Compiler Construction

Lecture 16: Code Generation II (The Translation)

Winter Semester 2018/19

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1819/cc/

Recap: Intermediate Code

Syntax of EPL

Definition (Syntax of EPL)

The (ambiguous) syntax of EPL is defined as follows:

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

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 $s = (pc, d, p) \in S$ is given by

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

Recap: Intermediate Code

AM Instructions

Definition (AM instructions)

The set of AM instructions is divided into

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

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

branching 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`)` ($z \in \mathbb{Z}$)

Structure of Procedure Stack I

The semantics of procedure and transfer instructions requires a particular structure of the procedure stack $p \in PS$: it 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

- used to access non-local variables

dynamic link dl: points to previous frame (i.e., of calling procedure)

- used to remove topmost frame after termination of procedure call

return address ra: program counter after termination of procedure call

- 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

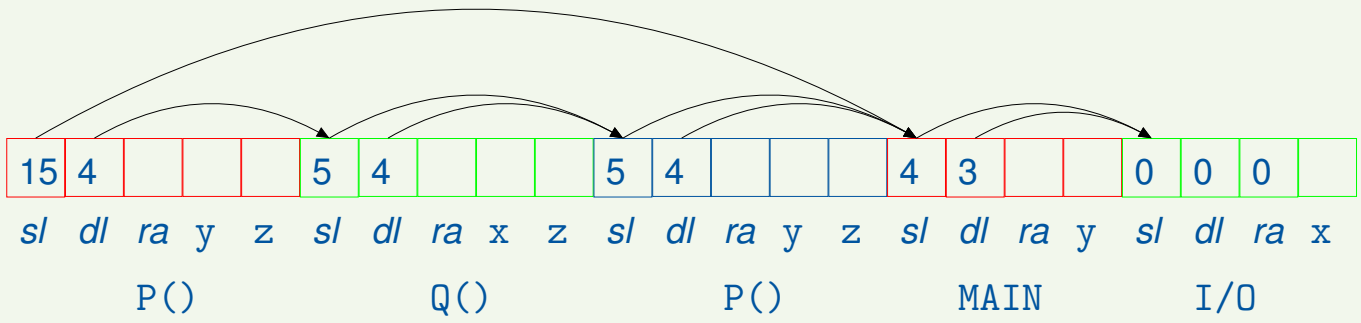
  ($sl = dl =$ I/O frame)

Structure of Procedure Stack III

Example 16.1 (cf. Example 15.4)

```
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

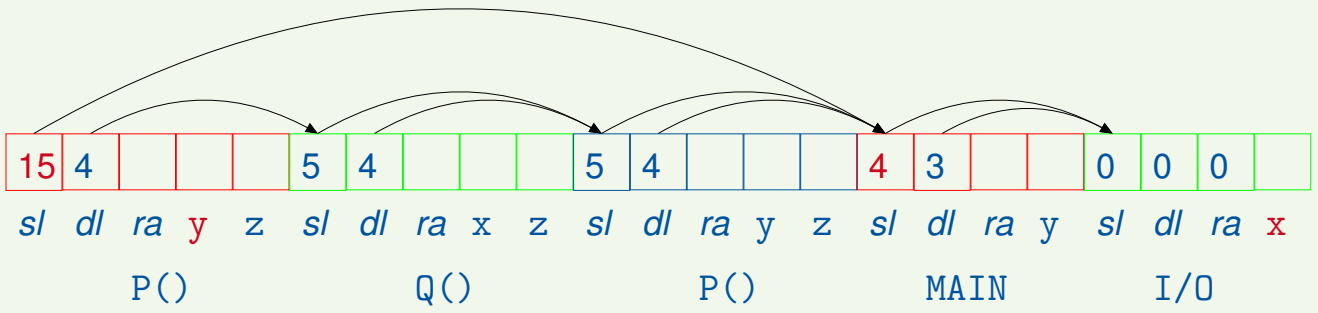
Observation:

- 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 16.2 (cf. Example 16.1)

```
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* $= 0$

Display Technique

- Optimisation technique to replace static links
- Implementation:
  - display = array of pointers to frames: *disp*
  - size of array = maximum block nesting depth of program
  - $disp[i]$ points to frame associated with the current scope at nesting depth i
- Usage: to access a non-local variable declared at nesting depth i,
  1. go to frame pointed to by $disp[i]$
  2. access variable via offset in this frame

  Advantage: constant-time access (exactly two steps regardless of level difference)
- Maintenance:
  - use global display for current procedure activation
  - caller saves display in its frame, and restores it when callee returns
  - display for callee constructed using techniques for static link handling, using static scoping

Semantics of Procedure and Transfer Instructions

The base Function

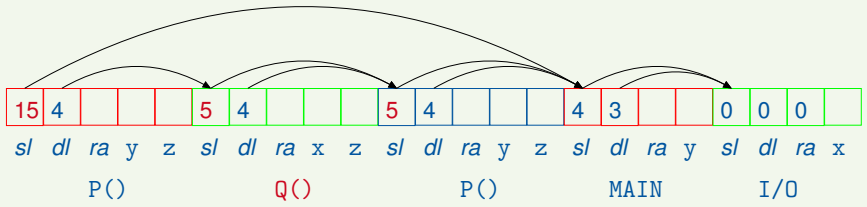
Computes static link information upon procedure call and variable access, given procedure stack and level difference

Definition 16.3 (base function)

The function $\mathrm{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$ is given by

$$\mathrm{base}(p, 0) := 1$$
$$\mathrm{base}(p, dif + 1) := \mathrm{base}(p, dif) + p.\mathrm{base}(p, dif)$$

Example 16.4 (cf. Example 16.2)



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

$$\mathrm{base}(p, 0) = 1$$
$$\Longrightarrow \mathrm{base}(p, 1) = 1 + p.1 = 6$$
$$\Longrightarrow \mathrm{base}(p, 2) = 6 + p.6 = 11$$
$$\Longrightarrow sl = \mathrm{base}(p, 2) + \underbrace{2}_{dl,ra} + \underbrace{2}_{y,z} = 15$$

Semantics of Procedure Instructions

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

  creates new frame and jumps to given address (= starting address of procedure)
- $\texttt{RET}$ removes the topmost frame and returns to the calling site

Definition 16.5 (Semantics of procedure instructions)

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

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

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

Semantics of Transfer Instructions

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

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

Definition 16.6 (Semantics of transfer instructions)

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

$$[\![\texttt{LOAD}(\mathit{dif}, \mathit{off})]\!](pc, d, p) := (pc + 1, d : p.(\mathrm{base}(p, \mathit{dif}) + \mathit{off} + 2), p)$$
$$[\![\texttt{STORE}(\mathit{dif}, \mathit{off})]\!](pc, d : z, p) := (pc + 1, d, p[\mathrm{base}(p, \mathit{dif}) + \mathit{off} + 2 \mapsto z])$$
$$[\![\texttt{LIT}(z)]\!](pc, d, p) := (pc + 1, d : z, p)$$

AM Programs and Their Semantics

Definition 16.7 (Semantics of AM programs)

An AM program is a sequence of $k \geq 1$ labelled 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

$$[\![P]\!](pc, d, p) := \begin{cases} [\![P]\!]([\![O_{pc}]\!](pc, d, p)) & \text{if } [\![O_{pc}]\!](pc, d, p) \text{ defined} \\ (pc, d, p) & \text{otherwise} \end{cases}$$

(in particular, $[\![P]\!](pc, d, p) = (pc, d, p)$ if $pc \notin [k]$)

Structure of Symbol Table

Goal: define translation mapping $\mathrm{trans} : \mathit{Pgm} \dashrightarrow \mathit{AM}$

The translation employs a symbol table:

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

whose entries are created by declarations:

- constant declarations: $(\texttt{const}, z)$
  - value $z \in \mathbb{Z}$
- variable declarations: $(\texttt{var}, \mathit{lev}, \mathit{off})$
  - declaration level $\mathit{lev} \in \mathit{Lev} := \mathbb{N}$ (0 $\cong$ I/O, 1 $\cong$ MAIN, ...)
  - offset $\mathit{off} \in \mathit{Off} := \mathbb{N}$
  - offset and difference between usage and declaration level determine procedure stack entry
- procedure declarations: $(\texttt{proc}, \mathit{ca}, \mathit{lev}, \mathit{loc})$
  - code address $\mathit{ca} \in \mathit{PC}$
  - declaration level $\mathit{lev} \in \mathit{Lev}$
  - number of local variables $\mathit{loc} \in \mathit{Size} := \mathbb{N}$

Maintaining the Symbol Table

Function $\mathrm{update}(D, \mathrm{st}, \mathit{lev})$ specifies update of symbol table st according to declaration D (with respect to current level lev):

Definition 16.8 (update function)

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

$$\mathrm{update}(D_C\ D_V\ D_P, \mathrm{st}, \mathit{lev}) := \mathrm{update}(D_P, \mathrm{update}(D_V, \mathrm{update}(D_C, \mathrm{st}, \mathit{lev}), \mathit{lev}), \mathit{lev})$$
if all identifiers in $D_C\ D_V\ D_P$ different

$$\mathrm{update}(\varepsilon, \mathrm{st}, \mathit{lev}) := \mathrm{st}$$

$$\mathrm{update}(\texttt{const}\ I_1 := z_1, \ldots, I_n := z_n;, \mathrm{st}, \mathit{lev}) := \mathrm{st}[I_1 \mapsto (\texttt{const}, z_1), \ldots, I_n \mapsto (\texttt{const}, z_n)]$$

$$\mathrm{update}(\texttt{var}\ I_1, \ldots, I_n;, \mathrm{st}, \mathit{lev}) := \mathrm{st}[I_1 \mapsto (\texttt{var}, \mathit{lev}, 1), \ldots, I_n \mapsto (\texttt{var}, \mathit{lev}, n)]$$

$$\mathrm{update}(\texttt{proc}\ I_1; K_1; \ldots; \texttt{proc}\ I_n; K_n;, \mathrm{st}, \mathit{lev}) := \mathrm{st}[I_1 \mapsto (\texttt{proc}, a_1, \mathit{lev}, \mathrm{size}(K_1)), \ldots, I_n \mapsto (\texttt{proc}, a_n, \mathit{lev}, \mathrm{size}(K_n))]$$
with "fresh" addresses $a_1, \ldots, a_n$ where $\mathrm{size}(D_C\ \texttt{var}\ I_1, \ldots, I_n;\ D_P C) := n$

The Initial Symbol Table

Reminder: an EPL program $P = \texttt{in/out}\ I_1, \ldots, I_n; K\,. \in Pgm$ has a semantics of type $\mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

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

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

Thus the corresponding initial symbol table has n entries:

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

Translation of Programs

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

1. Create MAIN frame for executing C (remember: $\texttt{CALL}(ca, dif, loc)$)
2. Stop program execution after return

Definition 16.9 (Translation of programs)

The mapping

$$\mathrm{trans} : \mathit{Pgm} \dashrightarrow \mathit{AM}$$

is defined by

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

with “fresh” address a

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 to handle recursion)
3. Create code for C (using the updated symbol table)

Definition 16.10 (Translation of blocks)

The mapping

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

("block translation") is defined by

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

with "fresh" address a'

Translation of Declarations

Translation of D: generate code for the procedures declared in D

Definition 16.11 (Translation of declarations)

The mapping

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

("declaration translation") is defined by

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

Translation of Commands

Definition 16.12 (Translation of commands)

The mapping

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

("command translation") is defined by

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

Translation of Boolean Expressions

Definition 16.13 (Translation of Boolean expressions)

The mapping

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

("Boolean expression translation") is defined by

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

Translation of Arithmetic Expressions

Definition 16.14 (Translation of arithmetic expressions)

The mapping

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

("arithmetic expression translation") is defined by

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

A Translation Example

Example: Factorial Function I

Example 16.15 (Factorial function; cf. Example 15.3)

Source code:

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

Intermediate code:

$\mathrm{trans}(\texttt{in/out } x;K.)$

$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$
$\mathrm{kt}(K,\mathrm{st}_{I/O},a_0,1)$
$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$
$\mathrm{dt}(D,\mathrm{update}(D,\mathrm{st}_{I/O},1),1)$
$\mathrm{ct}(C,\mathrm{update}(D,\mathrm{st}_{I/O},1),a_0,1)$
$a_2 : \texttt{RET};$
$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$
$\mathrm{dt}(D,\mathrm{st}',1)$
$\mathrm{ct}(C,\mathrm{st}',a_0,1)$
$a_2 : \texttt{RET};$
$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$
$\mathrm{kt}(K_F,\mathrm{st}',a_1,2)$
$\mathrm{ct}(C,\mathrm{st}',a_0,1)$
$a_2 : \texttt{RET};$
$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$
$\mathrm{ct}(C_F,\mathrm{st}',a_1,2)$
$a_3 : \texttt{RET};$
$\mathrm{ct}(C,\mathrm{st}',a_0,1)$
$a_2 : \texttt{RET};$
$1 : \texttt{CALL}(a_0,0,1);$
$2 : \texttt{JMP}(0);$

$\mathrm{trans}(\texttt{in/out } I_1,\ldots,I_n;K.) :=$
$1 : \texttt{CALL}(a,0,\mathrm{size}(K));\ \mathrm{kt}(D\ C,\mathrm{st},a,lev) :=$
$2 : \texttt{JMP}(0);$
$\mathrm{kt}(K,\mathrm{st}_{I/O},a,1)$
$\mathrm{dt}(D,\mathrm{update}(D,\mathrm{st},lev),lev)$
$\mathrm{ct}(C,\mathrm{update}(D,\mathrm{st},lev),a,lev)$
$a' : \texttt{RET};$

$\mathrm{kt}(K_1,\mathrm{st},a_1,lev+1)$
$\vdots$
$\mathrm{kt}(K_n,\mathrm{st},a_n,lev+1)$

where $\mathrm{st}(I_j) = (\texttt{proc}, a_j, \ldots, \ldots)$ for every $j \in [n]$

$\mathrm{bt}(B,\mathrm{st},a,lev)$
$a' : \texttt{JFALSE}(a'');$
$\mathrm{ct}(C_1,\mathrm{st},a'+1,lev)$

$\mathrm{update}(\texttt{var } I_1,\ldots,I_n;,\mathrm{st},lev) :=$
$\mathrm{st}[I_1 \mapsto (\texttt{var},lev,1),\ldots,I_n \mapsto (\texttt{var},lev,n)]$
$\mathrm{update}(\texttt{proc } I_1;K_1;\ldots;\texttt{proc } I_n;K_n;,\mathrm{st},lev) :=$
$\mathrm{st}[I_1 \mapsto (\texttt{proc},a_1,lev,\mathrm{size}(K_1)),\ldots,I_n \mapsto (\texttt{proc},a_n,lev,\mathrm{size}(K_n))]$

$\mathrm{ct}(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2,\mathrm{st},a,lev) :=$

$\mathrm{bt}(A_1 > A_2,\mathrm{st},a,lev) :=$

$\mathrm{dt}(\texttt{proc } I_1;K_1;\ldots$

Example: Factorial Function II

Example 16.15 (Factorial function; continued)

Code with symbolic addresses:

```
 1 : CALL(a0,0,1);
 2 : JMP(0);
a1 : LOAD(2,1);
     LIT(1);
     GT;
a4 : JFALSE(a3);
     LOAD(1,1);
     LOAD(2,1);
     MULT;
     STORE(1,1);
     LOAD(2,1);
     LIT(1);
     SUB;
     STORE(2,1);
     CALL(a1,1,0);
a3 : RET;
a0 : LIT(1);
     STORE(0,1);
     CALL(a1,0,0);
     LOAD(0,1);
     STORE(1,1);
a2 : RET;
```

Linearised ($a_0 = 17, a_1 = 3, a_2 = 22, a_3 = 16, a_4 = 6$):

```
 1 : CALL(17,0,1);
 2 : JMP(0);
 3 : LOAD(2,1);
 4 : LIT(1);
 5 : GT;
 6 : JFALSE(16);
 7 : LOAD(1,1);
 8 : LOAD(2,1);
 9 : MULT;
10 : STORE(1,1);
11 : LOAD(2,1);
12 : LIT(1);
13 : SUB;
14 : STORE(2,1);
15 : CALL(3,1,0);
16 : RET;
17 : LIT(1);
18 : STORE(0,1);
19 : CALL(3,0,0);
20 : LOAD(0,1);
21 : STORE(1,1);
22 : RET;
```

A Translation Example

Example: Factorial Function III

Example 16.15 (Factorial function; continued)

Execution for $x = 2$:

```
 1 : CALL(17,0,1);
 2 : JMP(0);
 3 : LOAD(2,1);
 4 : LIT(1);
 5 : GT;
 6 : JFALSE(16);
 7 : LOAD(1,1);
 8 : LOAD(2,1);
 9 : MULT;
10 : STORE(1,1);
11 : LOAD(2,1);
12 : LIT(1);
13 : SUB;
14 : STORE(2,1);
15 : CALL(3,1,0);
16 : RET;
17 : LIT(1);
18 : STORE(0,1);
19 : CALL(3,0,0);
20 : LOAD(0,1);
21 : STORE(1,1);
22 : RET;
```

PC	*DS*	*PS*
1	ε	0 : 0 : 0 : 2
17	ε	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
18	1	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
19	ε	4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
3	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
4	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
5	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
6	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
7	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
8	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
9	1 : 2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
10	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
11	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
12	2	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
13	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
14	1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
15	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
3	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
4	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
5	1 : 1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
6	0	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
20	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
21	2	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
22	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
2	ε	0 : 0 : 0 : 2
0	ε	0 : 0 : 0 : 2

Source code:

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

Correctness of the Translation

Theorem 16.16 (Correctness of translation)

For every $P \in Pgm$, $n \in \mathbb{N}$, and $(z_1, \ldots, z_n), (z'_1, \ldots, z'_n) \in \mathbb{Z}^n$:

$$\begin{aligned} & [\![P]\!](z_1, \ldots, z_n) = (z'_1, \ldots, z'_n) \\ \iff & [\![\mathrm{trans}(P)]\!](1, \varepsilon, 0 : 0 : 0 : z_1 : \ldots : z_n) = (0, \varepsilon, 0 : 0 : 0 : z'_1 : \ldots : z'_n) \end{aligned}$$

Proof.

see M. Mohnen: *A Compiler Correctness Proof for the Static Link Technique by means of Evolving Algebras*, Fundamenta Informaticae 29(3), 1997, pp. 257–303 □