Compiler Construction

Lecture 22: Code Generation VII (Static & Dynamic 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

Modified Syntax of EPL

Definition (Modified syntax of EPL)

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

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

The Modified Abstract Machine AM

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

Definition (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 (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:

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

Outline

Modifying the Symbol Table

$$\begin{aligned} Tab := \{\mathrm{st} \mid \mathrm{st} : Ide \dashrightarrow\ & (\{\texttt{const}\} \times (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R})) \\ & \cup (\{\texttt{var}\} \times Ide \times \mathbb{N}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{bool}, \texttt{int}, \texttt{real}\} \times \{1\}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{array}\} \times \mathbb{Z}^2 \times Ide \times \mathbb{N}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{record}\} \times (Ide^2 \times \mathbb{N})^* \times \mathbb{N})\} \end{aligned}$$

Remarks:

- Variable descriptor $(\texttt{var}, I, n)$: type I, memory address n
- Last component of `type` entry: memory requirement
  (base types: 1 "cell")
- Array descriptor $(\texttt{type}, \texttt{array}, z_1, z_2, I, n)$:
  bounds z_1, z_2, component type I
- Record descriptor $(\texttt{type}, \texttt{record}, I_1, J_1, o_1, \ldots, I_l, J_l, o_l, n)$: selector I_k, component type J_k, memory offset o_k
- "Indexed" table lookup: $\mathrm{st}(I.I_k) := (J_k, o_k)$
  if $\mathrm{st}(I) = (\texttt{type}, \texttt{record}, \ldots, I_k, J_k, o_k, \ldots, n)$

Maintaining the Symbol Table I

The symbol table is again maintained by the function update(D, st) which specifies the update of symbol table st according to declaration D.

For the sake of simplificity we assume that $D = D_C\ D_T\ D_V \in Dcl$ is flattened, i.e., that every subtype is named by an identifier:

- If $D_T = \texttt{type}\ \ I_1 \texttt{:=} T_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} T_n \texttt{;}$, then for every $k \in [n]$
  - $T_k \in \{\texttt{bool}, \texttt{int}, \texttt{real}\}$ or
  - $T_k \in \{I_1, \ldots, I_{k-1}\}$ or
  - $T_k = \texttt{array[}z_1 \texttt{..} z_2\texttt{]}\ \ \texttt{of}\ \ I_j$ where $j \in [k-1]$ or
  - $T_k = \texttt{record}\ \ J_1 \texttt{:} I_{j_1} \texttt{;} \ldots \texttt{;} J_l \texttt{:} I_{j_l}\ \ \texttt{end}$ where $j_1, \ldots, j_l \in [k-1]$
- For D_T as above, D_V must be of the form $D_V = \texttt{var}\ \ J_1 \texttt{:} I_{j_1} \texttt{;} \ldots \texttt{;} J_k \texttt{:} I_{j_k} \texttt{;}$ where $j_1, \ldots, j_k \in [n]$

Maintaining the Symbol Table II

Definition 22.1 (Modified update function)

update : $Dcl \times Tab \dashrightarrow Tab$ is defined by

$$\begin{aligned}
\text{update}(D_C\ D_T\ D_V, \text{st}) &:= \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, \text{st}))) \\
\text{update}(\varepsilon, \text{st}) &:= \text{st} \\
\text{update}(\texttt{const}\ I_1\texttt{:=}c_1\texttt{;}\ldots\texttt{;}I_n\texttt{:=}c_n\texttt{;}, \text{st}) & \\
&:= \text{st}[I_1 \mapsto (\texttt{const}, c_1), \ldots, I_n \mapsto (\texttt{const}, c_n)] \\
\text{update}(\texttt{type}\ I\texttt{:=bool;}D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{bool}, 1)]) \\
\text{update}(\texttt{type}\ I\texttt{:=int;}D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{int}, 1)]) \\
\text{update}(\texttt{type}\ I\texttt{:=real;}D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{real}, 1)]) \\
\text{update}(\texttt{type}\ I\texttt{:=}J\texttt{;}D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto \text{st}(J)]) \\
\text{update}(\texttt{type}\ I\texttt{:=array[}z_1\texttt{..}z_2\texttt{]}\ \texttt{of}\ J\texttt{;}D'_T, \text{st}) & \\
&:= \text{update}(\texttt{type}\ D'_T, \\
&\quad \text{st}[I \mapsto (\texttt{type}, \texttt{array}, z_1, z_2, J, k \cdot n)]) \\
&\quad \text{if } \text{st}(J) = (\texttt{type}, \ldots, n) \text{ and } k = z_2 - z_1 + 1 \\
\text{update}(\texttt{type}\ I\texttt{:=record}\ I_1\texttt{:}J_1\texttt{;}\ldots\texttt{;}I_l\texttt{:}J_l\ \texttt{end;}D'_T, \text{st}) & \\
&:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto \\
&\quad (\texttt{type}, \texttt{record}, I_1, J_1, 0, I_2, J_2, n_1, \ldots, \\
&\quad I_l, J_l, \textstyle\sum_{i=1}^{l-1} n_i, \sum_{i=1}^{l} n_i)]) \\
&\quad \text{if } \text{st}(J_i) = (\texttt{type}, \ldots, n_i) \text{ for } i \in [l] \\
\text{update}(\texttt{var}\ I_1\texttt{:}J_1\texttt{;}\ldots\texttt{;}I_n\texttt{:}J_n\texttt{;}, \text{st}) &:= \text{st}[I_1 \mapsto (\texttt{var}, J_1, 0), I_2 \mapsto (\texttt{var}, J_2, n_1), \ldots, \\
&\quad I_n \mapsto (\texttt{var}, J_n, \textstyle\sum_{i=1}^{n-1} n_i)] \\
&\quad \text{if } \text{st}(J_i) = (\texttt{type}, \ldots, n_i) \text{ for } i \in [l]
\end{aligned}$$

Maintaining the Symbol Table III

Example 22.2 (Modified update function)

Let $D :=$
```
type Bool=bool; Int=int;
     Array=array[1..20] of Bool;
     Record=record S:Array; T:Int end;
var x:Int; y:Array; z:Record;
```

Then

$$\begin{aligned}\text{update}(D, \text{st}) = \text{st}[\ & \texttt{Bool} \mapsto (\texttt{type}, \texttt{bool}, 1),\\ & \texttt{Int} \mapsto (\texttt{type}, \texttt{int}, 1),\\ & \texttt{Array} \mapsto (\texttt{type}, \texttt{array}, 1, 20, \texttt{Bool}, 20),\\ & \texttt{Record} \mapsto (\texttt{type}, \texttt{record}, \texttt{S}, \texttt{Array}, 0, \texttt{T}, \texttt{Int}, 20, 21),\\ & \texttt{x} \mapsto (\texttt{var}, \texttt{Int}, 0),\\ & \texttt{y} \mapsto (\texttt{var}, \texttt{Array}, 1),\\ & \texttt{z} \mapsto (\texttt{var}, \texttt{Record}, 21)]\end{aligned}$$

Translation of Variables I

The translation employs the following auxiliary function to determine the type identifier of a given variable:

Definition 22.3 (vtype function)

The mapping

$$\text{vtype} : \mathit{Var} \times \mathit{Tab} \dashrightarrow \mathit{Ide}$$

is given by

$$\begin{aligned}
\text{vtype}(I, \text{st}) &:= J \\
&\quad \text{if } \text{st}(I) = (\texttt{var}, J, n) \\
\text{vtype}(V\,[E]\,, \text{st}) &:= J \\
&\quad \text{if } \text{vtype}(V, \text{st}) = I \\
&\quad \text{and } \text{st}(I) = (\texttt{type}, \texttt{array}, z_1, z_2, J, n) \\
\text{vtype}(V\,.\,I, \text{st}) &:= J \\
&\quad \text{if } \text{vtype}(V, \text{st}) = I' \text{ and } \text{st}(I'.I) = (J, o)
\end{aligned}$$

Translation of Variables II

The function vt generates code for computing the memory address of a variable (and storing it on the data stack):

Definition 22.4 (Translation of variables)

The mapping

$$\mathrm{vt} : \mathit{Var} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

is given by

$$\begin{aligned}
\mathrm{vt}(I, \mathrm{st}) &:= \texttt{LIT}(n)\texttt{;} \\
&\quad \text{if } \mathrm{st}(I) = (\texttt{var}, J, n) \\
\mathrm{vt}(V\texttt{[}E\texttt{]}, \mathrm{st}) &:= \mathrm{vt}(V, \mathrm{st}) && \% \text{ address of } V \\
&\quad \mathrm{et}(E, \mathrm{st}) && \% \text{ array index} \\
&\quad \texttt{CAB}(z_1\texttt{,}z_2\texttt{);} && \% \text{ bounds checking} \\
&\quad \texttt{LIT}(z_1\texttt{); SUB;} && \% \text{ index difference} \\
&\quad \texttt{LIT}(n\texttt{); MULT;} && \% \text{ relative address} \\
&\quad \texttt{ADD;} && \% \text{ address of } V\texttt{[}E\texttt{]} \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I \text{ and } \mathrm{st}(I) = (\texttt{type}, \texttt{array}, z_1, z_2, J, m) \\
&\quad \text{and } \mathrm{st}(J) = (\texttt{type}, \ldots, n) \\
\mathrm{vt}(V\texttt{.}I, \mathrm{st}) &:= \mathrm{vt}(V, \mathrm{st}) && \% \text{ address of } V \\
&\quad \texttt{LIT}(o)\texttt{;} && \% \text{ offset} \\
&\quad \texttt{ADD;} && \% \text{ address of } V\texttt{.}I \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I' \text{ and } \mathrm{st}(I'.I) = (J, o)
\end{aligned}$$

Translation of Expressions

Definition 22.5 (Translation of expressions)

The mapping

$$\text{et} : \mathit{Exp} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

is given by

$$\begin{array}{rcl}
\text{et}(c, \text{st}) & := & \texttt{LIT(}c\texttt{);} \\
\text{et}(V, \text{st}) & := & \begin{cases} \texttt{LIT(}c\texttt{);} & \text{if } V \in \mathit{Ide} \text{ and } \text{st}(V) = (\texttt{const}, c) \\ \text{vt}(V, \text{st}) & \text{if } \text{st}(I) = (\texttt{var}, J, n) \\ \texttt{LOAD;} & \end{cases} \\
\text{et}(E_1\texttt{+}E_2, \text{st}) & := & \text{et}(E_1, \text{st}) \\
 & & \text{et}(E_2, \text{st}) \\
 & & \texttt{ADD;} \\
 & \vdots &
\end{array}$$

Translation of Commands and Programs

Definition 22.6 (Translation of commands)

For the mapping

$$\text{ct} : \mathit{Cmd} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

only the handling of assignments needs to be adapted:

$$\begin{array}{rll} \text{ct}(V := E, \text{st}) := & \text{vt}(V, \text{st}) & \% \text{ address of left-hand side} \\ & \text{et}(E, \text{st}) & \% \text{ value of right-hand side} \\ & \texttt{STORE;} & \end{array}$$

Definition 22.7 (Translation of programs)

The mapping

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

is defined by

$$\text{trans}(D\ C) := \text{ct}(C, \text{update}(D, \text{st}_\emptyset))$$

Outline

Translation Example I

Example 22.8

$P =$
```
type Int=int; Array=array[1..10] of Int;
var a:Array; i:Int;
```
$\Big\} D$

$\Big\}$
```
i:=1;
while i<=10 do
  a[i]:=i; i:=i+1;
```
$\Big\} C$

$$
\begin{aligned}
\text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\
\text{st} &:= \text{update}(D, \text{st}_\emptyset) \\
&= \text{st}_\emptyset[\ \texttt{Int} \mapsto (\texttt{type}, \texttt{int}, 1), \\
&\qquad \texttt{Array} \mapsto (\texttt{type}, \texttt{array}, 1, 10, \texttt{Int}, 10), \\
&\qquad \texttt{a} \mapsto (\texttt{var}, \texttt{Array}, 0), \\
&\qquad \texttt{i} \mapsto (\texttt{var}, \texttt{Int}, 10)] \\
\text{ct}(C, \text{st}) &= \text{ct}(\texttt{i:=1}, \text{st}) \\
&\quad\ \text{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \text{st}) \\
\text{ct}(\texttt{i:=1}, \text{st}) &= \text{vt}(\texttt{i}, \text{st}) \quad \% \text{ adr}(\texttt{i}) \\
&\quad\ \text{et}(1, \text{st}) \quad \% \text{ val}(1) \\
&\quad\ \texttt{STORE;} \\
&= \texttt{LIT(10); LIT(1); STORE;}
\end{aligned}
$$

Translation Example II

Example 22.8 (continued)

$$
\begin{aligned}
&\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad = a: \ \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
&\qquad\quad \texttt{JFALSE}(a'); \\
&\qquad\quad \mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad\quad \texttt{JMP}(a); \\
&\qquad a': \\
&\mathrm{et}(\texttt{i<=10}, \mathrm{st}) = \texttt{LIT(10); LOAD; LIT(10); LE;} \\
&\mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) = \mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st})\ \mathrm{ct}(\texttt{i:=i+1}, \mathrm{st}) \\
&\mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st}) = \mathrm{vt}(\texttt{a[i]}, \mathrm{st}) \quad \% \ \mathrm{adr}(\texttt{a[i]}) \\
&\qquad\qquad \mathrm{et}(\texttt{i}, \mathrm{st}) \quad \% \ \mathrm{val}(\texttt{i}) \\
&\qquad\qquad \texttt{STORE;} \\
&\mathrm{vt}(\texttt{a[i]}, \mathrm{st}) = \mathrm{vt}(\texttt{a}, \mathrm{st}) \quad \% \ \mathrm{adr}(\texttt{a}) \\
&\qquad\qquad \mathrm{et}(\texttt{i}, \mathrm{st}) \quad \% \ \mathrm{val}(\texttt{i}) \\
&\qquad\qquad \texttt{CAB(1,10);} \quad \% \ \text{bounds checking} \\
&\qquad\qquad \texttt{LIT(1); SUB;} \quad \% \ \text{index diff.} \\
&\qquad\qquad \texttt{LIT(1); MULT;} \quad \% \ \text{rel. address} \\
&\qquad\qquad \texttt{ADD;} \quad \% \ \mathrm{adr}(\texttt{a[i]}) \\
&\mathrm{vt}(\texttt{a}, \mathrm{st}) = \texttt{LIT(0);} \\
&\mathrm{et}(\texttt{i}, \mathrm{st}) = \texttt{LIT(10); LOAD;} \\
&\mathrm{ct}(\texttt{i:=i+1}, \mathrm{st}) = \texttt{LIT(10); LIT(10); LIT(1); ADD; STORE;}
\end{aligned}
$$

Outline

Variant Records

Example 22.9 (Variant records in Pascal)

```
TYPE Coordinate = RECORD
                    nr: INTEGER;
                    CASE type: (cartesian, polar) OF
                      cartesian: (x, y: REAL);
                      polar: (r : REAL; phi: INTEGER )
                    END
                  END;
VAR pt: Coordinate;
pt.type := cartesian; pt.x := 0.5; pt.y := 1.2;
```

Implementation:

- Allocate memory for "biggest" variant
- Share memory between variant fields

Dynamic Arrays

Example 22.10 (Dynamic arrays in Pascal)

```
FUNCTION Sum(VAR a: ARRAY OF REAL): REAL;
  VAR
    i: INTEGER; s: REAL;
  BEGIN
    s := 0.0; FOR i := 0 to HIGH(A) do s := s + a[i] END; Sum := s
  END
```

Implementation:

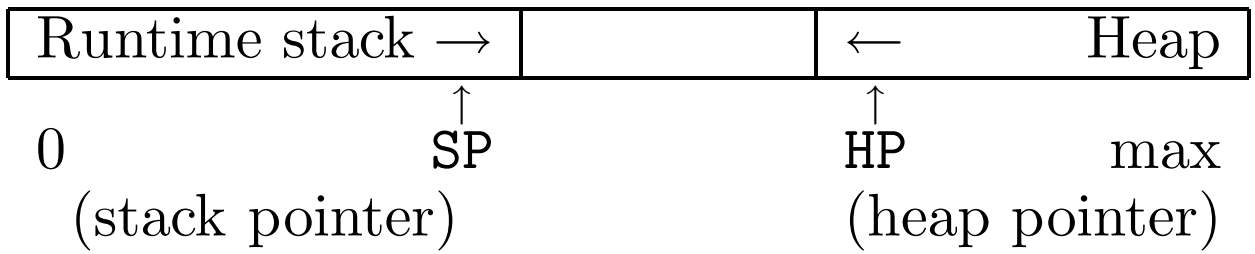
- Memory requirements unknown at compile time but determined by actual function/procedure parameters
  $\Longrightarrow$ no heap required
- Use array descriptor with following fields as parameter value:
  - starting memory address of array
  - size of array
  - lower index of array (possibly fixed by 0)
  - upper index of array (actually redundant)
- Use data stack or index register to access array elements

Dynamic Memory Allocation I

- Dynamically manipulated data structures (lists, trees, graphs, ...)
- So far: creation of (static) objects by declaration
- Now: creation of (dynamic) objects by explicit memory allocation
- Access by (implicit or explicit) pointers
- Deletion by explicit deallocation or garbage collection
  (= automatic deallocation of unreachable objects)
- Implementation: runtime stack not sufficient
  (lifetime of objects generally exceeds lifetime of procedure calls)

$\Longrightarrow$ new data structure: heap

- Simplest form of organization:



Dynamic Memory Allocation II

- New instruction: NEW
  - allocates n memory cells where n = topmost value of runtime stack
  - returns address of first cell
  - formal semantics:

```
if HP - <SP> > SP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

- But: collision check required for every operation which increases SP (e.g., expression evaluations)
- Efficient solution: add extreme stack pointer EP
  - points to topmost SP which will be used in the computation of current procedure
  - set by procedure entry code
  - statically computable at compile time
  - modified semantics of NEW:

```
if HP - <SP> > EP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

Garbage Collection

- explicitly by programmer (e.g., Pascal's `dispose` or C's `free`) or ...
- automatically by runtime system (garbage collection; e.g., Java)
  1. recursively mark all reachable objects
  2. deallocate all unreachable objects

  (careful design required for concurrent and/or realtime systems)
- management of deallocated memory areas by free list