Compiling the Splb and Spla languages

Kelley van Evert, s4046854

June 10, 2014

Contents

1 Introduction

This paper describes the implementation of a compiler for the fictional "Splb" language, and for an extension "Spla" adding higher order functions that can be arbitrarily nested.

The compilers, which are written in Haskell, consist of a parser, built upon the notion of parser combinators, a type checker, which implements the well-known Algorithm W for the Hindley-Milner type system, and a code generator, generating assembly for the "Simple Stack Machine" machine emulator.

2 The Splb language

2.1 Language overview

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

2.2 Safety

Splb is a mostly statically type-safe language. The only runtime exceptions that can occur are the use of uninitialized variables and the access or storage of head/tail segments of empty lists.

2.3 Formal specification

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

Formal grammar of Splb

Unary operator	$\&$ ::=	**`!`** $\mid$ **`-`**	*boolean not, resp. integer negation*
Binary operator	$\otimes$ ::=	**`+`** $\mid$ **`-`** $\mid$ **`*`** $\mid$ **`/`** $\mid$ **`%`**	*arithmetic operations*
	$\mid$	**`==`** $\mid$ **`!=`**	*boolean/integer equality*
	$\mid$	**`>=`** $\mid$ **`<=`** $\mid$ **`>`** $\mid$ **`<`**	*integer comparing*
	$\mid$	**`&&`** $\mid$ **`||`**	*boolean operations*
	$\mid$	**`:`**	*list construction*
Integer	n ::=	[**`0-9`**]$^+$	
Boolean	b ::=	**`true`** $\mid$ **`false`**	
Identifier	x, y ::=	[**`a-z`**][**`_a-z0-9'`**]*	
Type	τ ::=	x $\mid$ **`unit`** $\mid$ **`bool`** $\mid$ **`int`** $\mid$ **`[`** τ **`]`** $\mid$ **`(`** τ **`,`** τ **`)`**	
Polymorphic type	σ ::=	τ $\mid$ **`forall`** x $(\textvisiblespace x)^*$**`.`** σ	
Function declaration	F ::=	τ x **`(`** $(\tau\ x\ ($**`,`** $\tau\ x)^*)?$ **`)`** B	
Literal	ℓ ::=	b $\mid$ n $\mid$ **`[]`** $\mid$ **`()`**	***`[]`*** *is the empty list,* ***`()`*** *is unit*
Expression	E ::=	$a \mid f \mid \ell \mid E \otimes E \mid \&E$	
	$\mid$	**`(`** E **`,`** E **`)`**	
	$\mid$	**`(`** E **`)`**	
	$\mid$	**`let`** x **`:=`** E **`in`** E	*included for Hindley-Milner typing*
Function call	f ::=	x **`(`** $(E\ ($**`,`** $E)^*)?$ **`)`**	
Field	d ::=	**`hd`** $\mid$ **`tl`** $\mid$ **`fst`** $\mid$ **`snd`**	
Access	a ::=	x [**`.`** d]*	
Block	B ::=	**`{`** $(\tau\ x$ **`;`**$)^*$ S^* **`}`**	
Statement	S ::=	**`skip;`**	
	$\mid$	**`if`** **`(`** E **`)`** B (**`else`** B)?	
	$\mid$	**`while`** **`(`** E **`)`** B	
	$\mid$	a **`:=`** E **`;`**	
	$\mid$	f **`;`**	
	$\mid$	**`return`** $E?$ **`;`**	
	$\mid$	B	
Program	P ::=	F^*	

3 The Spla language

3.1 Language overview

The Splb language extends Spla by treating functions as first-class values, than can be passed around freely, and defined in nested manners. The compiling complexity thus introduced is that of properly handling lexical scope, and the issue of variables outliving execution of functions in which they were defined.

The goal of this extension was to be able to parse Spla in Spla itself. This might still be achieved, but as of yet is pragmatically infeasible (due to wildly inefficient generated machine code, and slow execution of SSM).

The Spla language, being on the one hand functional enough to handle higher order functions in many manners, but on the other hand not supporting partial application and closely resembling the Splb language, including the C-like syntax, makes it easy to think of as a “little brother” of Javascript. The author is aware that this might not be regarded as a popular choice.

Spla supports string notation, but regards this as syntactical sugar for integer lists, where characters as regarded as their ASCII-codes. The author also slightly extended SSM to include a **`trapchr`** instruction, and Spla includes a **`printchr`** function, to allow for the printing of string output.

3.2 Safety

Spla is as (un)safe as Splb is.

3.3 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	$\& ::= \texttt{!} \mid \texttt{-}$	*boolean not, resp. integer negation*
Binary operator	$\otimes ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	*arithmetic operations*
	$\mid \texttt{==} \mid \texttt{!=}$	*boolean/integer equality*
	$\mid \texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<}$	*integer comparing*
	$\mid \texttt{\&\&} \mid \texttt{\|\|}$	*boolean operations*
	$\mid \texttt{:}$	*list construction*
Integer	$n ::= [\texttt{0-9}]^{+}$	
Boolean	$b ::= \texttt{true} \mid \texttt{false}$	
Identifier	$x, y ::= [\texttt{a-z}][\texttt{\_a-z0-9'}]^{*}$	
Type	$\tau ::= x \mid \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{[}\ \tau\ \texttt{]} \mid \texttt{(}\ \tau\ \texttt{,}\ \tau\ \texttt{)} \mid \tau\ \texttt{->}\ \tau$	
Polymorphic type	$\sigma ::= \tau \mid \texttt{forall}\ x\ (\llcorner x)^{*}\texttt{.}\ \sigma$	
Literal	$\ell ::= b \mid n \mid \texttt{[]} \mid \texttt{()}$	*`[]` is the empty list, `()` is unit*
Expression	$E ::= a \mid f \mid \ell \mid E \otimes E \mid \&E$	
	$\mid \texttt{(}\ E\ \texttt{,}\ E\ \texttt{)}$	
	$\mid \texttt{(}\ E\ \texttt{)}$	
	$\mid \texttt{fun}\ \texttt{(}\ (x\ (\texttt{,}\ x)^{*})?\ \texttt{)}\ B$	
	$\mid \texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ E$	*included for Hindley-Milner typing*
Function call	$f ::= x\ \texttt{(}\ (E\ (\texttt{,}\ E)^{*})?\ \texttt{)}$	
Field	$d ::= \texttt{hd} \mid \texttt{tl} \mid \texttt{fst} \mid \texttt{snd}$	
Access	$a ::= x\ [\texttt{.}\ d]^{*}$	
Block	$B ::= \texttt{\{}\ S^{*}\ \texttt{\}}$	
Statement	$S ::= \texttt{skip;}$	
	$\mid \texttt{if}\ \texttt{(}\ E\ \texttt{)}\ B\ (\texttt{else}\ B)?$	
	$\mid \texttt{while}\ \texttt{(}\ E\ \texttt{)}\ B$	
	$\mid a\ \texttt{=}\ E\ \texttt{;}$	
	$\mid f\ \texttt{;}$	
	$\mid \texttt{return}\ E?\ \texttt{;}$	
	$\mid \mathrm{B}$	
Program	$P ::= F^{*}$	

4 Notation and conventions

Although the above specified grammar of Splb 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.

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

- 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:

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

- 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.

5 Semantics

5.1 Valid programs

Valid programs are those that:

- are syntactically valid as by the above stated grammar;
- contain a function **`main`** of type **`unit -> int`** (a criterion that is included in the type system);
- are found valid by the Splb typing system.

5.2 Dynamic exceptions

Note that above stated criteria do not guarantee dynamic safety. The two dynamic exceptions that can occur are as follows, and will be caught at runtime by jumping to respective exception labels.

- Use of uninitialized variables. Program will then jump to the **`_EXCEPTION_UninitiatedVar`** label and halt.
- 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, so it can happen that the heap overflows. What happens then is left unspecified.

5.3 Variable lifetime and lexical scoping

Variables live in blocks and let-expressions, and their lifetime ends when the block/let computations end. As a corollary, variables never outlive a function computation.

Splb features lexical scoping through variables introduced in blocks and let expressions. Function shadowing, as well as variable shadowing, is allowed. Variables are declared at the beginning of each block to avoid further complications. (It is not neccessary to make decisions as to what some variable x might refer to, if that variable were to be declared later on in the same block, as such declarations are not permitted.)

5.4 Typing system

Splb features a Hindley-Milner polymorphic type system. The possible function types are those admitted by the grammar as a *(Polynorphic type)* of the form $\sigma = \texttt{forall} \ \dots \texttt{.} \ \tau_1 \ \texttt{->} \ \tau_2$. These are not included in the concrete grammar of Splb, instead they are implicitly generalized as in, for example, Haskell. This means that a function x, concretely defined as

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

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

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

where

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

For the rest of this section, Γ will serve as an environment of the form $x_1 : \sigma_1, \dots x_n : \sigma_n$, including local variables, let-introduced variables, user-defined functions and the following language builtin functions and operators:

$$\begin{aligned}
\texttt{isEmpty} &: \texttt{forall} \ x\texttt{.} \ \texttt{[}x\texttt{]} \ \texttt{->} \ \texttt{bool} \\
\texttt{print} &: \texttt{forall} \ x\texttt{.} \ x \ \texttt{->} \ \texttt{unit}
\end{aligned}$$

$$\begin{aligned}
\texttt{!} &: \texttt{bool -> bool} \\
\texttt{-} &: \texttt{int -> int}
\end{aligned}$$

$$\begin{aligned}
\texttt{+} &: \texttt{int -> int -> int} & \texttt{-} &: \texttt{int -> int -> int} \\
\texttt{*} &: \texttt{int -> int -> int} & \texttt{/} &: \texttt{int -> int -> int} \\
\texttt{\%} &: \texttt{int -> int -> int}
\end{aligned}$$

$$\begin{aligned}
\texttt{==} &: \texttt{int -> int -> bool} & \texttt{!=} &: \texttt{int -> int -> bool} \\
\texttt{<=} &: \texttt{int -> int -> bool} & \texttt{>=} &: \texttt{int -> int -> bool} \\
\texttt{<} &: \texttt{int -> int -> bool} & \texttt{>} &: \texttt{int -> int -> bool}
\end{aligned}$$

$$\begin{aligned}
\texttt{\&\&} &: \texttt{bool -> bool -> bool} \\
\texttt{||} &: \texttt{bool -> bool -> bool}
\end{aligned}$$

$$\texttt{:} : \texttt{forall} \ x\texttt{.} \ x \ \texttt{->} \ \texttt{[}x\texttt{]} \ \texttt{->} \ \texttt{[}x\texttt{]}$$

5.4.1 Expressions

The binary relation ($\sqsubseteq$) on polymorphic types expresses when one type is more general than another, e.g. $\sigma' \sqsubseteq \sigma$ states that σ' is more general than σ.

Expression typing judgements

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

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

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

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

$$\text{(var)}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \text{(unop)}\ \frac{\Gamma \vdash \& : \tau_1 \to \tau_2 \qquad \Gamma \vdash E : \tau_1}{\Gamma \vdash \&E : \tau_2}$$

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

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

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

$$\text{(instantiate)}\ \frac{\Gamma \vdash E : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash E : \sigma} \qquad \text{(generalize)}\ \frac{\Gamma \vdash E : \sigma \qquad x \notin \text{fv}(\Gamma)}{\Gamma \vdash E : \texttt{\textbf{forall}}\ x\texttt{\textbf{.}}\ \ \sigma}$$

5.4.2 Statements and blocks

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

- The first, $\overline{\Gamma \vdash \text{block/stmt } \{\,\square\,\} \text{ OK}[\tau]}$, 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, $\overline{\Gamma \vdash \text{block/stmt } \{\,\square\,\} \text{ ret}}$, tells us that block/statement $\square$ will definitively return.

Block/stmt OK

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

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

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

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

$$\text{(block)}\ \frac{\forall i \in \{1 \dots n\} :\ \Gamma, x_1 : \tau_1, \dots x_k : \tau_k \vdash \text{stmt}\ \{S_i\}\ \text{OK}[\tau]}{\Gamma \vdash \text{block}\ \{\texttt{\{}\tau_1\ \ x_1 \dots \tau_k\ \ x_k\texttt{;}\ \ S_1 \dots S_n\texttt{\}}\}\ \text{OK}[\tau]}$$

Block/stmt returning

$$\frac{\vdash \text{block}\ \{B_1\}\ \text{ret}}{\vdash \text{stmt}\ \{\texttt{if (}E\texttt{)}\ B_1\ \texttt{else}\ B_2\}\ \text{ret}} \qquad \frac{\vdash \text{block}\ \{B_2\}\ \text{ret}}{\vdash \text{stmt}\ \{\texttt{if (}E\texttt{)}\ B_1\ \texttt{else}\ B_2\}\ \text{ret}}$$

$$\frac{\vdash \text{block}\ \{B\}\ \text{ret}}{\vdash \text{stmt}\ \{B\}\ \text{ret}} \qquad \frac{}{\vdash \text{stmt}\ \{\texttt{return}\ E\}\ \text{ret}}$$

$$\frac{\exists j \in \{1 \dots n\} :\ \Gamma' \vdash \text{stmt}\ \{S_i\}\ \text{ret}}{\vdash \text{block}\ \{\texttt{\{}\dots\texttt{;}\ \ S_1 \dots S_n\texttt{\}}\}\ \text{ret}}$$

5.4.3 Wrapping up

Concluding, we type check function declarations and the whole program using the judgements $\dfrac{}{\Gamma \vdash \text{fundecl}\ \{\,F\,\}\ \text{OK}}$ and $\dfrac{}{\vdash \text{program}\ \{\,P\,\}\ \text{OK}}$.

Program/function declaration OK

$$\frac{\Gamma, x_1 : \tau_1 \ldots x_n : \tau_n \vdash \text{block } \{B\} \text{ OK}[\tau] \qquad \Gamma, x_1 : \tau_1 \ldots x_n : \tau_n \vdash \text{block } \{B\} \text{ ret}}{\Gamma \vdash \text{fundecl } \{\tau \;\; x \;\; \texttt{(}\tau_1 \;\; x_1\texttt{,} \;\; \ldots \tau_n \;\; x_n\texttt{)} \;\; B\} \text{ OK}}$$

$$\frac{\exists j \in \{1 \ldots n\} : \; (\texttt{main} : \texttt{unit} \to \texttt{int}) \in \Gamma \qquad \forall j \in \{1 \ldots n\} : \; \Gamma \vdash \text{fundecl } F_j \text{ OK}}{\vdash \text{program } \{F_1 \ldots F_n\} \text{ OK}}$$

$$\textit{where} \quad \Gamma = \begin{array}{l} \langle \textit{Language builtins, as described above} \rangle, \\ \text{ident}(F_1) : \text{sig}(F_1), \\ \vdots \\ \text{ident}(F_n) : \text{sig}(F_n) \end{array}$$

$$\textit{and} \quad \text{ident}(\tau \;\; x \;\; \texttt{(}\ldots\texttt{)} \;\; B) = x$$

$$\textit{and} \quad \text{sig}(\tau_{n+1} \;\; x \;\; \texttt{(}\tau_x \;\; i_1\texttt{,} \;\; \ldots \tau_n \;\; x_n\texttt{)} \;\; B) = \texttt{forall} \;\; y_1 \ldots y_k\texttt{.} \;\; \tau_1 \;\; \texttt{->} \;\; \ldots \;\; \texttt{->} \;\; \tau_{n+1}$$

$$\textit{and} \quad \bigcup_{j=1}^{n+1} \text{fv}(\tau_j) = \{y_1 \ldots y_k\}$$

6 SSM compilation

In this section I explain how Splb code is translated to SSM assembly. SSM (Simple Stack Machine) is a stack machine emulator developed for educational purposes. It is described at `http://www.staff.science.uu.nl/~dijks106/SSM/`.

6.1 Compilation schemes

$$\text{numLocals}_{\text{block}}(V_1 \dots V_k \;\; S_1 \dots S_n) = k + \sum_{i=1}^{n} \text{numLocals}_{\text{stmt}}(S_i)$$

$$\begin{aligned}
\text{numLocals}_{\text{stmt}}(\texttt{skip;}) &= 0 \\
\text{numLocals}_{\text{stmt}}(\texttt{if (}E\texttt{)}\ B_1\ \texttt{else}\ B_2) &= \text{numLocals}_{\text{block}}(B_1) + \text{numLocals}_{\text{block}}(B_2) \\
&\quad + \text{numLocals}_{\text{expr}}(E) \\
\text{numLocals}_{\text{stmt}}(\texttt{while (}E\texttt{)}\ B) &= \text{numLocals}_{\text{block}}(B) + \text{numLocals}_{\text{expr}}(E) \\
\text{numLocals}_{\text{stmt}}(a\ \texttt{:=}\ E\texttt{;}) &= 0 + \text{numLocals}_{\text{expr}}(E) \\
\text{numLocals}_{\text{stmt}}(f\texttt{;}) &= 0 \\
\text{numLocals}_{\text{stmt}}(\texttt{return}\ E\texttt{;}) &= \text{numLocals}_{\text{expr}}(E) \\
\text{numLocals}_{\text{stmt}}(B) &= \text{numLocals}_{\text{block}}(B)
\end{aligned}$$

$$\begin{aligned}
\text{numLocals}_{\text{expr}}(a) &= 0 \\
\text{numLocals}_{\text{expr}}(f) &= 0 \\
\text{numLocals}_{\text{expr}}(\ell) &= 0 \\
\text{numLocals}_{\text{expr}}(E_1 \otimes E_2) &= \text{numLocals}_{\text{expr}}(E_1) + \text{numLocals}_{\text{expr}}(E_2) \\
\text{numLocals}_{\text{expr}}(\&E) &= \text{numLocals}_{\text{expr}}(E) \\
\text{numLocals}_{\text{expr}}(\texttt{(}E_1\texttt{,}E_2\texttt{)}) &= \text{numLocals}_{\text{expr}}(E_1) + \text{numLocals}_{\text{expr}}(E_2) \\
\text{numLocals}_{\text{expr}}(\texttt{let}\ i\ \texttt{:=}\ E_1\ \texttt{in}\ E_2) &= 1 + \text{numLocals}_{\text{expr}}(E_1) + \text{numLocals}_{\text{expr}}(E_2)
\end{aligned}$$

Program/fundecl/funcall

$$[\![F_1 \dots F_n]\!]_{\text{program}} = \begin{array}{l} \texttt{link 0} \\ \texttt{ldc 0} \\ \texttt{bsr \_main} \\ \texttt{ldr RR} \\ \texttt{trap 0} \\ \texttt{halt} \\ [\![F_1]\!]_{\text{fundecl}} \\ \vdots \\ [\![F_n]\!]_{\text{fundecl}} \\ \langle \textit{Language builtins} \rangle \end{array} \qquad \text{(Program)}$$

$$[\![\tau\ \ i\texttt{(}\tau_1\ \ i_1 \dots \tau_n\ \ i_n\texttt{)}\ \ B]\!]_{\text{fundecl}} = \begin{array}{ll} \texttt{\_}i\texttt{:} & \texttt{link}\ 2 \cdot \text{numLocals}(B) \\ & [\![B]\!]^{\Gamma}_{\text{block}} \end{array} \qquad \text{(Function declaration)}$$

$$\textit{where}\ \ \Gamma = [i_j \mapsto -2 - n + j]\ \textit{for}\ j \in \{1 \dots n\}$$

$$[\![i\texttt{(}E_1 \dots E_n\texttt{)}]\!]^{\Gamma}_{\text{funcall}} = \begin{array}{l} [\![E_1]\!]^{\Gamma}_{\text{expr}} \\ \vdots \\ [\![E_n]\!]^{\Gamma}_{\text{expr}} \\ \texttt{bsr \_}i \\ \texttt{ajs}\ -n \end{array} \qquad \text{(Function call)}$$

$$\llbracket \tau_1\ \ i_1\texttt{;}\ \ \dots\ \ \tau_k\ \ i_k\texttt{;}\ \ S_1 \dots S_n \rrbracket^{\Gamma}_{\text{block}} = \begin{array}{l} \llbracket S_1\rrbracket^{\Gamma'}_{\text{stmt}} \\ \vdots \\ \llbracket S_n\rrbracket^{\Gamma'}_{\text{stmt}}\end{array} \qquad \text{(block)}$$

$$\begin{aligned} \textit{where}\ \ \text{locals}(\Gamma) &= \{(i,n) \in \Gamma \mid n \geq 0\} \\ \textit{and}\ \ \Gamma' &= \Gamma[i_j \mapsto 2 \cdot |\text{locals}(\Gamma)| + 2j - 1]\ \textit{for}\ j \in \{1 \dots k\} \end{aligned}$$

$$\llbracket \texttt{skip;} \rrbracket^{\Gamma}_{\text{stmt}} = \texttt{ajs 0} \qquad \text{(skip)}$$

$$\llbracket B \rrbracket^{\Gamma}_{\text{stmt}} = \llbracket B \rrbracket^{\Gamma}_{\text{block}} \qquad \text{(blockstmt)}$$

$$\llbracket \texttt{return}\ \ E\texttt{;} \rrbracket^{\Gamma}_{\text{stmt}} = \begin{array}{l} \llbracket E\rrbracket^{\Gamma}_{\text{expr}} \\ \texttt{str RR} \\ \texttt{unlink} \\ \texttt{ret} \end{array} \qquad \text{(return)}$$

$$\llbracket f \rrbracket^{\Gamma}_{\text{stmt}} = \llbracket f \rrbracket^{\Gamma}_{\text{funcall}} \qquad \text{(funcall)}$$

$$\llbracket i\ \ \texttt{:=}\ \ E\texttt{;} \rrbracket^{\Gamma}_{\text{stmt}} = \begin{array}{l} \llbracket E\rrbracket^{\Gamma}_{\text{expr}} \\ \texttt{stl}\ \ \Gamma(i) \\ \texttt{ldc 1} \\ \texttt{stl}\ \ \Gamma(i)+1\ \ \textit{// mark variable as initialized} \end{array} \qquad \text{(assign}_1\text{)}$$

$$\llbracket a\texttt{.}d\ \ \texttt{:=}\ \ E\texttt{;} \rrbracket^{\Gamma}_{\text{stmt}} = \begin{array}{l} \llbracket E\rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket a\rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket d\rrbracket_{\text{fieldstorage}} \end{array} \qquad \text{(assign}_2\text{)}$$

$$\llbracket \texttt{if (}E\texttt{)}\ \ B_1\ \ \texttt{else}\ \ B_2 \rrbracket^{\Gamma}_{\text{stmt}} = \begin{array}{ll} & \llbracket E\rrbracket^{\Gamma}_{\text{expr}} \\ & \texttt{brf}\ \ \underline{\textit{else}} \\ & \llbracket B_1\rrbracket^{\Gamma}_{\text{block}} \\ & \texttt{bra}\ \ \underline{\textit{end}} \\ \underline{\textit{else}}\texttt{:} & \texttt{ajs 0} \\ & \llbracket B_2\rrbracket^{\Gamma}_{\text{block}} \\ \underline{\textit{end}}\texttt{:} & \texttt{ajs 0} \end{array} \qquad \text{(if)}$$

$$\llbracket \texttt{while (}E\texttt{)}\ \ B \rrbracket^{\Gamma}_{\text{stmt}} = \begin{array}{ll} \underline{\textit{while}}\texttt{:} & \texttt{ajs 0} \\ & \llbracket E\rrbracket^{\Gamma}_{\text{expr}} \\ & \texttt{brf}\ \ \underline{\textit{end}} \\ & \llbracket B\rrbracket^{\Gamma}_{\text{block}} \\ & \texttt{bra}\ \ \underline{\textit{while}} \\ \underline{\textit{end}}\texttt{:} & \texttt{ajs 0} \end{array} \qquad \text{(while)}$$

Expression compilation

$$\llbracket n \rrbracket^{\Gamma}_{\text{expr}} = \texttt{ldc}\ n \qquad \text{(int)}$$

$$\llbracket \texttt{\textbf{true}} \rrbracket^{\Gamma}_{\text{expr}} = \texttt{ldc 0xFFFFFFFF} \qquad \text{(true)}$$

$$\llbracket \texttt{\textbf{false}} \rrbracket^{\Gamma}_{\text{expr}} = \texttt{ldc 0} \qquad \text{(false)}$$

$$\llbracket \texttt{\textbf{()}} \rrbracket^{\Gamma}_{\text{expr}} = \texttt{ldc 0} \qquad \text{(unit)}$$

$$\llbracket \texttt{\textbf{[]}} \rrbracket^{\Gamma}_{\text{expr}} = \texttt{ldc 0} \qquad \text{(emptylist)}$$

$$\llbracket \& E \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \llbracket E \rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket \& \rrbracket_{\text{unop}} \end{array} \qquad \text{(unop)}$$

$$\llbracket E_1 \otimes E_2 \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \llbracket E_1 \rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket E_2 \rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket \otimes \rrbracket_{\text{binop}} \end{array} \qquad \text{(binop)}$$

$$\llbracket f \rrbracket^{\Gamma}_{\text{expr}} = \llbracket f \rrbracket^{\Gamma}_{\text{funcall}} \qquad \text{(funcall)}$$

$$\llbracket i \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \texttt{ldl}\ \Gamma(i) + 1 \\ \texttt{brf \_EXCEPTION\_UninitiatedVar} \\ \texttt{ldl}\ \Gamma(i) \end{array} \qquad \text{(access}_1\text{)}$$

$$\llbracket a\texttt{.}d \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \llbracket a \rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket d \rrbracket_{\text{fieldaccess}} \end{array} \qquad \text{(access}_2\text{)}$$

$$\llbracket \texttt{\textbf{(}} E_1 \texttt{\textbf{,}} E_2 \texttt{\textbf{)}} \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \llbracket E_1 \rrbracket^{\Gamma}_{\text{expr}} \\ \llbracket E_2 \rrbracket^{\Gamma}_{\text{expr}} \\ \texttt{ajs -1} \\ \texttt{sth} \\ \texttt{ajs 1} \\ \texttt{sth} \\ \texttt{ajs -1} \end{array} \qquad \text{(tuple)}$$

$$\llbracket \texttt{\textbf{let}}\ i\ \texttt{\textbf{:=}}\ E_1\ \texttt{\textbf{in}}\ E_2 \rrbracket^{\Gamma}_{\text{expr}} = \begin{array}{l} \llbracket E_1 \rrbracket^{\Gamma}_{\text{expr}} \\ \texttt{stl}\ \Gamma(i) \\ \texttt{ldc 1} \\ \texttt{stl}\ \Gamma(i) + 1\ \ \textit{// mark variable as initialized} \\ \llbracket E_2 \rrbracket^{\Gamma[i:=\ldots]}_{\text{expr}} \end{array} \qquad \text{(let)}$$

Field storage

$[\![\texttt{fst}]\!]_{\text{fieldstorage}} =$ (fst)

```
sta 0
lds 2
```

$[\![\texttt{snd}]\!]_{\text{fieldstorage}} =$ (snd)

```
sta 1
lds 2
```

$[\![\texttt{hd}]\!]_{\text{fieldstorage}} =$ (hd)

```
lds 0  // first check list non-emptyness
ldc 0
eq
brt _EXCEPTION_EmptyList
sta 0  // then store hd
lds 2
```

$[\![\texttt{tl}]\!]_{\text{fieldstorage}} =$ (tl)

```
lds 0  // first check list non-emptyness
ldc 0
eq
brt _EXCEPTION_EmptyList
sta 1  // then store tl
lds 2
```

Field access

$[\![\texttt{fst}]\!]_{\text{fieldaccess}} =$ `ldh 0` (fst)

$[\![\texttt{snd}]\!]_{\text{fieldaccess}} =$ `ldh 1` (snd)

$[\![\texttt{hd}]\!]_{\text{fieldaccess}} =$ (hd)

```
lds 0  // first check list non-emptyness
ldc 0
eq
brt _EXCEPTION_EmptyList
ldh 0  // then access hd
```

$[\![\texttt{tl}]\!]_{\text{fieldaccess}} =$ (tl)

```
lds 0  // first check list non-emptyness
ldc 0
eq
brt _EXCEPTION_EmptyList
ldh 1  // then access tl
```

Operations

$[\![\texttt{!}]\!]_{\text{unop}} = \texttt{not}$	$[\![\texttt{-}]\!]_{\text{unop}} = \texttt{neg}$
$[\![\texttt{+}]\!]_{\text{binop}} = \texttt{add}$	$[\![\texttt{-}]\!]_{\text{binop}} = \texttt{sub}$
$[\![\texttt{*}]\!]_{\text{binop}} = \texttt{mul}$	$[\![\texttt{/}]\!]_{\text{binop}} = \texttt{div}$
$[\![\texttt{\%}]\!]_{\text{binop}} = \texttt{mod}$	
$[\![\texttt{==}]\!]_{\text{binop}} = \texttt{eq}$	$[\![\texttt{!=}]\!]_{\text{binop}} = \texttt{ne}$
$[\![\texttt{>=}]\!]_{\text{binop}} = \texttt{ge}$	$[\![\texttt{<=}]\!]_{\text{binop}} = \texttt{ge}$
$[\![\texttt{>}]\!]_{\text{binop}} = \texttt{gt}$	$[\![\texttt{<}]\!]_{\text{binop}} = \texttt{le}$
$[\![\texttt{\&\&}]\!]_{\text{binop}} = \texttt{and}$	$[\![\vert\vert]\!]_{\text{binop}} = \texttt{or}$

$[\![\texttt{:}]\!]_{\text{binop}} =$ `ajs -1`
`sth`
`ajs 1`
`sth`
`ajs -1`	

6.2 Calling convention

To call a function i, first we put its arguments on the stack, then we branch to _i, and when the function returns (via `unlink` and `ret`) 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.

Stack after link

	—	*// formal argument #1*
		⋮
	—	*// formal argument #k*
	—	*// PC return address*
MP	—	*// previous MP*
	0	*// local variable #1*
	0	*// local variable #1 initialized?*
		⋮
	0	*// local variable #n*
SP	0	*// local variable #n initialized?*

6.3 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 collection, however runtime exceptions are generated when an emptylist segment is accesses or written to.

7 LLVM IR compilation

A Code

dependence etc