Compiling the Spla language

Kelley van Evert, s4046854

June 21, 2014

Contents

1 Introduction

This paper describes the implementation of a compiler for the "Spla" language, in the context of a course on compiler construction given at the Radboud University Nijmegen. The Spla language was designed by the author as an extension of the "Splb" language used in earlier parts of the course, so as to allow it to compile itself.

The compiler is written in Haskell and consist of a parser, built upon the notion of parser combinators, and a code generator, generating assembly for the "Simple Stack Machine" machine emulator by Atze Dijkstra. A type checker was available for the Splb language, implementing the well-known Algorithm W, but time constraints did not allow for its extension to Spla.

2 The Spla language

2.1 Language overview

The Spla language was designed by the author as an extension of the Splb language used in earlier parts of the course, adding *whichever features were deemed neccessary or useful* so as to allow it to compile itself. Where Splb was a straight-forward C-like low-level imperative language, Spla now features the extensions and modifications:

New language constructs

- *Higher-order functions*, as values, allowing for the programming of constructs such as parser combinators, etc.
- *Nested function definitions*, where functions are expressed as values, allowing for easier programming.
- *Algebraic data types and pattern matching*, allowing for complex data structures and operations, typically useful for the definition and usage of abstract syntax trees, etc.
- *Type aliases*.
- *Strings*, though through the simple and inefficient hack of treating the string type as an alias for lists of integers, where each integer denotes the corresponding ASCII character.

Semantic differences from Splb

- Equality is now structural equality, where Splb treated list/tuple equality by-reference.

Oddities

As such the language has become a weird, and frankly somewhat uninformed, hybrid of the functional en imperative paradigms. The following key points explain this:

- Strict evaluation, allowing for stately imperative programs to be written.
- Aggregate data types (lists, tuples, and algebraic data types) are passed by reference, though equality is checked structurally.
- Functions are expressed and passed around as values, allowing for complex functional programs to be written, though partial function application is not allowed.

2.2 Safety

Spla is type-safe, and the only dynamic exception that can occur is that of accessing head or tail segments of empty lists.

2.3 Formal specification

The following ENBF-inspired grammar specifies syntactically valid Spla programs. Spla syntax is written in bold monotype, and regular expressions are used for convenience.

Formal grammar of Spla

Unary operator	$\& ::=$	`!` | `-`
Binary operator	$\otimes ::=$	`+` | `-` | `*` | `/` | `%` | `==` | `!=` | `>=` | `<=` | `>` | `<` | `&&` | `||`
Integer	$n ::=$	$[$`0-9`$]^+$
Boolean	$b ::=$	`true` | `false`
Identifier	$x, y ::=$	$[$`a-z`$][$`_a-z0-9'`$]^*$
Constructor	$c ::=$	$[$`A-Z`$][$`_a-z0-9'`$]^*$
Type	$\tau ::=$	`'`x | `unit` | `bool` | `int` | `[` τ `]` | `(` τ `,` τ `)` | τ `->` τ
	|	c | c `(` τ $($`,` $\tau)^*$ `)`
Polymorphic type	$\sigma ::=$	τ | `forall` x $(\sqcup x)^*$`.` σ
Literal	$\ell ::=$	b | n | `[]` | `()`
Expression	$E ::=$	a | d | f | ℓ | $E \otimes E$ | $\&E$
	|	E `:` E
	|	`(` E `,` E `)`
	|	`(` E `)`
	|	`fun (` $(x$ $(,$ $x)^*)?$ `)` B
	|	`let` x `=` E `in` E *included for Hindley-Milner typing*
	|	`match` E `{` $R_{Expr}{}^*$ `}`
Match rule	$R_{Expr} ::=$	`|` $E_{Pattern}$ `->` E
Pattern	$E_{Pattern} ::=$	x | d | ℓ
	|	E `:` E
	|	`(` E `,` E `)`
	|	`(` E `)`
Function call	$f ::=$	x `(` $(E$ $(,$ $E)^*)?$ `)`
Data	$d ::=$	c | c `(` E $(,$ $E)^*$ `)`
Field	$h ::=$	`hd` | `tl` | `fst` | `snd`
Access	$a ::=$	x $[$`.` $h]^*$
Block	$B ::=$	`{` S^* `}`
Statement	$S ::=$	`skip;`
	|	`if (` E `)` B $($`else` $B)?$
	|	`while (` E `)` B
	|	a `=` E `;`
	|	f `;`
	|	`return` $E?$ `;`
	|	B
	|	`let` x `=` S `in` S *included for Hindley-Milner typing*
	|	`match` E `{` $R_{Stmt}{}^*$ `}`
Match rule	$R_{Stmt} ::=$	`|` $E_{Pattern}$ `->` S
Program	$P ::=$	F^*

Semantically valid Spla programs are those that are syntactically valid, satisfy certain static criteria, see Subsection 4.2, and are found valid by the type system expounded in Subsection 4.4.

3 Notation, terminology and conventions

3.1 Abstract concrete grammar

Although the above specified grammar of Spla is its concrete grammar, we will use it throughout this paper as its abstract syntax tree as well, to facilitate reading, taking into account the following points:

- We omit a few right-hand side clauses from the abstract syntax tree, by performing the rewriting rules listed below, where x_{fresh} stands for some freshly chosen identifier.

$$\begin{aligned}
\texttt{(}E\texttt{)} &\longmapsto E \\
\texttt{if (}E\texttt{) then }B &\longmapsto \texttt{if (}E\texttt{) then }B\texttt{ else \{ skip; \}} \\
\texttt{return;} &\longmapsto \texttt{return ();} \\
\tau\ x\ \texttt{()}\ B &\longmapsto \tau\ x\ \texttt{(unit }x_{\text{fresh}}\texttt{)}\ B \\
c &\longmapsto c\texttt{()}
\end{aligned}$$

- When accounting for some piece of abstract syntax containing a sequence of similar segments, we sometimes tacitly omit separating or intermediate syntax, as in the following examples:

$$\begin{aligned}
&\tau\ x\ \texttt{(}\tau_1\ x_1\texttt{,}\ \tau_2\ x_2\texttt{,}\ \tau_3\ x_3\texttt{,}\ \tau_4\ x_4\texttt{)}\ B \text{ e.g. written as } \tau\ x\ \texttt{(}\tau_1\ x_1\texttt{,}\ \ldots\tau_4\ x_4\texttt{)}\ B \\
&x\texttt{(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{,}\ E_4\texttt{)}\ B \text{ e.g. written as } x\texttt{(}E_1 \ldots E_4\texttt{)} \\
&\texttt{\{}\tau_1\ x_1\texttt{;}\ \tau_2\ x_2\texttt{;}\ \tau_3\ x_3\texttt{;}\ S_1\ S_2\ S_3\texttt{\}} \text{ e.g. written as } \texttt{\{}\tau_1\ x_1\texttt{;}\ \ldots\tau_3\ x_3\texttt{;}\ \overline{S}\texttt{\}}
\end{aligned}$$

- Field access dot notation is left associative, e.g. $x\texttt{.}d_1\texttt{.}d_2\texttt{.}d_3$ stands for $(((x\texttt{.}d_1)\texttt{.}d_2)\texttt{.}d_3)$.
- Usual binding powers apply to binary and unary operations in expressions.

3.2 Assembly signatures

We denote the effect of arbitrary pieces of assembly code on the stack by assigning it an *assembly signature*. An assembly signature has the shape n `->` m, and means that the given bit a assembly uses the top n elements of the stack as arguments, and results in the new top m elements of the stack. (The stack size now being positively changed by $m - n$.) If A is a piece of assembly, we write A `::` n `->` m.

3.3 Terminology

Access (variable) An access variable is the *normal* kind of variable used in Spla code, and *access* stands for the act of accessing a value it implies. It is mainly terminology used to contrast with the notion of a *capturing variable*.

Capturing variable A capturing variable is a variable in a match pattern (expression), which therefore is not an access variable, but declares a (captured) local in the lexical scope of its match rule if the match rule applies.

Lexical scope/context A lexical scope is a textual range in Spla code in which a variable lives. A lexical context is an execution environment in which variables of a lexical scope are assigned values.

4 Semantics

4.1 Lexical scoping

Spla features lexical scoping through the following language constructs:

- global scope;
- let construct;
- block;
- function body;
- match rule;

Function an variable shadowing is allowed, but constructor shadowing is not. Variables are visible everywhere in their declaring lexical scope, though may not be accessed until declared.

It must be noted that the notion of a *lexical context* is different than that of a *lexical scope*. A lexical scope is a textual range in Spla code in which a variable lives. A lexical context is an execution environment in which variables of a lexical scope are assigned values. Lexical contexts are created at runtime whenever execution enters a lexical scope or a function is called.

4.2 Static check

Aside from being well-typed, a syntactically valid program must first satisfy the following (statically checkable) criteria to be found a valid Spla program. Parenthesized items are not criteria, but remarks.

Type declarations

A declared type is either a type alias or an ADT.

- (Order of type declarations does not matter.)
- Type aliases declarations may not be circular.
- All constructors must have different names.
- All declared types must have different names.
- Declared types must not have free type variables (that is, free type variables must be captured as type arguments to the declared type).
- A constructor can be used as a function, but otherwise a data expression must have the right amount of arguments, i.e. partial application does not exist.

Lexical issues

- All variables declared in a lexical scope must have different names.
- (Shadowing variables or functions in outer lexical scopes is allowed.)
- Shadowing constructors is not allowed.
- In a given lexical scope, access to a variable may only occur *after* its declaration. This also means that if the variable is not yet declared, but another variable with the same name is defined in an outer scope, the access does not reach out to the other variable, but is invalid.

Match construct

- An expression match must cover all cases. (Whereas a statement match need not.)
- (As each match rule defines a new lexical scope, and capturing variables are not access variables but define locals in this lexical scope, they must all have different names.)

Returning

- Each function must have a returning statement. A returning statement is a return statement, or a statement that always returns. E.g., an if statement always returns iff both branches always return, etc.

Miscellaneous

- Each program must contain a function **`main`** of type **`unit -> int`**.

4.3 Dynamic exceptions

Note that above stated criteria do not guarantee dynamic safety. The only dynamic exception that can occur in Spla is that of access or storage of **`hd`** or **`tl`** segments of empty lists. Program will then jump to the **`_EXCEPTION_EmptyList`** label and halt.

Moreover, heap storage is as of yet not garbage collected.

4.4 Typing system

Spla features a Hindley-Milner polymorphic type system. The possible function types are those admitted by the grammar as a *(Polynorphic type)* of the form $\sigma =$ **`forall`** $\ldots$ **`.`** τ_1 **`->`** τ_2. These are not included in the concrete grammar of Spla, instead they are implicitly generalized as in, for example, Haskell. This means that a function x, concretely defined as

$$\tau_{n+1}\ \ x\ \ \texttt{(}\tau_1\ \ x_1\texttt{,}\ \ \ldots \tau_n\ \ x_n\texttt{)}\ \ B$$

is implicitly regarded as having the $\lambda 2$ type signature

$$x : \texttt{forall}\ \ y_1 \ldots y_k\texttt{.}\ \ \tau_1\ \ \texttt{->}\ \ \ldots\ \ \texttt{->}\ \ \tau_n\ \ \texttt{->}\ \ \tau_{n+1}$$

where

$$\bigcup_{j=1}^{n+1} \mathrm{fv}(\tau_j) = \{y_1 \ldots y_k\}.$$

The type system of Spla is pretty much exactly the same as that of Splb. As we have not yet completed its implementation, we haven't included any specifics of it in this document. Refer to the Splb document for the general idea.

5 Implementation notes

5.1 Code infrastructure

Code is divided into groups of modules as follows.

Language

`Spla.Language` Describes language in terms of syntax fragments, AST, operators, lexing tokens, type theory, and basic as well as general operations on these data.

Parser

`ParserCombinators` A basic parser combinator library.

`Spla.Lexer` Defines **`splalex :: String -> [Tokens]`**.

`Spla.Parser` Defines **`splaparse :: [Tokens] -> Program`**.

Static analysis

`Spla.TypeCheck` Defines the function **`staticCheck :: Program -> Program`**, which enforces static semantic criteria on syntactically valid Spla programs, as well as the function **`typecheck :: Program -> Program`**, which does actual type checking (and inferring). Note that this module is not completed, due to time constraints, but can, in principle, be easily extended from the static analysis module of Spla.

Code generation

`Spla.SSMCompile` Defines **`compileP :: Program -> Asm`**.

All groups depend on the language group, through for the rest they are fully independent.

5.2 Coding patterns

5.2.1 Traits

The AST is defined with ADTs such as **`Program`**, **`Stmt`** and **`Expr`**. As the AST is often traversed to perform a specific action, such as compiling or checking for free variables, we have often used what we call the "trait" pattern, where a Haskell class defines a specific trait (compileable, having lexical meaning, etc), and applicable ADTs implement this class. As such we have, for example, the following traits:

- To deal with compilation (implemented by **`Program`**, **`Stmt`**, **`Expr`**, **`Let`**, **`Match`** and **`Lit`**)

```
class Compileable a where
  compile :: String -> a -> Compiler Asm
```

- To deal with lexical issues (implemented by **`Access`**, **`Stmt`**, **`Expr`**, **`Let`**, **`Match`**, **`MatchRule`** and **`FunCall`**)

```
class LexicalVars a where
  fv :: a -> Set String
  subst :: Subst -> a -> a
```

5.2.2 Alternative compilation schemes

Instead of defining a separate ADT, we decided to use the normal expression ADT to represent match pattern expressions. This approach avoids a lot of (messy) code duplication in the compiler (which we would have had by defining a new ADT, or boxing), but also introduces an obvious problem with regard to the "trait" approach explained above. Therefore we adopted an approach in which we pass a flag to the **`compile`** function indicating what alternative compilation scheme should be used. For example, the expression ADT has three different compilation schemes:

Normal Compiles expressions as values, as usual.

Match pattern expression Allows compiling match wildcards (_) as well, and doesn't compile identifiers (as they are now to be regarded as capture variables instead of access variables).

Match check expression Regards given expression as a match pattern expression. Instead of compiling to assembly producing a single element on the heap, it compiles to assembly that compares the top two elements on the heap via its input, capturing expressions in the current match rule lexical context when encountering capture variables, and producing a boolean answer on the stack.

As an illustration of this approach, here is an outline of the implementation of the **Compileable** trait by the **Expr** ADT:

```
instance Compileable Expr where
  -- don't compile as access variable, instead just load dummy value
  compile "match_expr" (E_Access (Ident id))
  -- allow this compilation, just load dummy value
  compile "match_expr" (E_MatchWildcard)

  compile "match_check" (E_MatchWildcard) -- always true
  compile "match_check" (E_Lit l)
  compile "match_check" (E_Data (Data cName args))
  compile "match_check" (E_Tuple e1 e2)
  compile "match_check" (E_BinOp ":" e1 e2)
  compile "match_check" (E_Access (Ident x)) -- capture

  -- normal expression compilation...
```