left-hand sides are disjoint, e. g.

```
LET even = \ n . IF n == 0 THEN True ELSE odd (n-1)
    odd  = \ n . IF n == 1 THEN True ELSE even (n-1)
IN even m
```

As we can see in the last example, the names introduced in a let-expression may be used in the right-hand sides, even mutually recursive.

There is, however, an important restriction for SL:

Restriction

In a set of mutually recursive definitions, all right-hand sides must be λ-expressions.

This restriction includes the special case of a single recursive definition, i. e., the following definition is allowed

```
LET sum = \ n . IF n == 0 THEN 0 ELSE n + sum (n-1)
IN sum 10
```

whereas

```
LET ones = Cons 1 ones
    head = \ (Cons x xs) . x
IN head ones
```

or

```
LET f = g 1 + 1
    g = \ n . IF x == 1 THEN 1 ELSE f + x
IN f
```

are not. The reason is that the above expressions will never terminate using SL's eager evaluation strategy.

JavaScript quotes SL allows the programmer to define JavaScript quotes, i. e., "embedded" JavaScript snippets. JavaScript quotes are SL expressions and run inside the `DOM` monad. Additionally, SL provides a mechanism to reference to SL expressions in the quoted JavaScript code. As an example, let us consider the definition

```
DEF f x y = yield (x + y) &= (\ n. {| alert("Result" + $n) |})
```

A JavaScript quote is defined using the `{|` and `|}` braces. The JavaScript code given in a quote can refer to any SL value visible in the current scope by prefixing its name with a `$` character. In our example, we refer to the pattern variable `n` bound by the λ-abstraction encapsulating the JavaScript quote to show the result of adding `x` and `y` to the user with the help of a dialog window.

Since JavaScript quotes run inside the built-in `DOM` monad, a plain JavaScript quote always has the type `DOM Void`. To be able to observe a value encapsulated by the `DOM` monad the programmer has to ascribe the JavaScript quote with a corresponding type, e.g.,

```
DEF ratio = LET getWindowHeight = {| window.outerHeight |} : DOM Int
                getWindowWidth  = {| window.outerWidth |} : DOM Int
            IN getWindowWidth  &= (\ width.
               getWindowHeight &= (\ height.
               yield (width / height)))
```

In this example, we use JavaScript quotes to access the DOM to determine the current height and width of the browser window. The function `ratio` uses these information to calculate the current ratio of the browser window, the type of this function is `DOM Int`.

Further syntactic rules There are a few additional syntactic rules left out for brevity in the grammar of Fig. 1. We name them here in prose:

- Application associates to the left.
- λ-bodies reach as far to the right as possible, i. e.

  ```
  \ x . map (inc x) (Cons 1 Nil) = \ x . (map (inc x) (Cons 1 Nil))
  ```

  and not

  ```
  \ x . map (inc x) (Cons 1 Nil) = (\ x . map (inc x)) (Cons 1 Nil)
  ```

- The arithmetic operators associate to the left; `*`, `/`, bind tighter than `+`, `-` resp.
- The boolean operators associate to the left.
- The comparison operators are not associative.
- The arithmetic operators have highest precedence, followed by comparison operators, and boolean operators with lowest precedence.
- Function application by juxtaposition has highest priority.

3 Programs in SL

A program in SL consists of a series of data definitions, function definitions, and function signatures and exactly one nullary main function. The main function contains the expression to be evaluated, i. e., the starting point of the program. The type of the main-function must be `DOM Void`.

The standard prelude (cf. next section) is implicitly part of every SL program since types like `List` and `Bool` are required to assign types to certain SL expressions.

4 SL standard prelude

The standard prelude of SL is included in every program (cf. Sec. 3). It is very small and consists only of two data type definitions for lists and booleans (file `Prelude.sl`):

```
-- Standard prelude of SL

DATA Bool = True | False

DATA List a = Nil | Cons a (List a)
```

5 SL compiler

5.1 Building and using the compiler

The SL compiler uses the *Simple Build Tool* (sbt)[1] as its build system.

Running the command `assembly` inside the sbt shell

```
sbt> assembly
```

yields a Java archive containing the bundled SL compiler.

TODO

describe usage of compiler

[1] `http://www.scala-sbt.org/`