MIF08 Feuille d'accompagnement

Mini-while (syntaxe abstraite)

Expressions Booléennes :

$$\begin{array}{rcll} b & ::= & true & constante \\ & | & false & constante \\ & | & b\ or\ b & ou \\ & | & b\ and\ b & et \\ & | & \ldots & \end{array}$$

Expressions numériques :

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

Instructions :

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \text{affectation} \\ & | & skip & \text{rien} \\ & | & S_1; S_2 & \text{séquence} \\ & | & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 & \text{test} \\ & | & \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} & \text{boucle} \end{array}$$

Typage

On ajoute les déclarations dans le langage :

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

Les informations de déclaration fournissent $\Gamma : Var \to Basetype$ construit à l'aide des règles :

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

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

Ainsi la déclaration `var x,y:int;` fournit $\Gamma : \begin{cases} x & \to int \\ y & \to int \end{cases}$.

Ensuite un jugement de type est de la forme $\Gamma \vdash e : \tau \in Basetype$ pour Γ construit comme précédemment. Les programmes/instructions bien typé(e)s sont de type `void`. Un programme est bien typé si son code type `void` sous l'environnement Γ :

$$\frac{D \to \Gamma \quad \Gamma \vdash C : \texttt{void}}{\Gamma \vdash DC : \texttt{void}}$$

Règles pour les expressions (les autres sont similaires) :

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

Règles pour les instructions :

$$\frac{\Gamma \vdash e : \texttt{t} \qquad \Gamma \vdash x : \texttt{t}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S1\ \texttt{else}\ S2 : \texttt{void}}$$

$$\frac{\Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash S1; S2 : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

LEIA ISA

Liste des instructions de la machine cible. Pour initialiser des registres à constante, vous pouvez utiliser la macro `.let r 585`). L'instruction `snif op1 <condition> op2` "skip next if" désactive l'instruction suivante si la condition est vraie. Les opérandes 1 et 2 sont des registres (temporaires dans le cas du 3-adresse, physiques sinon) ou des constantes immédiates. La condition peut-être : `eq`, `neq`, `sgt`, `slt`, `gt`, `ge`, `lt` et `le`.

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	

Génération de code 3-adresses

On rappelle the le code 3 addresses LEIA a le même jeu d'instructions que le code LEIA standard, à part pour les conditions qui utilisent l'instruction `condJUMP(label,t1,condition,t2)` et que les registres sont remplacés par des temporaires. Vous pouvez utiliser `.let` si vous le désirez.

`newTemp` : $() \to \mathbb{N}$ crée un nouveau temporaire (temp0, temp1, ...) et `newLabel` : $() \to \mathbb{N}$ crée un nouveau label (donnez le nom que vous voulez).

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

x	`# récupère le temporaire associé à 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`

$e_1 < e_2$	`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)`
	`# copie du résultat dans le tmp associé à x`
	`let loc = loc(x) in`
	`code.add(instructionCOPY(loc,dr))`

S1 ; S2	`#concatène les codes générés`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse, saute à "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)`
	`#si la condition est fausse saute à la fin`
	`code.add(`
	`InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S) #exécute S`
	`code.add(InstructionJUMP(ltest))#et va au test`
	`code.addLabel(lendwhile) #fin while`