Compiler Construction

Lecture 20: Code Generation IV (Jumping Code & 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/cc09/

Winter semester 2009/10

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:

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

Translation of Boolean Expressions

Definition (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

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

Outline

Boolean Expressions with Sequential Semantics

So far: Boolean expressions with strict semantics

$$b_1 \diamondsuit \bot = \bot$$
$$\bot \diamondsuit b_2 = \bot$$

Now: Boolean expressions with sequential semantics

$$\mathsf{false} \wedge \bot = \mathsf{false}$$
$$\mathsf{true} \vee \bot = \mathsf{true}$$
$$\bot \diamondsuit b = \bot$$

Idea:

- employ jump rather than Boolean instructions ("jumping code")
- equip bt and ct with two additional address parameters:

  a_t: target address for true

  a_f: target address for false

Jumping Code for Boolean Expressions

Definition 20.1 (Jumping code for Boolean expressions)

The mapping

$$\text{sbt} : \mathit{BExp} \times \mathit{Tab} \times \mathit{PC}^3 \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("sequential Boolean expression translation") is defined by

$$\begin{aligned}
\text{sbt}(A_1 \texttt{ < } A_2, \text{st}, a, a_t, a_f, l) &:= \begin{array}{l} \text{at}(A_1, \text{st}, a, l) \\ \text{at}(A_2, \text{st}, a', l) \\ a'' : \texttt{LT;} \\ a''+1 : \texttt{JFALSE(}a_f\texttt{);} \\ a''+2 : \texttt{JMP(}a_t\texttt{);} \end{array} \\
\text{sbt}(\texttt{not } B, \text{st}, a, a_t, a_f, l) &:= \text{sbt}(B, \text{st}, a, a_f, a_t, l) \\
\text{sbt}(B_1 \texttt{ and } B_2, \text{st}, a, a_t, a_f, l) &:= \begin{array}{l} \text{sbt}(B_1, \text{st}, a, a', a_f, l) \\ \text{sbt}(B_2, \text{st}, a', a_t, a_f, l) \end{array} \\
\text{sbt}(B_1 \texttt{ or } B_2, \text{st}, a, a_t, a_f, l) &:= \begin{array}{l} \text{sbt}(B_1, \text{st}, a, a_t, a', l) \\ \text{sbt}(B_2, \text{st}, a', a_t, a_f, l) \end{array}
\end{aligned}$$

Jumping Code for Commands

Definition 20.2 (Jumping code for commands)

The mapping

$$\text{sct} : \mathit{Cmd} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("sequential command translation") is defined by

$$\begin{aligned}
\text{sct}(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \text{st}, a, l) := \; & \text{sbt}(B, \text{st}, a, a_t, a_f, l) \\
& \text{sct}(C_1, \text{st}, a_t, l) \\
& a_f - 1 : \texttt{JMP}(a'); \\
& \text{sct}(C_2, \text{st}, a_f, l) \\
& a' : \\
\text{sct}(\texttt{while } B \texttt{ do } C, \text{st}, a, l) := \; & \text{sbt}(B, \text{st}, a, a_t, a_f, l) \\
& \text{sct}(C, \text{st}, a_t, l) \\
& a_f - 1 : \texttt{JMP}(a); \\
& a_f :
\end{aligned}$$

(remaining cases analogously)

Example: Jumping Code

Example 20.3

Translation of `while not (x < 1) and (x < y) do` C:

Strict:

```
1 : LOAD(x);
    LIT(1);
    LT;
    NOT;
    LOAD(x);
    LOAD(y);
    LT;
    AND;
    JFALSE(a);
    ct(C, ...)
    JMP(1);
a : ...
```

If `x` $= 0$:
9 instructions executed

Sequential:

```
1 : LOAD(x);
    LIT(1);
    LT;
    JFALSE(6);
    JMP(a);
6 : LOAD(x);
    LOAD(y);
    LT;
    JFALSE(a);
    JMP(11);
11 : sct(C, ...)
    JMP(1);
a : ...
```

If `x` $= 0$:
5 instructions executed

$\Longrightarrow$ generally: longer code, but shorter executions

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 20.4 (Modified syntax of EPL)

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

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

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 20.5 (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 20.6 (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 20.1 (Modified update function)

$\text{update} : \mathit{Dcl} \times \mathit{Tab} \dashrightarrow \mathit{Tab}$ 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{]\ 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 20.2 (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

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