Compiling the Splb language

Kelley van Evert, s4046854

May 23, 2014

Contents

1 The Splb language

1.1 Language overview

The Splb language is a simple, C-like, imperative language. It's primary feature is a Hindley-Milner typing system, allowing for type-safe definition of mutually recursive functions and polymorphic functions. It also features block-level lexical scoping, has integer and Boolean primitive types, and composed tuple and list types by reference.

1.2 Safety

Splb is a mostly statically type-safe language. Dynamic expceptions still exist, and include the use of uninitialized variables and the access or storage of head/tail segments of empty lists.

1.3 Formal specification

The following ENBF-inspired grammar specifies syntactically valid Splb programs. Every program is required to have a `main` function of type `unit -> int`.

Unary operator $-$ ::= `"!"` | `"-"` | `"~"`

Binary operator $\otimes$ ::= `"+"` | `"-"` | `"*"` | `"/"` | `"%"`
| `"=="` | `"!="`
| `">="` | `"<="` | `">"` | `"<"`
| `"&&"` | `"||"`
| `":"`

Integer n ::= [0-9]+

Boolean b ::= `"true"` | `"false"`

Identifier i ::= [a-z][_a-z0-9']*

Parameter p ::= t i

Type τ ::= i | `"bool"` | `"int"` | `"["` τ `"]"` | `"("` τ `","` τ `")"`

Variable declaration V ::= t i `";"`

Function declaration F ::= t i `"("` (p (`","` p)*)? `")"` B

Literal ℓ ::= b | n | `"[]"` | `"()"`

Expression E ::= a | f | ℓ | $E \otimes E$ | $-E$
| `"("` E `","` E `")"`
| `"("` E `")"`
| `"let"` v `":="` E `"in"` E

Function call f ::= i `"("` (E (`","` E)*)? `")"`

Field d ::= `"hd"` | `"tl"` | `"fst"` | `"snd"`

Access a ::= i [`"."` d]*

Block B ::= `"{"` V* S* `"}"`

Statement S ::= `"skip;"`
| `"if"` `"("` E `")"` B (`"else"` B)?
| `"while"` `"("` E `")"` B
| a `":="` E `";"`
| f `";"`
| `"return"` E? `";"`
| B

Program P ::= F*

2 Semantics

2.1 Typing system

Splb features a Hindley-Milner polymorphic type system.

2.1.1 Expressions

Expressions are quite straightforward. The environment Γ includes local variables, let-introduced variables, language builtin functions and user-defined functions.

$$\text{(int)}\quad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{(bool)}\quad \frac{}{\Gamma \vdash b : \texttt{bool}}$$

$$\text{(unit)}\quad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}} \qquad \text{(emptylist)}\quad \frac{}{\Gamma \vdash \texttt{[]} : \texttt{[}\tau\texttt{]}}$$

$$\text{(hd)}\quad \frac{\Gamma \vdash a : \texttt{[}\tau\texttt{]}}{\Gamma \vdash a\texttt{.hd} : \tau} \qquad \text{(tl)}\quad \frac{\Gamma \vdash a : \texttt{[}\tau\texttt{]}}{\Gamma \vdash a\texttt{.tl} : \texttt{[}\tau\texttt{]}}$$

$$\text{(fst)}\quad \frac{\Gamma \vdash a : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}}{\Gamma \vdash a\texttt{.fst} : \tau_1} \qquad \text{(snd)}\quad \frac{\Gamma \vdash a : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}}{\Gamma \vdash a\texttt{.snd} : \tau_2}$$

$$\text{(unop)}\quad \frac{\Gamma \vdash - : \tau_1 \to \tau_2 \qquad \Gamma \vdash E : \tau_1}{\Gamma \vdash -E : \tau_2}$$

$$\text{(var)}\quad \frac{}{\Gamma, i : \tau \vdash i : \tau} \qquad \text{(binop)}\quad \frac{\Gamma \vdash \otimes : \tau_1 \to \tau_2 \to \tau_3 \qquad \Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \otimes E_2 : \tau_3}$$

$$\text{(funcall)}\quad \frac{\Gamma \vdash i : \tau_1 \to \cdots \to \tau_n \to \tau_{\text{res}} \qquad \forall_{i=1\ldots n} : \ \Gamma \vdash E_i : \tau_i}{\Gamma \vdash i(E_1 \ldots E_n) : \tau_{\text{res}}}$$

$$\text{(tuple)}\quad \frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash \texttt{(}E_1\texttt{,}E_2\texttt{)} : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}} \qquad \text{(let)}\quad \frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma, i : \tau_1 \vdash E_2 : \tau_2}{\Gamma \vdash \{\texttt{let } i \texttt{ := } E_1 \texttt{ in } E_2\} : \tau_2}$$

2.1.2 Statements and blocks

Typing of statements (and blocks) proceeds with two types of judgements.

- The first, $\overline{\Gamma \vdash \square \text{ OK}[\tau]}$, tells us that, given environment Γ, the block/statement $\square$ returns a result of type τ (if at all). These inferences are listed below.
- The second, $\overline{\Gamma \vdash \square \text{ returns}}$, tells us that block/statement $\square$ will definitively return. These inferences are quite straightforward, and have been ommited below. All statements definitively return a value except the `while` statement and `if` statement without an `else` clause. A block definitively returns a value if at least one of its enclosed statements definitively returns a value.

$$\text{(blockstmt)} \quad \frac{\Gamma \vdash \text{block } B \text{ OK}[\tau]}{\Gamma \vdash \text{stmt } B \text{ OK}[\tau]} \qquad \text{(while)} \quad \frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash \text{block } B \text{ OK}[\tau]}{\Gamma \vdash \text{stmt } \{\texttt{while (}E\texttt{)} \ B\} \text{ OK}[\tau]}$$

$$\text{(skip)} \quad \frac{}{\Gamma \vdash \text{stmt } \texttt{skip} \text{ OK}[\tau]} \qquad \text{(if}_1\text{)} \quad \frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash \text{block } B \text{ OK}[\tau]}{\Gamma \vdash \text{stmt } \{\texttt{if (}E\texttt{)} \ B\} \text{ OK}[\tau]}$$

$$\text{(if}_2\text{)} \quad \frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash \text{block } B_1 \text{ OK}[\tau] \qquad \Gamma \vdash \text{block } B_2 \text{ OK}[\tau]}{\Gamma \vdash \text{stmt } \{\texttt{if (}E\texttt{)} \ B_1 \ \texttt{else} \ B_2\} \text{ OK}[\tau]}$$

$$\text{(return}_1\text{)} \quad \frac{}{\Gamma \vdash \text{stmt } \{\texttt{return}\} \text{ OK}[\tau]} \qquad \text{(return}_2\text{)} \quad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \text{stmt } \{\texttt{return} \ E\} \text{ OK}[\tau]}$$

$$\text{(assign)} \quad \frac{\Gamma \vdash a : \tau' \qquad \Gamma \vdash E : \tau'}{\Gamma \vdash \text{stmt } \{a \ \texttt{:=} \ E\} \text{ OK}[\tau]} \qquad \text{(funcall)} \quad \frac{\Gamma \vdash f : \tau'}{\Gamma \vdash \text{stmt } \{f\} \text{ OK}[\tau]}$$

$$\text{(block)} \quad \frac{\forall_{i=1\ldots n} : \ \Gamma, \overline{V} \vdash S_i \text{ OK}[\tau]}{\Gamma \vdash \text{block } \{\overline{V} \ \ S_1 \ldots S_n\} \text{ OK}[\tau]}$$

3 Compilation

3.1 Compilation schemes

The whole program is compiled to:

```
link 0 // enter program
ldc 0 // unit arg for main()
bsr _main // call main()
ldr RR // load main result value from RR
trap 0 // print result
halt
⟨function declarations⟩
⟨language builtins⟩
```

Each function declaration is compiled to:

```
_⟨name⟩:  link ⟨number of locals⟩
⟨body⟩
```

Where the number of locals includes actual locals introduced in blocks as well as let-introduced variables. Return statements end by putting the resulting value in the `RR` return register, and then returning via

```
unlink
return
```

3.2 Calling convention

To call a function ⟨*fn*⟩, first we put its arguments on the stack, then we branch to _⟨*fn*⟩, and when the function returns (via `unlink`, `ret`, as described above) we retrieve the return value from `RR` and put it on the place where the first argument was originally put. This subsequently becomes the new stack pointer.

3.3 Heap / by-reference data types

Lists and tuples are treated by reference. References point to a stack location, or are '0' in case of an empty list.

We implement lists as linked lists, where each node consists of two adjacent heap cells, the first containing the value of the node, the second being the address of the next node (or '0' for the empty list / final node). Tuples are also simply two adjacent heap cells. References to lists and tuples always point to the first of the two cells.

There is currently no garbage management, however runtime exceptions are generated when an emptylist segment is accesses or written to.