CAP Companion Sheet

Mini-while, Mini-ML (abstract syntax)

Mini-while:

Numerical expressions:

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e + e & \textit{addition} \\ & | & e \times e & \textit{multiplication} \\ & | & \ldots & \end{array}$$

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \text{assign} \\ & | & \textit{skip} & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \\ & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

Mini-ML

$$\begin{array}{rcll} e & ::= & x & \text{identificator} \\ & | & c & \text{constant } (1, 2, \ldots, \textit{true}, \ldots) \\ & | & op & \text{primitive } (+, \times, \textit{fst}, \ldots) \\ & | & \texttt{fun } x \to e & \text{function} \\ & | & e\ e & \text{application} \\ & | & (e, e) & \text{pair} \\ & | & \texttt{let } x = e \texttt{ in } e & \text{local binding} \end{array}$$

Grammars-Visitors

Attributes An attribute is a mapping from elements of a given grammar to "an information". Defining a grammar attribution consists in giving a name and a type to the attributes you give to terminal/non terminal symbols of the grammar, and a recursive way to compute them from the grammar rules.

Python-ANTLR syntax

- Acces to a given non-terminal `name`: `ctx.name()`, if more than one: `ctx.name(0)`, `ctx.name(1)`...
- Recursive calls to sons : `self.visit(son)`
- Parsed chain of a terminal: `xx.getText()`.

Typing, static semantics

For mini-while We add declarations for the language:

$$D(decl) \quad ::= \quad var\ x : t \quad \text{type declaration}$$

From declarations we infer $\Gamma : Var \to Basetype$ with the two following rules:

$$\frac{}{varx : t \to_d [x \mapsto t]}$$

$$\frac{D_1 \to_d \Gamma_1 \quad D_2 \to_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \to_d \Gamma_1 \cup \Gamma_2}$$

Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$. For statements we invent the `void` type.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while } b \texttt{ do } S \texttt{ done} : \texttt{void}} \qquad \frac{D \to \Gamma \quad \Gamma \vdash C : \texttt{void}}{\Gamma \vdash DC : \texttt{void}}$$

Monomorphic typing of mini-ML

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}$$

Operational semantics

Natural semantics (big step) for mini-while : $(Stm, State) \to State$ where $State = Var \to \mathbb{Z}$

$$(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\texttt{skip}, \sigma) \to \sigma$$

Arithmetic:
$\mathcal{A} : expr \mapsto State \to \mathbb{Z}$

$$\mathcal{A}[v]\sigma = value(v)$$

$$\mathcal{A}[x]\sigma = \sigma(x)$$

$$\mathcal{A}[e_1 + e_2]\sigma = \mathcal{A}[e_1] + \mathcal{A}[e_2].$$

Boolean:

$\mathcal{B} : bexpr \mapsto State \to \mathbb{B}$

$$\frac{(S_1, \sigma) \to \sigma' \ (S_2, \sigma') \to \sigma''}{\big((S_1; S_2), \sigma\big) \to \sigma''}$$

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S_1, \sigma) \to \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = \mathit{ff} : \frac{(S_2, \sigma) \to \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S, \sigma) \to \sigma', (\texttt{while } b \texttt{ do } S, \sigma') \to \sigma''}{(\texttt{while } b \texttt{ do } S, \sigma) \to \sigma''}$$

$$\text{if } \mathcal{B}[b]\sigma = \mathit{ff} : (\texttt{while } b \texttt{ do } S, \sigma) \to \sigma$$

Reduction semantics for Mini-ML (call-by-value) $e \xrightarrow{v} v$ where e an expression and v a value following the following abstract syntax:

$$\begin{array}{rcll} v & ::= & c & \text{constant} \\ & | & op & \text{primitive} \\ & | & \texttt{fun } x \to e & \text{function} \\ & | & (v, v) & \text{pair} \end{array}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\texttt{let } x = e_1 \texttt{ in } e_2 \xrightarrow{v} v}$$

$$\frac{}{c \xrightarrow{v} c} \qquad \frac{e_1 \xrightarrow{v} (\texttt{fun } x \to e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1\ e_2 \xrightarrow{v} v}$$

$$\frac{}{op \xrightarrow{v} op}$$

$$\frac{e_1 \xrightarrow{v} + \quad e_2 \xrightarrow{v} (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \xrightarrow{v} n}$$

$$\frac{}{(\texttt{fun } x \to e) \xrightarrow{v} (\texttt{fun } x \to e)}$$

$$\frac{e_1 \xrightarrow{v} fst \quad e_2 \xrightarrow{v} (v_1, v_2)}{e_1\ e_2 \xrightarrow{v} v_1}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)}$$

Dataflow Analysis (liveness)

- A variable that appears on the left hand side of an assignment is *killed* by the block. Tests do no kill variables.
- A *generated* variable is a variable that appears in the block.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup \{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} & \end{cases}$$

$$LV_{entry}(\ell) = \big(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\big) \cup gen_{LV}(\ell)$$

3 address code generation

`newTemp()` : $\rightarrow \mathbb{N}$ and `newLabel()` : $\rightarrow \mathbb{N}$.

c	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, c))`
	`return dr`

x	`#get the place associated to x.`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, 1))`
	`return dr`

$e_1 < e_2$	`dr <-newTemp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`dr <- newTemp()`
	`endrel <- newLabel()`
	`code.add(InstructionLET(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)`
	`code.add(InstructionLET(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

x = e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`if x has a location loc:`
	`code.add(instructionCOPY(loc,dr))`
	`else:`
	`storeLocation(x,dr)`

S1; S2	`#concat codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` b `do` S `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S) #execute S`
	`code.add(InstructionJUMP(ltest))#and jump to the test`
	`code.addLabel(lendwhile) #else it is done.`

LEIA ISA

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	`wmem`	wmem	write to memory	
0	0	0	1	`add`	ALU	addition	$z(i)$
0	0	1	0	`sub`	ALU	subtraction	$z(i)$
0	0	1	1	`snif`	snif	skip next if	$s(i)$
0	1	0	0	`and`	ALU	logical bitwise and	$s(i)$
0	1	0	1	`or`	ALU	logical bitwise or	$s(i)$
0	1	1	0	`xor`	ALU	logical bitwise xor	$s(i)$
0	1	1	1	`lsl`	ALU	logical shift left	$z(i)$
1	0	0	0	`lsr`	ALU	logical shift right	$z(i)$
1	0	0	1	`asr`	ALU	arithmetic shift right	$z(i)$
1	0	1	0	`call`	call	sub-routine call	
1	0	1	1	`jump` `return`	jump	relative jump if `offset` $\neq 1$ return from call if `offset` $= 1$	
1	1	0	0	`letl`	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	`leth`	leth	8-bit constant to high half of Rd	
1	1	1	0	`print`	print	print or refresh	
1	1	1	1	`rmem` `copy`	rmem	read from memory if i=0 register-to-register copy if i=1	