Compiler Construction

Lecture 17: Code Generation III (Jumping Code & Static Data Structures)

Summer Semester 2017

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-17/cc/

Syntax of EPL

Definition (Syntax of EPL)

The syntax of EPL is defined as follows:

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

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

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

Boolean Expressions with Sequential Semantics

So far: Boolean expressions with strict semantics ($\bot$ = nontermination/runtime error)

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

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

Now: Boolean expressions with sequential semantics

$$\text{false} \wedge \bot = \text{false}$$

$$\text{true} \vee \bot = \text{true}$$

$$\bot \diamondsuit b = \bot$$

Idea:

- employ jumping 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 17.1 (Jumping code for Boolean expressions)

The mapping

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

("sequential Boolean expression translation") is defined by

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

Jumping Code for Commands

Definition 17.2 (Jumping code for commands)

The mapping

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

("sequential command translation") is defined by

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

(remaining cases analogously)

Boolean Expressions with Sequential Semantics

Example: Jumping Code

Example 17.3

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

Implementation 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 requirements known at compile time
- dynamic data structures: memory requirements runtime dependent
  - heap, pointers, garbage collection, ...

First step:

- static data structures (arrays and records)
- inductive type definitions
- no blocks or procedures (only for simplification; “orthogonal” extension)

Modified Syntax of EPL

Definition 17.4 (Modified syntax of EPL)

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

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

Static Semantics I

- All identifiers in a declaration D have to be different.
- In $T = \texttt{record}\ I_1 : T_1 \,;\, \ldots \,;\, I_n : T_n\ \texttt{end}$, all selectors I_j must be different.
- In $T = \texttt{array}\,[z_1 \,..\, z_2]\ \texttt{of}\ T$, $z_1 \leq z_2$.
- Type definitions must not be recursive:
  if $D_T = \texttt{type}\ I_1 := T_1 \,;\, \ldots \,;\, I_n := T_n \,;$ and type identifier I occurs in T_j, then $I \in \{I_1, \ldots, I_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be declared in D_T.
- 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"`)
  – risk of undetected errors; for programming-in-the-small (scripting languages)
  strong typing: explicit casting by programmer
  – enhanced software reliability; for programming-in-the-large
- Instantiation of operators/functions/procedures/... for different parameter types: polymorphism or overloading

$$\texttt{+} : \texttt{int} \times \texttt{int} \to \texttt{int} \qquad \texttt{+} : \texttt{real} \times \texttt{real} \to \texttt{real}$$

The Modified Abstract Machine AM

- Additional main storage for keeping data values
- Procedure stack not required anymore (as procedures no longer supported)

Definition 17.5 (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 $DS := \mathbb{R}^*$ ($\texttt{true} \mathrel{\hat{=}} 1$, $\texttt{false} \mathrel{\hat{=}} 0$; top to the right), and
- the main storage $MS := \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{R}\}$.

New AM Instructions

Definition 17.6 (New AM instructions)

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

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

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

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

Modifying the Symbol Table

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

Remarks:

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

Maintaining the Symbol Table I

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

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

Maintaining the Symbol Table II

Definition 17.7 (Modified update function)

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

$$
\begin{aligned}
&\mathrm{update}(D_C\ D_T\ D_V, \mathrm{st}) \\
&\quad := \mathrm{update}(D_V, \mathrm{update}(D_T, \mathrm{update}(D_C, \mathrm{st}))) \\
&\mathrm{update}(\varepsilon, \mathrm{st}) \\
&\quad := \mathrm{st} \\
&\mathrm{update}(\texttt{const}\ \ I_1 \texttt{:=} c_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} c_n \texttt{;}, \mathrm{st}) \\
&\quad := \mathrm{st}[I_1 \mapsto (\texttt{const}, c_1), \ldots, I_n \mapsto (\texttt{const}, c_n)] \\
&\mathrm{update}(\texttt{var}\ \ I_1 \texttt{:} J_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:} J_n \texttt{;}, \mathrm{st}) \\
&\quad := \mathrm{st}[I_1 \mapsto (\texttt{var}, J_1, o_1), \ldots, I_n \mapsto (\texttt{var}, J_n, o_n)] \\
&\qquad \text{if } \mathrm{st}(J_k) = (\texttt{type}, \ldots, m_k) \text{ and } o_k := \textstyle\sum_{i=1}^{k-1} m_i \text{ for } k \in [n]
\end{aligned}
$$

Maintaining the Symbol Table III

Definition 17.7 (Modified update function; continued)

$$\mathrm{update}(\texttt{type } I\texttt{:=bool};D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto (\texttt{type},\texttt{bool},1)])$$

$$\mathrm{update}(\texttt{type } I\texttt{:=int};D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto (\texttt{type},\texttt{int},1)])$$

$$\mathrm{update}(\texttt{type } I\texttt{:=real};D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto (\texttt{type},\texttt{real},1)])$$

$$\mathrm{update}(\texttt{type } I\texttt{:=}J;D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto \mathrm{st}(J)])$$

$$\mathrm{update}(\texttt{type } I\texttt{:=array [}z_1\texttt{..}z_2\texttt{] of } J;D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto (\texttt{type},\texttt{array},z_1,z_2,J,n\cdot m)]) \text{ if } \mathrm{st}(J)=(\texttt{type},\ldots,m) \text{ and } n := z_2 - z_1 + 1$$

$$\mathrm{update}(\texttt{type } I\texttt{:=record } I_1\texttt{:}J_1\texttt{;}\ldots\texttt{;}I_n\texttt{:}J_n\texttt{ end};D'_T,\mathrm{st}) := \mathrm{update}(\texttt{type } D'_T,\mathrm{st}[I \mapsto (\texttt{type},\texttt{record},I_1,J_1,o_1,\ldots,I_n,J_n,o_n,o_{n+1})]) \text{ if } \mathrm{st}(J_k)=(\texttt{type},\ldots,m_k) \text{ and } o_k := \textstyle\sum_{i=1}^{k-1} m_i \text{ for } k \in [n]$$

Maintaining the Symbol Table IV

Example 17.8 (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

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

Translation of Variables I

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

Definition 17.9 (vtype function)

The mapping

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

is given by

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

Translation of Variables II

Function vt generates code for computing the memory address of a variable on the data stack:

Definition 17.10 (Translation of variables)

The mapping $\mathrm{vt} : \mathit{Var} \times \mathit{Tab} \dashrightarrow \mathit{AM}$ is given by

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

Translation of Expressions

Definition 17.11 (Translation of expressions)

The mapping

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

is given by

$$\begin{aligned}
\mathrm{et}(c, \mathrm{st}) &:= \texttt{LIT(}c\texttt{);} \\
\mathrm{et}(V, \mathrm{st}) &:= \begin{cases} \texttt{LIT(}c\texttt{);} & \text{if } V \in \mathit{Ide} \text{ and } \mathrm{st}(V) = (\texttt{const}, c) \\ \mathrm{vt}(V, \mathrm{st}) & \text{otherwise} \\ \texttt{LOAD;} & \end{cases} \\
\mathrm{et}(E_1 \texttt{+} E_2, \mathrm{st}) &:= \mathrm{et}(E_1, \mathrm{st}) \\
&\quad\ \mathrm{et}(E_2, \mathrm{st}) \\
&\quad\ \texttt{ADD;} \\
&\ \ \vdots
\end{aligned}$$

Translation of Commands and Programs

Definition 17.12 (Translation of commands)

For the mapping

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

only the handling of assignments needs to be adapted:

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

Definition 17.13 (Translation of programs)

The mapping

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

is defined by

$$\mathrm{trans}(D\ C) := \mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}_\emptyset)) \quad \text{where } \mathrm{st}_\emptyset(I) \text{ is undefined for every } I \in \mathit{Ide}$$

Translation Example I

Example 17.14

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

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

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

Translation Example II

Example 17.14 (continued)

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