Programming Languages and Compiler Design

Generation of Assembly-code

Yliès Falcone, Jean-Claude Fernandez

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes
(Université Joseph Fourier, Grenoble INP)

Academic Year 2015 - 2016

Outline - Generation of Assembly-code

Outline - Generation of Assembly-code

Main issues for code generation

- input: (well-typed) source pgm AST (or intermediate code)
- output: machine level code (assembly, relocatable, or absolute code)

Expected properties for the output

- compliance with the target machine
  instruction set, architecture, memory access, OS, ...
- correctness of the generated code
  semantically equivalent to the source pgm
- optimality w.r.t. non-functional criteria
  execution time, memory size, energy comsumption, ...

Main issues for code generation (ctd)

Tasks of the Code Generator

- **Instruction selection**: choosing appropriate target-machine instructions to implement the (IR) statements.
  Complexity depends on:
  - how abstract is the IR,
  - "expressiveness of instruction set" (e.g., support of some types),
  - expected quality of the output code according to some criteria (speed and size).
- **Registers allocation and assignment**: deciding what variables to keep in which registers at every location (when the target machine uses registers).
- **Instruction ordering**: deciding the scheduling order for the execution of instructions.
  - It affects the efficiency of the code and the required registers.
  - It is generally not possible to obtain an optimal (NP-complete) $\Rightarrow$ heuristics

A pragmatic approach

AST

↓

intermediate code generation

↓

Intermediate Representation 1 ⇄ optimization(s)

↓

. . .

↓

Intermediate Representation n ⇄ optimization(s)

↓

(final) code generation

↓

target machine code

Intermediate Representations

- Abstractions of a real target machine
  - generic code level instruction set
  - simple addressing modes
  - simple memory hierarchy
- Examples
  - a "stack machine"
  - a "register machine"
  - etc.

Remark Other intermediate representations are used in the optimization phases. □

Outline - Generation of Assembly-code

Machine "M"

Machine with Registers

- Unlimited registers, denoted by `Ri`.
- Special registers:
  - program counter `PC`,
  - stack pointer `SP`,
  - frame pointer `FP`,
  - register `R0` (contains always 0).

  (the exact purpose of these registers will become clear later)

Instructions, addresses, and integers take 4 bytes in memory.

Addressing

- Address of variable `x` is `E - offx` where:
  - `E` = address of the environment where `x` is defined
  - `offx` = offset of `x` within this environment (staticaly computed, stored in the symbol table)
- Addressing modes:
  `Ri`, `val` (immediate), `Ri +/- Rj`, `Ri +/- offset`

Instruction Set

- Usual arithmetic instructions `OPER`: `ADD`, `SUB`, `AND`, etc.
- Usual (conditional) branch instructions `BRANCH`: `BA`, `BEQ` (=), `BGT` (>), BLT (<), `BGE` (≥), `BLE` (≤), `BNE` (≠).

instruction	informal semantics
`OPER Ri, Rj, Rk`	Ri ← Rj oper Rk
`OPER Ri, Rk, val`	Ri ← Rj oper val
`CMP Ri, Rj`	Ri - Rj (set cond flags)
`LD Ri, [adr]`	Ri ← Mem[adr]
`ST Ri, [adr]`	Mem[adr] ← Ri
`BRANCH label`	if cond then PC ← `label` else PC ← PC + 4
`CALL label`	branch to the procedure labelled with `label` `PUSH(PC)` || PC← `label`
`CALL R`	branch to the address contained in register `R` `PUSH(PC)` || PC← `R`
`RET`	end of procedure

Language **While**

Reminder

p	::=	d ; s
d	::=	var x | d ; d
s	::=	x := a | s ; s | if b then s else s | while b do s od
a	::=	n | x | a + a | a * a | ...
b	::=	a = a | b and b | not b | ...

Remark Terms are well-typed.
→ distinction between boolean and arithmetic expr. □

Language `While`

Reminder

Informal code generation

Give the "Machine M" code for the following statements:

1. `y := x+42 * (3+y)`
2. `if (not x=1) then x := x+1`
   `else x := x-1 ; y := x ;`

Outline - Generation of Assembly-code

Functions for code generation

Notation

- Code*: instruction sequences for machine "M"
- ||: concatenation operator for code and sequences of code

`GCStm` : Stm → Code*

`GCStm(s)` *computes the code* `C` *corresponding to statement* `s`.

`GCAExp` : Exp → Code* × `Reg`

`GCAExp(e)` returns a pair `(C, i)` where `C` is the code allowing to

1. computes the value of `e`,
2. stores it in `Ri`.

`GCBExp` : BExp× $\mathcal{L}$abel× $\mathcal{L}$abel → Code*

`GCBExp(b, ltrue, lfalse)` produces the code `C` that computes the value of `b` and branches to label `ltrue` when this value is "true" and to `lfalse` otherwise.

Auxiliary functions

`AllocRegister`	:	→ `Reg` allocates a new register `Ri`
`newLabel`	:	→ `Labels` produces a new `label`
`GetOffset`	:	Var → $\mathbb{Z}$ returns the offset corresponding to the specified name which depends on the position at which the variable is declared (shall be defined precisely for blocks and procedures)

Function `GCStm`

Assignments, sequential and iterative compositions

`GCStm (x := e)`	=	`Let`
		
		`in`
			(C,i)=`GCAExp`(e),
			k=`GetOffset`(x)
			C|| ST Ri, [FP + k]

`GCStm` (s_1 ; s_2)	=	`Let`
		
		`in`
			C_1 = `GCStm`(s_1),
			C_2 = `GCStm`(s_2)
			C_1 || C_2

`GCStm (while e do s od)`	=	`Let`
		
		`in`
			lb=`newLabel()`,
			ltrue=`newLabel()`,
			lfalse=`newLabel()`
			lb:||
			`GCBExp`(e,ltrue,lfalse)||
			ltrue:||
			`GCStm`(s)||
			BA lb||
			lfalse:

Function GCStm (ctd)

Conditional statement

GCStm (if e then s_1 else s_2)	=	Let	lnext=newLabel(),
			ltrue=newLabel(),
			lfalse=newLabel()
		in	GCBExp(e,ltrue,lfalse)‖
			ltrue:
			GCStm(s_1)‖
			BA lnext ‖
			lfalse:‖
			GCStm(s_2)‖
			lnext:

Function GCAexp

Arithmetic expressions

GCAExp(x)	=	Let	i=AllocRegister()
			k=GetOffset(x)
		in	((LD Ri, [FP + k]),i)
GCAExp(n)	=	Let	i=AllocRegister()
		in	((ADD R_i, R_0, n),i)
GCAExp(e_1 + e_2)	=	Let	(C_1,i_1)=GCAExp(e_1),
			(C_2,i_2)=GCAExp(e_2),
			k=AllocRegister()
		in	((C_1‖C_2‖ ADD Rk, Ri_1, Ri_2),k)

Function GCBexp

Boolean expressions

GCBExp (e_1 = e_2,ltrue,lfalse)	=	Let	(C_1, i_1)=GCAExp(e_1),
			(C_2, i_2)=GCAExp(e_2),
		in	C_1‖C_2‖
			CMP Ri_1, Ri_2
			BEQ ltrue
			BA lfalse
GCBExp (e_1 and e_2,ltrue,lfalse)	=	Let	l=newLabel()
		in	GCBExp(e_1,l,lfalse)‖
			l:‖
			GCBExp(e_2,ltrue,lfalse)
GCBExp(NOT e,ltrue,lfalse)	=		GCBExp(e,lfalse,ltrue)

Exercise

Informal code generation

Give the "Machine M" code for the following statements:

1. x := 10; while $x > 10$ do $x := x - 1$ od
2.

Adding new statements to **While**

Extend the code generation function

- to consider statements of the form repeat S until b,
- to consider Boolean expressions of the form b1 xor b2,
- to consider arithmetical expressions of the form b ? e1 : e2.

Outline - Generation of Assembly-code

Blocks

Syntax

$$
\begin{array}{rcl}
S & ::= & \cdots \mid \textbf{begin } D_V \text{ ; } S \textbf{ end} \\
D_V & ::= & \textbf{var } x \mid D_V \text{ ; } D_V
\end{array}
$$

Remark Variables are not initialized and assumed to be of type **Int**. □

Problems raised for code generation

→ to preserve scoping rules:

- local variables should be *visible* inside the block,
- their *lifetime* should be limited to block execution.

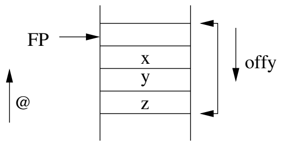
Possible locations to store local variables

→ registers vs memory

Storing local variables in memory - Example 1

Access to local variables within a block

```
begin
   var x ; var y ; var z ;
   ...
end
```



- A *memory environment* is associated to each declaration in D_V.
- Register FP contains the address of the current environment.
- (Static) offsets are associated to each local variables.

Definition (Offset of a local variable)

The offset of a local variable is $-4 \times i$, where i is the position of the variable in the sequence of local declarations.

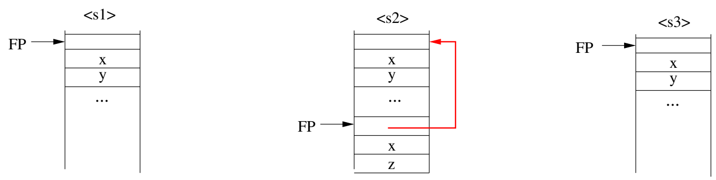
Example (Offset of a local variable)

For `var x ; var y ; var z ;`: `GetOffset(x)` = -4, `GetOffset(y)` = -8, `GetOffset(z)` = -12.

Storing local variables in memory - Example 2

Access to local variables in case of nested blocks

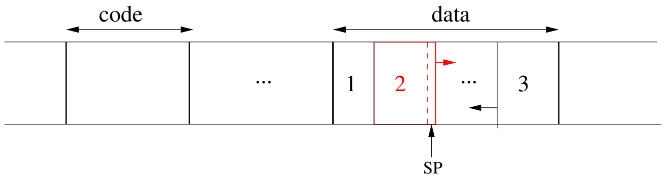
```
begin
   var x ; var y ; <s1>
   begin
      var x ; var z ; <s2>
   end ;
   <s3>
end
```



- entering/leaving a block → allocate/de-allocate a mem. env.
- nested block env. have to be linked together: "Ariane link"

⇒ a stack of memory environments . . . (∼ operational semantics)

Structure of the memory



1: global variables

2: execution stack, SP = last occupied address

3: heap (for dynamic allocation)

Code generation for variable declarations

SizeDecl : $D_V \rightarrow \mathbb{N}$

SizeDecl(d) *computes the size of declarations* d

SizeDecl (var x)	=	4 (x of type Int)	
SizeDecl (d_1 ; d_2)	=	Let	v_1 = SizeDecl(d_1), v_2 = SizeDecl(d_2)
		in	$v_1 + v_2$

Code generation for blocks

GCStm (begin d ; s ; end)	=	Let	size =SizeDecl(d), C=GCStm(s)
		in	ADD, SP, SP, -4 ‖ ST FP, [SP] ‖ ADD FP, SP, 0 ‖ ADD SP, SP, -size ‖ C ‖ ADD SP, FP, 0 ‖ LD FP, [SP] ‖ ADD SP, SP, 4 ‖

With the help of some auxiliary functions ...

prologue(size)	epilogue	push register (Ri)

ADD SP, SP, -4
ST FP, [SP]
ADD FP, SP, 0
ADD SP, SP, -size
	ADD SP, FP, 0
	LD FP, [SP]
	ADD SP, SP, +4
		ADD SP, SP, -4
		ST Ri, [SP]

GCStm (begin d ; s ; end)	=	Let	size =SizeDecl(d), C=GCStm(s)
		in	Prologue(size) ‖ C ‖ Epilogue

Access to variables from a block ?

```
...
begin
   var ...
   x := ...
end
```

What is the memory address of x ?

- if x is a local variable (w.r.t the current block)
  $\Rightarrow$ adr(x) = FP + GetOffset(x)
- if x is a non local variable
  $\Rightarrow$ it is defined in a "nesting" memory env. E
  $\Rightarrow$ adr(x) = adr(E) + GetOffset(x)
  adr(E) can be accessed through the "Ariane link" ...

Access to *non local* variables

The number n of indirections to perform on the "Ariane link" depends on the "distance" between:

- the nesting level of the current block : p
- the nesting level of the target environment : r

More precisely:

- $r \leq p$
- $n = p - r$

$\Rightarrow$ n can be statically computed ...

Example

```
begin
   var x ; /* env. E1, nesting level = 1 */
   begin
     var y ; /* env. E2, nesting level = 2 */
     begin
        var z ; /* env. E3, nesting level = 3 */
        x := y + z /* s, nesting level = 3 */
     end
   end
end
```

From statement s:

- no indirection to access to z
- 1 indirection to access to y
- 2 indirections to access to x

Code generation for variable access

1. the nesting level r of each identifier x is computed during type-checking;
2. it is associated to each occurrence of x in the AST (via the symbol table)
3. function GCStm keeps track of the current nesting level p (incremented/decremented at each block entry/exit)

adr(x) is obtained by executing the following code:

- if $r = p$:

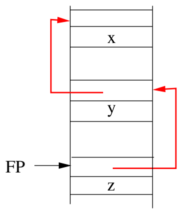
  FP + GetOffset(x)

- if $r < p$:

  LD Ri, [FP]
  LD Ri, [Ri]} $(p - r - 1)$ times
  Ri + GetOffset(x)

Example (ctn'd)

```
begin
   var x ; /* env. E1, nesting level = 1 */
   begin
     var y ; /* env. E2, nesting level = 2 */
     begin
        var z ; /* env. E3, nesting level = 3 */
        x := y + z /* s, nesting level = 3 */
     end
   end
end
```




```
LD  R1, [FP]          ! R1 = adr(E2)
LD R2,  [R1 + offy]   ! R2 = y
LD R3,  [FP + offz]   ! R3 = z
ADD R4, R2, R3        ! R4 = y+z
LD  R5, [FP]
LD  R5, [R5]          ! R5 = adr(E1)
ST R4,  [R5 + offx]   ! x = y + z
```

Code generated for statement s

Outline - Generation of Assembly-code

Syntax of Language **Proc**

Reminder

Procedure declarations:

$$
\begin{array}{rcl}
D_P & ::= & \textbf{proc}\ p\ (FP_L)\ \textbf{is}\ S\ ;\ D_P \mid \epsilon \\
FP_L & ::= & \textbf{x},\ FP_L \mid \epsilon
\end{array}
$$

Statements:

$$
\begin{array}{rcl}
S & ::= & \cdots \mid \textbf{begin}\ D_V\ ; D_P\ ;\ S\ \textbf{end} \mid \textbf{call}\ p(EP_L) \\
EP_L & ::= & AExp,\ EP_L \mid \epsilon
\end{array}
$$

FP_L: list of formal parameters; EP_L: list of effective parameters

Remark We assume value-passing of integer parameters. □

Example

```
var z ;

proc p1 () is
    begin
      proc p2(x, y) is z := x + y ;
      z := 0 ;
      call p2(z+1, 3) ;
    end

proc p3 (x) is
    begin
       var z ;
       call p1() ; z := z+x ;
    end

call p3(42) ;
```

Main issues for code generation with procedures

Procedure P is calling procedure Q . . .

Before the call:

- set up the memory environment of Q
- evaluate and "transmit" the effective parameters
- switch to the memory environment of Q
- branch to first intruction of Q

During the call:

- access to local/non local procedures and variables
- access to parameter values

After the call:

- switch back to the memory environment of P
- resume execution to the instruction of P following the call

Access to non-local variables

```
proc main is
begin              /* definition env. of p */
   var x ;
   proc p() is x:=3 ;
   proc q() is
     begin
         var x ;
         proc r() is call p() ;
         call r() ;
     end ;
   call q() ;
end
```

Static binding ⇒ when p is executed:

- access to the memory env. of main =
  definition environment of the callee, static link
- access to the memory env. of r
  memory environment of the caller, dynamic link

Information exchanged between *callers* and *callees*?

- parameter values
- return address
- address of the caller memory environment (dynamic link)
- address of the callee environment definition (static link)

This information should be stored in a memory zone:

- dynamically allocated
  (exact number of procedure calls cannot be foreseen at compile time)
- accessible from both parties
  (those addresses should be computable by the caller and the callee)

⇒
inside the execution stack, at well defined offsets w.r.t FP

A possible "protocol" between the two parties

Before the call, the caller:

- evaluates the effective parameters
- pushes their values
- pushes the static link of the callee
- pushes the return address, and branch to the callee's 1st instruction

When it begins, the callee:

- pushes FP (dynamic link)
- assigns SP to FP (memory env. address)
- allocates its local variables on the stack

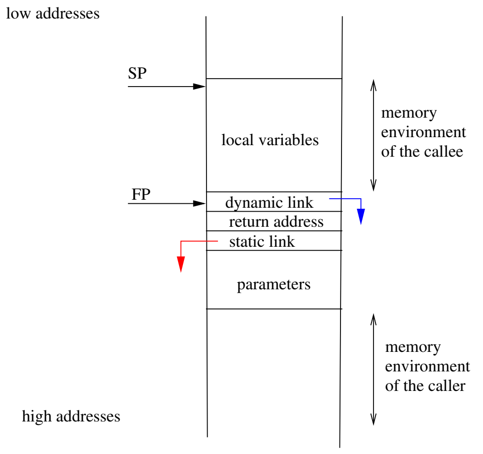
When it ends, the callee:

- de-allocates its local variables
- restores FP to caller's memory env. (dynamic link)
- branch to the return address, and pops it from the stack

After the call, the caller

- de-allocates the static link and parameters

Organization of the execution stack



Addresses, from the callee:

- loc. variables: FP+d, d<0
- dynamic link: FP
- return address: FP+4
- static link: FP+8
- parameters: FP+d, d>=12

Memory environment of the callee

...	0
Loc. var_n	←SP, FP- 4*n
...	
Loc. var_1	←FP-4
Dynamic link	←FP
Return address	←FP+4
Static link	←FP+8
Param_n	←FP+12
...	
Param_1	←FP+8+4*n

Definition (Offset of a variable or a parameter)

- For local variable $\texttt{var}_i$, as before, $\texttt{GetOffset}(\texttt{var}_i)$ is $-4 \times i$.
- For parameter $\texttt{param}_\texttt{i}$, $\texttt{GetOffset}(\texttt{param}_i)$ is $8 + 4 \times (n + 1 - i)$.

Code generation for a procedure declaration

GCProc : $D_P \rightarrow \text{Code}^*$
GCStm(dp) *computes the code* C *corresponding to procedure declaration* dp.

GCProc (**proc** p (FP_L) **is** s end) = Let C=GCStm(s) in Prologue(0) ‖ C ‖ Epilogue

GCProc (**proc** p (FP_L) **is** begin dv ; dp ; s end) = Let size =SizeDecl(dv), C=GCStm(s) in Prologue(size) ‖ C ‖ Epilogue

Remark GCProc is applied to each procedure declaration. □

Code generation for a procedure declaration (ctd)

Prologue & Epilogue

Prologue (size):

```
push (FP)             ! dynamic link
ADD FP, SP, 0         ! FP := SP
ADD SP, SP, -size     ! loc. variables allocation
```

Epilogue:

```
ADD SP, FP, 0         ! SP := FP, loc. var. de-allocation
LD FP, [SP]           ! restore FP
ADD SP, SP, +4        ! erase previous backup of FP
RET                   ! return to caller
```

RET:

```
LD PC, [SP]  //  ADD SP, SP, +4
```

Code generation for a procedure call

Four steps:

1. evaluate and push each effective parameter
2. push the static link of the callee
3. push the return address and branch to the callee
4. de-allocate the parameter zone

GCStm (**call** p (ep))	=	Let	(C, size) = GCParam(ep)
		in	
			C ‖
			Push (StaticLink(p)) ‖
			CALL p ‖
			ADD SP, SP, size+4

`CALL p`:

ADD R1, PC, +4 // Push (R1) // BA p

Parameters evaluation

GCParam : $EP_L \rightarrow \text{Code}^* \times \mathbb{N}$
GCStm(ep)=(c,n) *where* c *is the code to evaluate and "push" each effective parameter of* ep *and* n *is the size of pushed data.*

GCParam (ε)	=	(ε, 0)	
GCParam (a ; ep)	=	Let	
			(Ca, i) = GCAexp (a),
			(C, size) = GCParam (ep)
		in	
			(Ca ‖ Push (R_i) ‖ C, 4 + size)

Static link and non-local variable access?

Principle

- A global (unique) name is given to each identifier:

```
proc Main is
   proc P1 (...) is
     ...
        proc Pn (...) is
         begin
            var x ...
         end
```

  → x is named $Main.P_1.\cdots.P_n.x$

- This notation induces a partial order:

$$(Main{\cdot}P_1\cdots P_n \leq Main{\cdot}P'_1\cdots P'_{n'}) \Leftrightarrow (n \leq n' \text{ and } \forall k \leq n{\cdot}P_k = P'_k)$$

- For an identifier $x = Main \cdot P_1 \cdots P_n \cdot x$,
  $x^\bullet = Main.P_1 \cdots P_n$ is the definition environment of x
- For any identifier x (variable or procedure), procedure P can access x iff $x^\bullet \leq P$.

Static link and non-local variable access?

Examples

- A variable x declared in P can be accessed from P since $x^\bullet = P$ (hence $x^\bullet \leq P$).
- If g and x are declared in f, then x can be accessed from g since $x^\bullet = f$ and $f \leq g$.
- If x and f_1 are declared in $Main$, f_2 is declared in f_1, then x can be accessed from f_2 since $x^\bullet = Main$, $f_2 = Main.f_1.f_2$ ($x^\bullet \leq f_2$)
- If p_1 and p_2 are both declared in $Main$, x is declared in p_1, then x cannot be accessed from p_2, since $x^\bullet = Main.p_1$ and $Main.p_1 \not\leq Main.p_2$

Code Generation for accessing (non-) local identifiers

Let us consider:

- d_x: offset of x (variables or parameters) in its definition environment ($x^\bullet$);
- P: current procedure.

Condition	x = variable or parameter	x = procedure
$x^\bullet = P$	adr(x) = FP+d_x	SL(x) = FP
$x^\bullet < P$ $x = M.P_1 \cdots P_k$ $P = M.P_1 \cdots P_k \cdots P_n$	n-k-1 indirections LD R, [FP+8] LD R, [R+8] } $\times (n-k-1)$ adr(x) = R+d_x	n-k-1 indirections LD R, [FP+8] LD R, [R+8] } $\times (n-k-1)$ SL(x)=R

Back to the first example

```
var z ;
proc p1 () is
    begin
      proc p2(x, y) is z := x + y ;
      z := 0 ;
      call p2(z+1, 3) ;
    end
proc p3 (x) is
    begin
       var z ;
       call p1() ; z := z+x ;
    end
call p3(42) ;
```

Exercise

- Give the execution stack when p2 is executed.
- Give the code for procedures p1 and p2.