Compiler Construction

Lecture 21: Code Generation VI
(Translation Example & Static Data Structures)

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

Correction: update Function

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

Definition (update function)

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

is defined by

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

Outline

Extended Syntax of EPL

Definition (Extended syntax of EPL)

The extended syntax of EPL is defined as follows:

$$\vdots$$

$$Cmd: \quad C ::= I := A \mid C_1 ; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{while } B \texttt{ do } C \mid I(\underbrace{A_1, \ldots, A_p ; V_1, \ldots, V_q}_{\text{actual value and ref. parameters}})$$

$$\begin{aligned} Dcl: \quad D &::= D_C\ D_V\ D_P \\ D_C &::= \varepsilon \mid \texttt{const } I_1 := z_1, \ldots, I_n := z_n ; \\ D_V &::= \varepsilon \mid \texttt{var } I_1, \ldots, I_n ; \\ D_P &::= \varepsilon \mid \\ &\quad \texttt{proc } I(\underbrace{I_1, \ldots, I_p ; \texttt{ var } J_1, \ldots, J_q}_{\text{formal value and ref. parameters}}) ; K ; \texttt{ proc} \ldots \end{aligned}$$

$$\vdots$$

where $I_k, J_k, V_k \in Ide$

Definition (Extended abstract machine for EPL)

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

$$S := PC \times SP \times FP \times IR \times RS$$

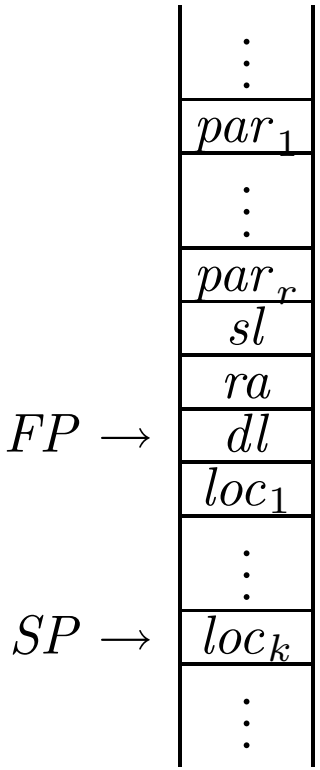
with

- the program counter $PC := \mathbb{N}$,
- the stack pointer $SP := \mathbb{N}$,
- the frame pointer $FP := \mathbb{N}$,
- the index register $IR := \mathbb{N}$, and
- the runtime stack $RS := (\mathbb{N} \to \mathbb{Z})$.

Characteristics:

- Closer to "real" machine (less powerful instructions)
- Runtime stack RS combines data stack DS and procedure stack PS
- Absolute addressing of stack entries
- Stack pointer SP points to top of stack
- Frame pointer FP points to (dynamic link field of) topmost frame
- Index register IR implements dereferencing of static links

Structure of Frames



↓ increasing stack addresses

actual value and reference parameters

static link to declaration environment

return address

dynamic link to previous frame

local variables

Successive construction:

- by calling procedure:
  1. Computation of actual parameters par_i
  2. Computation of static link sl using index register IR
  3. Jump to called procedure, setting return address ra
- by called procedure ("entry code"):
  4. Store pointer to previous frame as dynamic link dl
  5. Allocate memory for local variables loc_j

Translation of Extended EPL into AM Programs

Goal: modification of translation mapping

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

taking into account

- Procedures with parameters
- Modified AM instruction set

Modifying the Symbol Table

$$
\begin{aligned}
Tab := \{\mathrm{st} \mid \mathrm{st} : Ide \dashrightarrow\ & (\{\texttt{const}\} \times \mathbb{Z}) \\
& \cup (\{\texttt{var}\} \times Lev \times Off) \\
& \quad \% \text{ local variables and value parameters} \\
& \cup (\{\texttt{proc}\} \times PC \times Lev \times Size)\} \\
& \cup (\{\texttt{rpar}\} \times Lev \times Off)\} \\
& \quad \% \text{ reference parameters}
\end{aligned}
$$

Position of $FP \implies$ negative offsets possible $\implies Off := \mathbb{Z}$

Initial symbol table for $P = \texttt{in/out}\ I_1, \ldots, I_n; K\,. \in Pgm$:

$$\mathrm{st}_{\texttt{I/O}}(I_j) := (\texttt{var}, 0, j - n - 3) \qquad \text{for every } j \in [n]$$

update function: as before
(processing of procedure parameters by dt)

Outline

Translation Example I

Example 21.1 (Factorial function)

$P =$
```
in/out x,y;
proc F(x;var y);
  if x>1 then
    y:=y*x;
    F(x-1;y);
y:=1;
F(x,y)
```
$C_{\mathtt{F}}$, D, C, K

$\mathrm{trans}(P) =$
```
1 : PUSH FP; % static link
2 : CALL a0;
3 : JMP 0; % STOP
```
$\mathrm{kt}(K, \mathrm{st}_{\mathtt{I/O}}, a_0, 1, 0)$

where

$\mathrm{st}_{\mathtt{I/O}} = [\mathtt{x} \mapsto (\mathtt{var}, 0, -4), \mathtt{y} \mapsto (\mathtt{var}, 0, -3)]$

$\mathrm{kt}(K, \mathrm{st}_{\mathtt{I/O}}, a_0, 1, 0)$
$= \mathrm{dt}(D, \mathrm{st}', 1)$
```
a0 : PUSH FP; % dynamic link
LOAD FP,SP; % set FP
ADD SP,size(K); % local vars
```
($\mathrm{size}(K) = 0$)
$\mathrm{ct}(C, \mathrm{st}', a_0 + 3, 1)$
```
LOAD SP,FP;
POP FP; % reset FP
RET 1; % jump back
```

where st'
$= \mathrm{update}(D, \mathrm{st}_{\mathtt{I/O}}, 1)$
$= \mathrm{st}_{\mathtt{I/O}}[\mathtt{F} \mapsto (\mathtt{proc}, a_1, 1, 0)]$

$\mathrm{dt}(D, \mathrm{st}', 1)$
$= \mathrm{kt}(C_{\mathtt{F}}, \mathrm{st}'', a_1, 2, 2)$
```
a1 : PUSH FP; % dynamic link
LOAD FP,SP; % set FP
ADD SP,0; % local vars
```
$\mathrm{ct}(C_{\mathtt{F}}, \mathrm{st}'', a_1 + 3, 2)$
```
LOAD SP,FP;
POP FP; % reset FP
RET 3; % jump back
```

where $\mathrm{st}'' =$
$\mathrm{st}'[\mathtt{x} \mapsto (\mathtt{var}, 2, -4), \mathtt{y} \mapsto (\mathtt{rpar}, 2, -3)]$

$\mathrm{ct}(C_{\mathtt{F}}, \mathrm{st}'', a_1 + 3, 2)$
$= \mathrm{bt}(\mathtt{x>1}, \mathrm{st}'', a_1 + 3, 2)$
```
JFALSE a2;
```
$\mathrm{ct}(\mathtt{y:=y*x;\ F(x-1;y)}, \mathrm{st}'', a', 2)$
```
a2 :
```

$\mathrm{bt}(\mathtt{x>1}, \mathrm{st}'', a_1 + 3, 2)$
$=$
```
PUSH <FP-4>; % x
PUSH 1;
GT;
```

Translation Example II

Example 21.1 (Factorial function; continued)

ct(`y:=y*x; F(x-1;y)`$, \mathrm{st}'', a', 2)$
$=$ ct(`y:=y*x`$, \mathrm{st}'', a', 2)$
ct(`F(x-1;y)`$, \mathrm{st}'', a'', 2)$
ct(`y:=y*x`$, \mathrm{st}'', a', 2)$
$=$ at(`y*x`$, \mathrm{st}'', a', 2)$
`LOAD IR,<FP-3>;` % adr(y)
`POP <IR>;` % assign
at(`y*x`$, \mathrm{st}'', a', 2)$
$=$ `LOAD IR,<FP-3>;` % adr(y)
`PUSH <IR>;` % y
`PUSH <FP-4>;` % x
`MULT;`
ct(`F(x-1;y)`$, \mathrm{st}'', a'', 2)$
$=$ at(`x-1`$, \mathrm{st}'', a'', 2)$
`PUSH <FP-3>;` % adr(y)
`LOAD IR,<FP-2>;`
`PUSH IR;` % static link
`CALL` a_1`;`

at(`x-1`$, \mathrm{st}'', a'', 2)$
$=$ `PUSH <FP-4>;` % x
`PUSH 1;`
`SUB;`
$\mathrm{ct}(C, \mathrm{st}', a_0 + 3, 1)$
$=$ ct(`y:=1`$, \mathrm{st}', a_0 + 3, 1)$
ct(`F(x;y)`$, \mathrm{st}', a', 1)$
ct(`y:=1`$, \mathrm{st}', a_0 + 3, 1)$
$=$ `PUSH 1;`
`LOAD IR,<FP-2>;` % adr(y)
`POP <IR-3>;` % assign
ct(`F(x;y)`$, \mathrm{st}', a', 1)$
$=$ at(`x`$, \mathrm{st}', a', 1)$
`LOAD IR,<FP-2>;`
`PUSH IR-3;` % adr(y)
`PUSH FP;` % static link
`CALL` a_1`;`
at(`x`$, \mathrm{st}', a', 1)$
$=$ `LOAD IR,<FP-2>;`
`PUSH <IR-4>`

Translation Example III

Example 21.1 (Factorial function; continued)

Altogether:

```
 1 : PUSH FP;
 2 : CALL 27;
 3 : JMP 0;
 4 : PUSH FP;            % entry F
 5 : LOAD FP,SP;
 6 : ADD SP,0;
 7 : PUSH <FP-4>;        % x>1
 8 : PUSH 1;
 9 : GT;
10 : JFALSE 24;
11 : LOAD IR,<FP-3>;     % y:=y*x
12 : PUSH <IR>;
13 : PUSH <FP-4>;
14 : MULT;
15 : LOAD IR,<FP-3>;
16 : POP <IR>;
17 : PUSH <FP-4>;        % F(x-1;y)
18 : PUSH 1;
19 : SUB;
20 : PUSH <FP-3>;
21 : LOAD IR,<FP-2>;
22 : PUSH IR;
23 : CALL 4;
24 : LOAD SP,FP;         % exit F
25 : POP FP;
26 : RET 3;
27 : PUSH FP;            % entry MAIN
28 : LOAD FP;SP;
29 : ADD SP,0;
30 : PUSH 1;             % y:=1
31 : LOAD IR,<FP-2>;
32 : POP <IR-3>;
33 : LOAD IR,<FP-2>;     % F(x;y)
34 : PUSH <IR-4>;
35 : LOAD IR,<FP-2>;
36 : PUSH IR-3;
37 : PUSH FP;
38 : CALL 4;
39 : LOAD SP,FP;         % exit MAIN
40 : POP FP;
41 : RET 1;
```

For x = 1, y = 0 (bottom = $p(1)$, top = SP, FP, IR):

PC	RS
1	1 : 0 : 0 : 0 : 0
2	1 : 0 : 0 : 0 : 0 : 5
27	1 : 0 : 0 : 0 : 0 : 5 : 3
28	1 : 0 : 0 : 0 : 0 : 5 : 3 : 5
29	1 : 0 : 0 : 0 : 0 : 5 : 3 : 5
30	1 : 0 : 0 : 0 : 0 : 5 : 3 : 5
31	1 : 0 : 0 : 0 : 0 : 5 : 3 : 5 : 1
32	1 : 0 : 0 : 0 : 0 : 5 : 3 : 5 : 1
33	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5
34	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5
35	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1
36	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1
37	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2
38	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8
4	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39
5	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8
6	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8
7	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8
8	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 1
9	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 1 : 1
10	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 0
24	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8
25	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8
26	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39
39	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5
40	1 : 1 : 0 : 0 : 0 : 5 : 3 : 5
41	1 : 1 : 0 : 0 : 0 : 5 : 3
3	1 : 1 : 0 : 0 : 0
0	1 : 1 : 0 : 0 : 0

Outline

Alternative Implementation of Static Links

- **Alternative implementation** of static link dereferencing: display technique
- Display directly returns frame pointer for given level difference
- Usually organized as array
- Possible implementations:

  Local displays: called procedure maintains display array in its frame

  Global displays: frame pointer stored in global Static Link Array (SLA)
- For details see Aho/Lam/Sethi/Ullman: *Compilers: Principles, Techniques, and Tools*, 2nd ed., p. 449 ff)

Outline

Translation of Data Structures

Source code: data structures = arrays, records, lists, trees, ...
$\Longrightarrow$ structured state space, variables with components

Abstract machine: linear memory structure, cells for storing atomic data

Translation: mapping of structured state space to linear memory (= address computation)

- static data structures: memory reqirements known at compile time
- dynamic data structures: memory reqirements runtime dependent
  $\Longrightarrow$ heap, pointers, garbage collection, ...

First step:

- static data structures (arrays and records)
- inductive type definitions
- no procedures (for simplification; "orthogonal" extension)

Outline

Modified Syntax of EPL

Definition 21.2 (Modified syntax of EPL)

The modified syntax of EPL is defined as follows (where $n \geq 1$):

$$
\begin{array}{lll}
\mathbb{Z}: & z & (*\ z \text{ is an integer } *) \\
\mathbb{B}: & b ::= \texttt{true} \mid \texttt{false} & \\
\mathbb{R}: & r & (*\ r \text{ is a real number } *) \\
\mathit{Con}: & c ::= z \mid b \mid r & \\
\mathit{Ide}: & I & (*\ I \text{ is an identifier } *) \\
\mathit{Type}: & T ::= \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid I \mid \texttt{array}[z_1 \texttt{..} z_2]\ \texttt{of}\ T \mid & \\
 & \quad \texttt{record}\ I_1 \texttt{:} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:} T_n\ \texttt{end} & \\
\mathit{Var}: & V ::= I \mid V\,[E] \mid V\,.\,I & \\
\mathit{Exp}: & E ::= c \mid V \mid E_1 \texttt{ + } E_2 \mid E_1 \texttt{ < } E_2 \mid E_1\ \texttt{and}\ E_2 \mid \ldots & \\
\mathit{Cmd}: & C ::= V \texttt{:=} E \mid C_1 \texttt{;} C_2 \mid & \\
 & \quad \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ E\ \texttt{do}\ C & \\
\mathit{Dcl}: & D ::= D_C\ D_T\ D_V & \\
 & D_C ::= \varepsilon \mid \texttt{const}\ I_1 \texttt{:=} c_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} c_n \texttt{;} & \\
 & D_T ::= \varepsilon \mid \texttt{type}\ I_1 \texttt{:=} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} T_n \texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var}\ I_1 \texttt{:} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:} T_n \texttt{;} & \\
\mathit{Pgm}: & P ::= D\ C &
\end{array}
$$

Static Semantics I

- All identifiers in a declaration D have to be different.
- In $T = \texttt{record}\ I_1 \texttt{:} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:} T_n\ \texttt{end}$, all selectors I_j must be different.
- In $T = \texttt{array[}z_1 \texttt{..} z_2\texttt{]}\ \texttt{of}\ T$, $z_1 \leq z_2$.
- Type definitions must not be recursive:
  if $D_T = \texttt{type}\ I_1 \texttt{:=} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} T_n \texttt{;}$ and identifier I occurs in T_j, then $I \in \{I_1, \ldots, I_{j-1}\}$.
- The type identifiers used in in a variable declaration D_V must be declared.
- Every identifier used in a command C must be declared in D (as a constant or variable).
- Variables in expressions and assignments have a base type (`bool`/`int`/`real`; possibly via type identifiers).

Static Semantics II

- Array indices must have type `int`.
- Execution conditions (`while`) and branching expressions (`if`) must have type `bool`.
- The types of the left-hand side and of the right-hand side types of an assignment must be compatible.
- Type compatibility: $\mathbb{Z} \subseteq \mathbb{R}$ in mathematics, but not on computers (different representation)

  $\Longrightarrow$ type casts

  weak typing: implicit casting by compiler (`2.5 + 1`, `1 + "42"`)

  $\Longrightarrow$ risc of undetected "real" errors;
  for programming-in-the-small (script languages)

  strong typing: explicit casting by programmer

  $\Longrightarrow$ enhanced software reliability;
  for programming-in-the-large
- Instantiation of operators/functions/procedures/... for different parameter types: polymorphism or overloading

  `+` : `int` $\times$ `int` $\rightarrow$ `int` `+` : `real` $\times$ `real` $\rightarrow$ `real`

Outline

The Modified Abstract Machine AM

Since (recursive) procedures are no longer supported, a procedure stack is not required anymore.

Definition 21.3 (Modified abstract machine for EPL)

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

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

with

- the program counter $PC := \mathbb{N}$,
- the data stack $DS := \mathbb{R}^*$, and
- the main storage $MS := \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{R}\}$.

New AM Instructions

Definition 21.4 (New AM instructions)

- Procedure instructions are no longer needed.
- Transfer instructions (`LOAD(`dif`,`off`)`, `STORE(`dif`,`off`)`) are replaced by the following instructions with the respective semantics $[\![O]\!] : S \dashrightarrow S$:

$$\begin{aligned} [\![\texttt{LOAD}]\!](a, d : n, \sigma) &:= (a+1, d : \sigma(n), \sigma) \\ &\quad \text{if } n \in \mathbb{N} \\ [\![\texttt{STORE}]\!](a, d : n : r, \sigma) &:= (a+1, d, \sigma[n \mapsto r]) \\ &\quad \text{if } n \in \mathbb{N} \end{aligned}$$

- Moreover the following instruction for checking array bounds is introduced:

$$[\![\texttt{CAB}(z_1, z_2)]\!](a, d : z, \sigma) := \begin{cases} (a+1, d : z, \sigma) & \text{if } z \in \{z_1, \ldots, z_2\} \\ (0, d : \underbrace{\texttt{RTE}}_{\text{runtime error}}, \sigma) & \text{otherwise} \end{cases}$$