Compiler Construction

Lecture 19: Code Generation II (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/cc10/

Winter semester 2010/11

Outline

Syntax of EPL

Definition (Syntax of EPL)

The syntax of EPL is defined as follows:

$$
\begin{array}{lll}
\mathbb{Z}: & z & (*\ z \text{ is an integer } *) \\
Ide: & I & (*\ I \text{ is an identifier } *) \\
AExp: & A ::= z \mid I \mid A_1 \texttt{ + } A_2 \mid \ldots & \\
BExp: & B ::= A_1 \texttt{ < } A_2 \mid \texttt{not } B \mid B_1 \texttt{ and } B_2 \mid B_1 \texttt{ or } B_2 & \\
Cmd: & C ::= I \texttt{ := } A \mid C_1\texttt{;}C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid & \\
 & \quad \texttt{while } B \texttt{ do } C \mid I\texttt{()} & \\
Dcl: & D ::= D_C\ D_V\ D_P & \\
 & D_C ::= \varepsilon \mid \texttt{const } I_1 \texttt{ := } z_1\texttt{,}\ldots\texttt{,}I_n \texttt{ := } z_n\texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var } I_1\texttt{,}\ldots\texttt{,}I_n\texttt{;} & \\
 & D_P ::= \varepsilon \mid \texttt{proc } I_1\texttt{;}K_1\texttt{;}\ldots\texttt{;}I_n\texttt{;}K_n\texttt{;} & \\
Blk: & K ::= D\ C & \\
Pgm: & P ::= \texttt{in/out } I_1\texttt{,}\ldots\texttt{,}I_n\texttt{;}K\texttt{.} &
\end{array}
$$

Another Example: Factorial Function

Example 19.1 (Factorial function)

```
in/out x;
  var y;
  proc F;
    if x > 1 then
      y := y * x;
      x := x - 1;
      F()
  y := 1;
  F();
  x := y.
```

Dynamic Semantics of EPL

(omitting the details)

- To "run" a program, execute the main block in the state which is given by the input values
- Effect of statement = modification of state
  - assignment I `:=` A: update of I by value of A
  - composition C_1`;`C_2: sequential execution
  - branching `if` B `then` C_1 `else` C_2: test of B, followed by jump to respective brnach
  - iteration `while` B `do` C: execution of C as long as B is true
  - call I`()`: transfer control to body of I and return to subsequent statement afterwards
- Consequently, an EPL program $P =$ `in/out` I_1`,` $\ldots$ `,` I_n`;` K`.` $\in Pgm$ has as semantics a function

$$\mathfrak{M}[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$$

Outline

Definition 19.2 (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 19.3 (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 19.4 (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
  $(sl = dl = ra = 0)$

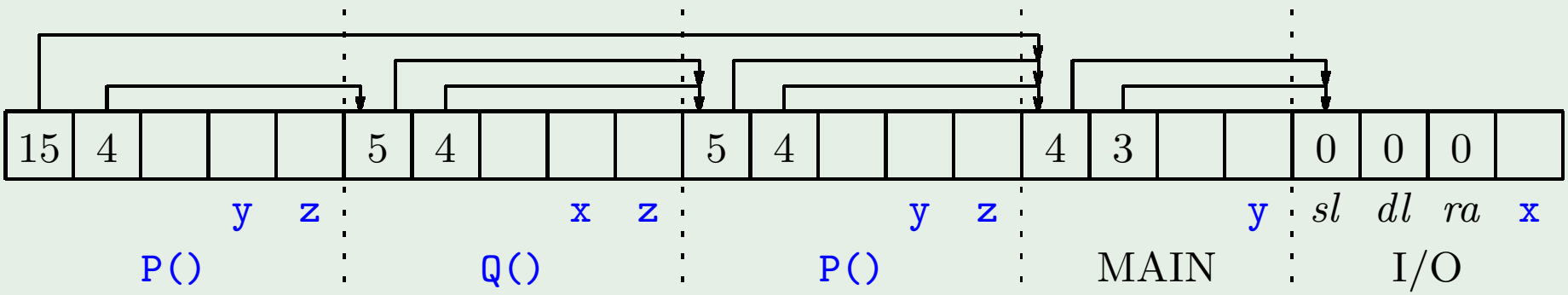
  MAIN frame: for keeping values of top-level block
  $(sl = dl =$ I/O frame$)$

Structure of Procedure Stack III

Example 19.5 (cf. Example 18.8)

```
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

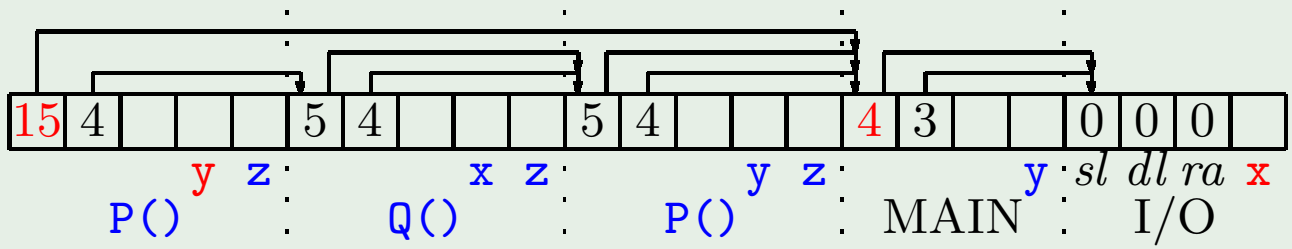
Observation:

- 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 19.6 (cf. Example 19.5)

```
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 19.7 (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 19.8 (cf. Example 19.6)

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 19.9 (Semantics of procedure instructions)

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

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

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

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 19.10 (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 19.11 (Semantics of AM programs)

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

$$P = 1 : O_1; \ldots; k : O_k$$

The set of all AM programs is denoted by AM.

The semantics of AM programs is determined by

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

with

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