Compiler Construction

Lecture 18: Code Generation III (Translation to Intermediate Code)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc08/

Summer semester 2008

Outline

The Abstract Machine AM

Definition (Abstract machine for EPL)

The abstract machine for EPL (AM) is defined by the state space

$$S := PC \times DS \times PS$$

with

- the program counter $PC := \mathbb{N}$,
- the data stack $DS := \mathbb{Z}^*$ (top of stack to the right), and
- the procedure stack (or: runtime stack) $PS := \mathbb{Z}^*$ (top of stack to the left).

Thus a state $s = (l, d, p) \in S$ is given by

- a program label $l \in PC$,
- a data stack $d = d.r : \ldots : d.1 \in DS$, and
- a procedure stack $p = p.1 : \ldots : p.t \in PS$.

AM Instructions

Definition (AM instructions)

The set of AM instructions is divided into

arithmetic instructions: `ADD`, `MULT`, ...

Boolean instructions: `NOT`, `AND`, `OR`, `LT`, ...

jumping instructions: `JMP(`ca`)`, `JFALSE(`ca`)` $(ca \in PC)$

procedure instructions: `CALL(`ca`,`dif`,`loc`)` $(ca \in PC,\ dif, loc \in \mathbb{N})$, `RET`

transfer instructions: `LOAD(`dif`,`off`)`, `STORE(`dif`,`off`)` $(dif, off \in \mathbb{N})$,
`LIT(`z`)` $(z \in \mathbb{Z})$

Semantics of Instructions

Definition (Semantics of AM instructions (1st part))

The semantics of an AM instruction O

$$[\![O]\!] : S \dashrightarrow S$$

is defined as follows:

$$
\begin{aligned}
[\![\texttt{ADD}]\!](l, d : z_1 : z_2, p) &:= (l+1, d : z_1 + z_2, p) \\
[\![\texttt{NOT}]\!](l, d : b, p) &:= (l+1, d : \neg b, p) && \text{if } b \in \{0,1\} \\
[\![\texttt{AND}]\!](l, d : b_1 : b_2, p) &:= (l+1, d : b_1 \wedge b_2, p) && \text{if } b_1, b_2 \in \{0,1\} \\
[\![\texttt{OR}]\!](l, d : b_1 : b_2, p) &:= (l+1, d : b_1 \vee b_2, p) && \text{if } b_1, b_2 \in \{0,1\} \\
[\![\texttt{LT}]\!](l, d : z_1 : z_2, p) &:= \begin{cases} (l+1, d : 1, p) & \text{if } z_1 < z_2 \\ (l+1, d : 0, p) & \text{if } z_1 \geq z_2 \end{cases} \\
[\![\texttt{JMP(}ca\texttt{)}]\!](l, d, p) &:= (ca, d, p) \\
[\![\texttt{JFALSE(}ca\texttt{)}]\!](l, d : b, p) &:= \begin{cases} (ca, d, p) & \text{if } b = 0 \\ (l+1, d, p) & \text{if } b = 1 \end{cases}
\end{aligned}
$$

Outline

Structure of Procedure Stack I

The semantics of procedure and transfer instructions requires a particular structure of the procedure stack $p \in PS$: is must be composed of frames (or: activation records) of the form

$$sl : dl : ra : v_1 : \ldots : v_k$$

where

static link sl: points to frame of surrounding declaration environment
$\Longrightarrow$ used to access non-local variables

dynamic link dl: points to previous frame (i.e., of calling procedure)
$\Longrightarrow$ used to remove topmost frame after termination of procedure call

return address ra: program label after termination of procedure call
$\Longrightarrow$ used to continue program execution after termination of procedure call

local variables v_i: values of locally declared variables

Structure of Procedure Stack II

- Frames are created whenever a procedure call is performed
- Two special frames:

  `I/O` frame: for keeping values of `in/out` variables
  ($sl = dl = ra = 0$)

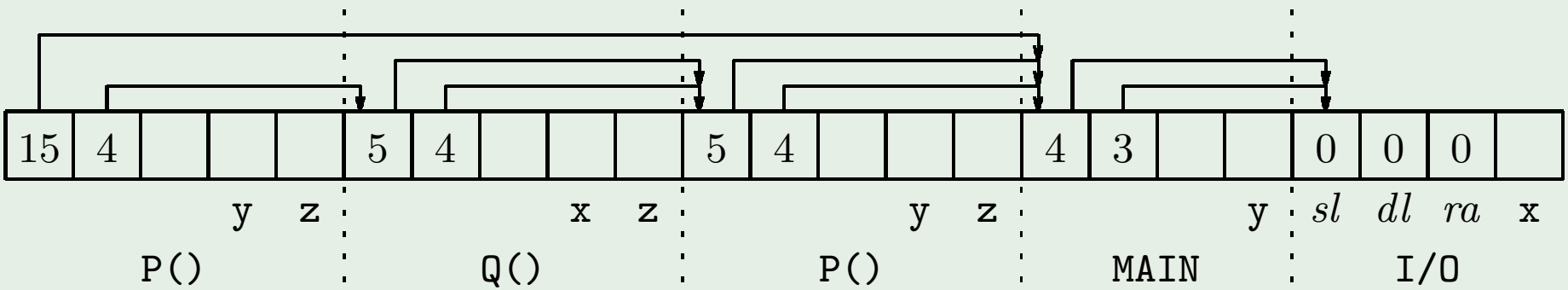
  `MAIN` frame: for keeping values of top-level block
  ($sl = dl =$ `I/O` frame)

Structure of Procedure Stack III

Example 18.1 (cf. Example 16.8)

```
in/out x;
  const c = 10;
  var y;
  proc P;
    var y, z;
    proc Q;
      var x, z;
      [... P() ...]
    [... Q() ...]
  proc R;
    [... P() ...]
  [... P() ...].
```

Procedure stack after second call of P:



Structure of Procedure Stack IV

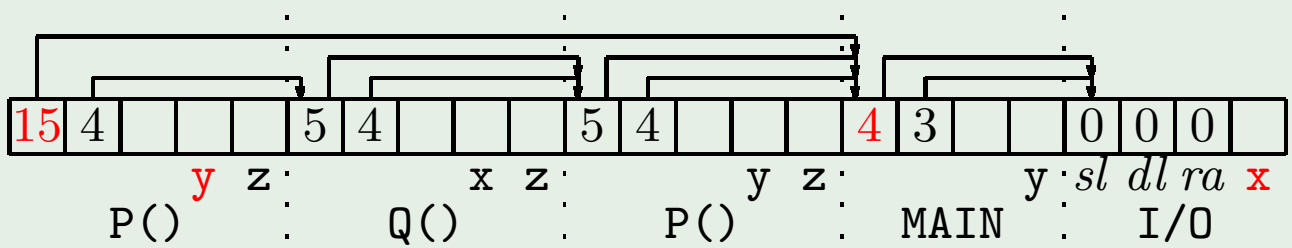
Observation:

- The usage of a variable in a procedure body refers to its innermost declaration.
- If the level difference between the usage and the declaration is *dif*, then a chain of *dif* static links has to be followed to access the corresponding frame.

Example 18.2 (cf. Example 18.1)

```
in/out x;
  const c = 10;
  var y;
  proc P;
    var y, z;
    proc Q;
      var x, z;
      [... P() ...]
    [... x ... y ... Q() ...]
  proc R;
    [... P() ...]
  [... P() ...].
```

Procedure stack after second call of P:



P uses x $\Longrightarrow$ *dif* $= 2$ P uses y $\Longrightarrow$ *dif* $= 0$

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned} \text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif) \end{aligned}$$

Example 18.4 (cf. Example 18.2)

In the second call of `P` (from `Q`): $dif = 2$

$$\begin{aligned} & \text{base}(p, 0) = 1 \\ \Longrightarrow\ & \text{base}(p, 1) = 1 + p.1 = 6 \\ \Longrightarrow\ & \text{base}(p, 2) = 6 + p.6 = 11 \end{aligned}$$

$$\Longrightarrow\ sl = \text{base}(p, 2) + \underbrace{2}_{\texttt{y,z}} + \underbrace{2}_{ra, dl} = 15$$

Semantics of Procedure Instructions

- `CALL(`ca`,`dif`,`loc`)` with
  - code address $ca \in PC$
  - level difference $dif \in \mathbb{N}$
  - number of local variables $loc \in \mathbb{N}$

  creates the new frame and jumps to the given address (= starting address of procedure)
- `RET` removes the topmost frame and returns to the calling site

Definition 18.5 (Semantics of procedure instructions)

The semantics of a procedure instruction O, $[\![O]\!] : S \dashrightarrow S$, is defined as follows:

$$[\![\texttt{CALL(}ca\texttt{,}dif\texttt{,}loc\texttt{)}]\!](l,d,p)$$
$$:= (ca, d, \underbrace{(\text{base}(p, dif) + loc + 2)}_{sl} : \underbrace{(loc+2)}_{dl} : \underbrace{(l+1)}_{ra} : \underbrace{0 : \ldots : 0}_{loc \text{ times}} : p)$$

$$[\![\texttt{RET}]\!](l, d, p.1 : \ldots : p.t)$$
$$:= (\underbrace{p.3}_{ra}, d, p.(\underbrace{p.2}_{dl} + 2) : \ldots : p.t) \qquad \text{if } t \geq p.2 + 2$$

Semantics of Transfer Instructions

- `LOAD(`dif`,`off`)` and `STORE(`dif`,`off`)` with
  - level difference $dif \in \mathbb{N}$
  - variable offset $off \in \mathbb{N}$

  respectively load and store variable values between data and procedure stack, following a chain of dif static links
- `LIT(`z`)` loads the literal constant $z \in \mathbb{Z}$

Definition 18.6 (Semantics of transfer instructions)

The semantics of a transfer instruction O, $[\![O]\!] : S \dashrightarrow S$, is defined as follows:

$$
\begin{aligned}
[\![\texttt{LOAD(}dif\texttt{,}off\texttt{)}]\!](l,d,p) &:= (l+1, d : p.(\text{base}(p, dif) + off + 2), p) \\
[\![\texttt{STORE(}dif\texttt{,}off\texttt{)}]\!](l,d:z,p) &:= (l+1, d, p[\text{base}(p, dif) + off + 2 \mapsto z]) \\
[\![\texttt{LIT(}z\texttt{)}]\!](l,d,p) &:= (l+1, d : z, p)
\end{aligned}
$$

Definition 18.7 (Semantics of AM programs)

An AM program is a sequence of $k \geq 1$ labeled AM instructions:

$$P = a_1 : O_{a_1}; \ldots; a_k : O_{a_k}$$

where $a_i \in PC$ with $a_i = a_1 + i - 1$ for every $i \in [k]$. The set of all AM programs is denoted by AM.

The semantics of AM programs is determined by

$$[\![.]\!] : AM \times S \dashrightarrow S$$

with

$$[\![P]\!](l, d, p) := \begin{cases} [\![P]\!]([\![O_l]\!](l, d, p)) & \text{if } l \in \{a_1, \ldots, a_k\} \\ (l, d, p) & \text{otherwise} \end{cases}$$

Outline

Translation of EPL into AM Programs

Goal: define translation mapping

$$\text{trans} : Pgm \dashrightarrow AM$$

The translation employs a symbol table:

$$\begin{aligned} Tab := \{\text{st} \mid \text{st} : Ide \dashrightarrow\ & (\{\texttt{const}\} \times \mathbb{Z}) \\ & \cup (\{\texttt{var}\} \times Lev \times Off) \\ & \cup (\{\texttt{proc}\} \times PC \times Lev \times Size)\} \end{aligned}$$

whose entries are created by declarations and are used for translating commands:

variable declarations: declaration level $lev \in Lev := \mathbb{N}$,
offset $off \in Off := \mathbb{N}$
(offset and difference between usage and declaration level determine procedure stack entry)

procedure declarations: code address $ca \in PC$, declaration level $lev \in Lev$, number of local variables $loc \in Size := \mathbb{N}$

Maintaining the Symbol Table

The symbol table is maintained by the function update(D, st, l) which specifies the update of symbol table st according to declaration D (with respect to current level l):

Definition 18.8 (update function)

$$\mathrm{update} : \mathit{Dcl} \times \mathit{Tab} \times \mathit{Lev} \dashrightarrow \mathit{Tab}$$

is defined by

$$\begin{aligned}
&\mathrm{update}(D_C\ D_V\ D_P, \mathrm{st}, l)\\
&\quad := \quad \mathrm{update}(D_P, \mathrm{update}(D_V, \mathrm{update}(D_C, \mathrm{st}, l), l), l)\\
&\qquad\quad \text{if all identifiers in } D_C\ D_V\ D_P \text{ different}\\
&\mathrm{update}(\varepsilon, \mathrm{st}, l)\\
&\quad := \quad \mathrm{st}\\
&\mathrm{update}(\texttt{const}\ I_1\ \texttt{:=}\ z_1\texttt{,}\ \ldots\texttt{,}I_n\ \texttt{:=}\ z_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \quad \mathrm{st}[I_1 \mapsto (\texttt{const}, z_1), \ldots, I_n \mapsto (\texttt{const}, z_n)]\\
&\mathrm{update}(\texttt{var}\ I_1\texttt{,}\ \ldots\texttt{,}I_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \quad \mathrm{st}[I_1 \mapsto (\texttt{var}, l, 1), \ldots, I_n \mapsto (\texttt{var}, l, n)]\\
&\mathrm{update}(\texttt{proc}\ I_1\texttt{;}K_1\texttt{;}\ \ldots\ \texttt{;}I_n\texttt{;}K_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \quad \mathrm{st}[I_1 \mapsto (\texttt{proc}, a_1, l, \mathrm{size}(K_1)), \ldots, I_n \mapsto (\texttt{proc}, a_n, l, \mathrm{size}(K_n))]\\
&\qquad\quad \text{with ``fresh'' addresses } a_1, \ldots, a_n\\
&\qquad\quad \text{where } \mathrm{size}(D_C\ \texttt{var}\ I_1\texttt{,}\ \ldots\texttt{,}I_n\texttt{;}\ D_P C) := n
\end{aligned}$$

The Initial Symbol Table

By Definition 17.9, an EPL program $P = \texttt{in/out}\ I_1, \ldots, I_n; K\ . \in Pgm$ has the semantics $\mathfrak{M}[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

Given $(z_1, \ldots, z_n) \in \mathbb{Z}^n$, we choose the initial state

$$s := (1, \varepsilon, \underbrace{0 : 0 : 0 : z_1 : \ldots : z_n}_{\texttt{I/O}\text{ frame}}) \in S = PC \times DS \times PS$$

Thus the corresponding initial symbol table has n entries:

$$\mathrm{st}_{\texttt{I/O}}(I_j) := (\texttt{var}, 0, j) \qquad \text{for every } j \in [n]$$

Translation of Programs

Translation of $\texttt{in/out}\ I_1, \ldots, I_n; D\ C$.:

1. Create `MAIN` frame for executing C
2. Stop program execution after return

Definition 18.9 (Translation of programs)

The mapping

$$\text{trans} : Pgm \dashrightarrow AM$$

is defined by

$$\begin{aligned}\text{trans}(\texttt{in/out}\ I_1, \ldots, I_n; K.) := {} & 1 : \texttt{CALL}(a, \texttt{0}, \text{size}(K)); \\ & 2 : \texttt{JMP}(\texttt{0}); \\ & \text{kt}(K, \text{st}_{\texttt{I/O}}, a, 1)\end{aligned}$$

Translation of Blocks

Translation of $D\ C$:

1. Update symbol table according to D
2. Create code for procedures declared in D
   (using the updated symbol table – recursion!)
3. Create code for C (using the updated symbol table)

Definition 18.10 (Translation of blocks)

The mapping

$$\mathrm{kt} : \mathit{Block} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

is defined by

$$\begin{aligned} \mathrm{kt}(D\ C, \mathrm{st}, a, l) := {} & \mathrm{dt}(D, \mathrm{update}(D, \mathrm{st}, l), l) \\ & \mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}, l), a, l) \\ & a' : \texttt{RET;} \end{aligned}$$

Translation of Declarations

Translation of D:

- Generate code for the procedures declared in D

Definition 18.11 (Translation of declarations)

The mapping

$$\mathrm{dt} : \mathit{Dcl} \times \mathit{Tab} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

is defined by

$$\begin{aligned}
&\mathrm{dt}(D_C\ D_V\ D_P, \mathrm{st}, l) \\
&\quad := \quad \mathrm{dt}(D_P, \mathrm{st}, l) \\
&\mathrm{dt}(\varepsilon, \mathrm{st}, l) \\
&\quad := \quad \varepsilon \\
&\mathrm{dt}(\texttt{proc}\ I_1\,\texttt{;}K_1\,\texttt{;}\ \ldots\ \texttt{;}I_n\,\texttt{;}K_n\,\texttt{;}, \mathrm{st}, l) \\
&\quad := \quad \mathrm{kt}(K_1, \mathrm{st}, a_1, l+1) \\
&\qquad\qquad \vdots \\
&\qquad\quad \mathrm{kt}(K_n, \mathrm{st}, a_n, l+1) \\
&\qquad\quad \text{where } \mathrm{st}(I_j) = (\texttt{proc}, a_j, \ldots, \ldots) \text{ for every } j \in [n]
\end{aligned}$$

Translation of Commands

Definition 18.12 (Translation of commands)

The mapping

$$\mathrm{ct} : \mathit{Cmd} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

is defined by

$$\begin{aligned}
\mathrm{ct}(I \texttt{ := } A, \mathrm{st}, a, l) &:= \mathrm{at}(A, \mathrm{st}, a, l)\\
&\quad a' : \texttt{STORE(}l - \mathit{lev}\texttt{,}\mathit{off}\texttt{)};\\
&\quad \text{if } \mathrm{st}(I) = (\texttt{var}, \mathit{lev}, \mathit{off})\\
\mathrm{ct}(I\texttt{()}, \mathrm{st}, a, l) &:= a : \texttt{CALL(}\mathit{ca}\texttt{,}l - \mathit{lev}\texttt{,}\mathit{loc}\texttt{)};\\
&\quad \text{if } \mathrm{st}(I) = (\texttt{proc}, \mathit{ca}, \mathit{lev}, \mathit{loc})\\
\mathrm{ct}(C_1\texttt{;}C_2, \mathrm{st}, a, l) &:= \mathrm{ct}(C_1, \mathrm{st}, a, l)\\
&\quad \mathrm{ct}(C_2, \mathrm{st}, a', l)\\
\mathrm{ct}(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \mathrm{st}, a, l) &:= \mathrm{bt}(B, \mathrm{st}, a, l)\\
&\quad a' : \texttt{JFALSE(}a''\texttt{)};\\
&\quad \mathrm{ct}(C_1, \mathrm{st}, a' + 1, l)\\
&\quad a'' - 1 : \texttt{JMP(}a'''\texttt{)};\\
&\quad \mathrm{ct}(C_2, \mathrm{st}, a'', l)\\
&\quad a''' :\\
\mathrm{ct}(\texttt{while } B \texttt{ do } C, \mathrm{st}, a, l) &:= \mathrm{bt}(B, \mathrm{st}, a, l)\\
&\quad a' : \texttt{JFALSE(}a'' + 1\texttt{)};\\
&\quad \mathrm{ct}(C, \mathrm{st}, a' + 1, l)\\
&\quad a'' : \texttt{JMP(}a\texttt{)};
\end{aligned}$$

Translation of Boolean Expressions

Definition 18.13 (Translation of Boolean expressions)

The mapping

$$\mathrm{bt} : \mathit{BExp} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

is defined by

$$\begin{array}{rcl}
\mathrm{bt}(A_1 \texttt{ < } A_2, \mathrm{st}, a, l) & := & \mathrm{at}(A_1, \mathrm{st}, a, l) \\
& & \mathrm{at}(A_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{LT;} \\
\mathrm{bt}(\texttt{not } B, \mathrm{st}, a, l) & := & \mathrm{bt}(B, \mathrm{st}, a, l) \\
& & a' : \texttt{NOT;} \\
\mathrm{bt}(B_1 \texttt{ and } B_2, \mathrm{st}, a, l) & := & \mathrm{bt}(B_1, \mathrm{st}, a, l) \\
& & \mathrm{bt}(B_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{AND;} \\
\mathrm{bt}(B_1 \texttt{ or } B_2, \mathrm{st}, a, l) & := & \mathrm{bt}(B_1, \mathrm{st}, a, l) \\
& & \mathrm{bt}(B_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{OR;}
\end{array}$$

Translation of Arithmetic Expressions

Definition 18.14 (Translation of arithmetic expressions)

The mapping

$$\mathrm{at} : \mathit{AExp} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

is defined by

$$\begin{aligned}
\mathrm{at}(z, \mathrm{st}, a, l) &:= a : \texttt{LIT}\,(z)\,; \\
\mathrm{at}(I, \mathrm{st}, a, l) &:= \begin{cases} a : \texttt{LIT}\,(z)\,; & \text{if } \mathrm{st}(I) = (\texttt{const}, z) \\ a : \texttt{LOAD}\,(l - \mathit{lev}\,, \mathit{off}\,)\,; & \text{if } \mathrm{st}(I) = (\texttt{var}, \mathit{lev}, \mathit{off}\,) \end{cases} \\
\mathrm{at}(A_1 \texttt{ + } A_2, \mathrm{st}, a, l) &:= \mathrm{at}(A_1, \mathrm{st}, a, l) \\
&\quad\ \mathrm{at}(A_2, \mathrm{st}, a', l) \\
&\quad\ a'' : \texttt{ADD}\,;
\end{aligned}$$