

Compiler Construction

Lecture 17: Code Generation III

(Implementation of Static Data Structures)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)



noll@cs.rwth-aachen.de

<http://moves.rwth-aachen.de/teaching/ss-14/cc14/>

Summer Semester 2014

Online Registration for Seminars and Practical Courses (Praktika) in Winter Term 2014/15

Who?

- Students of:
- Master Courses
 - Bachelor Informatik (~~Pro~~Seminar!)

Where?

www.graphics.rwth-aachen.de/apse

When?

04.07.2014 - 20.07.2014

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures
- 3 Static Data Structures
- 4 Modifying the Abstract Machine
- 5 Translation of Static Data Structures into AM Programs
- 6 A Translation Example

Definition (Syntax of EPL)

The **syntax of EPL** is defined as follows:

\mathbb{Z} : z (* z is an integer *)

Ide : I (* I is an identifier *)

$AExp$: $A ::= z \mid I \mid A_1 + A_2 \mid \dots$

$BExp$: $B ::= A_1 < A_2 \mid \text{not } B \mid B_1 \text{ and } B_2 \mid B_1 \text{ or } B_2$

Cmd : $C ::= I := A \mid C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid$
 $\text{while } B \text{ do } C \mid I()$

Dcl : $D ::= D_C D_V D_P$

$D_C ::= \varepsilon \mid \text{const } l_1 := z_1, \dots, l_n := z_n;$

$D_V ::= \varepsilon \mid \text{var } l_1, \dots, l_n;$

$D_P ::= \varepsilon \mid \text{proc } l_1; K_1; \dots; \text{proc } l_n; K_n;$

Blk : $K ::= D C$

Pgm : $P ::= \text{in/out } l_1, \dots, l_n; K.$

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures**
- 3 Static Data Structures
- 4 Modifying the Abstract Machine
- 5 Translation of Static Data Structures into AM Programs
- 6 A Translation Example

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...
⇒ **structured** state space, variables with components

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **structured** state space, variables with components

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

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **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, ...

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...
⇒ **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 procedures (for simplification; “orthogonal” extension)

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures
- 3 Static Data Structures**
- 4 Modifying the Abstract Machine
- 5 Translation of Static Data Structures into AM Programs
- 6 A Translation Example

Modified Syntax of EPL

Definition 17.1 (Modified syntax of EPL)

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

\mathbb{Z} :	z	(* z is an integer *)
\mathbb{B} :	$b ::= \text{true} \mid \text{false}$	(* b is a Boolean *)
\mathbb{R} :	r	(* r is a real number *)
<i>Con</i> :	$c ::= z \mid b \mid r$	(* c is a constant *)
<i>Ide</i> :	l	(* l is an identifier *)
<i>Type</i> :	$T ::= \text{bool} \mid \text{int} \mid \text{real} \mid l \mid \text{array}[z_1..z_2] \text{ of } T \mid$ $\text{record } l_1: T_1; \dots; l_n: T_n \text{ end}$	
<i>Var</i> :	$V ::= l \mid V[E] \mid V.l$	
<i>Exp</i> :	$E ::= c \mid V \mid E_1 + E_2 \mid E_1 < E_2 \mid E_1 \text{ and } E_2 \mid \dots$	
<i>Cmd</i> :	$C ::= V := E \mid C_1; C_2 \mid$ $\text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C$	
<i>Dcl</i> :	$D ::= D_C \ D_T \ D_V$	
	$D_C ::= \varepsilon \mid \text{const } l_1 := c_1; \dots; l_n := c_n;$	
	$D_T ::= \varepsilon \mid \text{type } l_1 := T_1; \dots; l_n := T_n;$	
	$D_V ::= \varepsilon \mid \text{var } l_1 : T_1; \dots; l_n : T_n;$	
<i>Pgm</i> :	$P ::= D \ C$	

- All identifiers in a declaration D have to be **different**.

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1..z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1; \dots; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1..z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1; \dots; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be **declared** in D_T .

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- 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).

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1..z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- 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).

- Array **indices** must have type **int**.

Static Semantics II

- Array **indices** must have type **int**.
- **Execution conditions** (**while**) and **branching expressions** (**if**) must have type **bool**.

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**.

- 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)

⇒ **type casts**

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

⇒ risc of undetected “real” errors;

for **programming-in-the-small** (script languages)

strong typing: explicit casting by programmer

⇒ enhanced software reliability;

for **programming-in-the-large**

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)

⇒ **type casts**

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

⇒ risc of undetected “real” errors;

for **programming-in-the-small** (script 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**

$+ : \text{int} \times \text{int} \rightarrow \text{int}$ $+ : \text{real} \times \text{real} \rightarrow \text{real}$

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures
- 3 Static Data Structures
- 4 Modifying the Abstract Machine**
- 5 Translation of Static Data Structures into AM Programs
- 6 A Translation Example

The Modified Abstract Machine AM

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

Definition 17.2 (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** $DS := \mathbb{R}^*$ (top to the right), and
- the **main storage** $MS := \{\sigma \mid \sigma : \mathbb{N} \rightarrow \mathbb{R}\}$.

Definition 17.3 (New AM instructions)

- **Procedure instructions** are no longer needed.

Definition 17.3 (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 $\llbracket O \rrbracket : S \dashrightarrow S$:

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

Definition 17.3 (New AM instructions)

- **Procedure instructions** are no longer needed.
- **Transfer instructions** ($\text{LOAD}(dif, off)$, $\text{STORE}(dif, off)$) are replaced by the following instructions with the respective semantics $\llbracket O \rrbracket : S \dashrightarrow S$:

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

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

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

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures
- 3 Static Data Structures
- 4 Modifying the Abstract Machine
- 5 Translation of Static Data Structures into AM Programs**
- 6 A Translation Example

Modifying the Symbol Table

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

Remarks:

- **Variable descriptor** (var, l, n) : type l , memory address n
- Last component of **type** entry: **memory requirement**
(base types: 1 “cell”)
- **Array descriptor** $(\text{type}, \text{array}, z_1, z_2, l, n)$:
bounds z_1, z_2 , component type l
- **Record descriptor** $(\text{type}, \text{record}, l_1, J_1, o_1, \dots, l_l, J_l, o_l, n)$: selector
 l_k , component type J_k , memory offset o_k
- “Indexed” table lookup: $\text{st}(l.l_k) := (J_k, o_k)$
if $\text{st}(l) = (\text{type}, \text{record}, \dots, l_k, J_k, o_k, \dots, n)$

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 .

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 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 $D_T = \text{type } l_1 := T_1; \dots; l_n := T_n;$, then for every $k \in [n]$
 - $T_k \in \{\text{bool}, \text{int}, \text{real}\}$ or
 - $T_k \in \{l_1, \dots, l_{k-1}\}$ or
 - $T_k = \text{array}[z_1..z_2]$ of l_j where $j \in [k-1]$ or
 - $T_k = \text{record } J_1 : l_{j_1}; \dots; J_l : l_{j_l}$ end where $j_1, \dots, j_l \in [k-1]$
- For D_T as above, D_V must be of the form
 $D_V = \text{var } J_1 : l_{j_1}; \dots; J_k : l_{j_k};$ where $j_1, \dots, j_k \in [n]$

Definition 17.1 (Modified update function)

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

$$\text{update}(D_C \ D_T \ D_V, st) := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st)))$$
$$\text{update}(\varepsilon, st) := st$$

Definition 17.1 (Modified update function)

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

$$\text{update}(D_C \ D_T \ D_V, st) := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st)))$$
$$\text{update}(\varepsilon, st) := st$$
$$\text{update}(\text{const } l_1 := c_1; \dots; l_n := c_n; , st)$$
$$:= st[l_1 \mapsto (\text{const}, c_1), \dots, l_n \mapsto (\text{const}, c_n)]$$

Maintaining the Symbol Table II

Definition 17.1 (Modified update function)

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

$$\text{update}(D_C \ D_T \ D_V, st) := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st)))$$

$$\text{update}(\varepsilon, st) := st$$

$$\text{update}(\text{const } h_1 := c_1; \dots; l_n := c_n; , st)$$

$$:= st[h_1 \mapsto (\text{const}, c_1), \dots, l_n \mapsto (\text{const}, c_n)]$$

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

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

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

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

$$\text{update}(\text{type } l := \text{array}[z_1 .. z_2] \text{ of } J; D'_T, st)$$

$$:= \text{update}(\text{type } D'_T,$$

$$st[l \mapsto (\text{type}, \text{array}, z_1, z_2, J, k \cdot n)])$$

$$\text{if } st(J) = (\text{type}, \dots, n) \text{ and } k = z_2 - z_1 + 1$$

$$\text{update}(\text{type } l := \text{record } h_1 : J_1; \dots; l_l : J_l \text{ end}; D'_T, st)$$

$$:= \text{update}(\text{type } D'_T, st[l \mapsto$$

$$(\text{type}, \text{record}, h_1, J_1, 0, h_2, J_2, n_1, \dots,$$

$$l_l, J_l, \sum_{i=1}^{l-1} n_i, \sum_{i=1}^l n_i)])$$

$$\text{if } st(J_i) = (\text{type}, \dots, n_i) \text{ for } i \in [l]$$

Maintaining the Symbol Table II

Definition 17.1 (Modified update function)

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

$$\text{update}(D_C \ D_T \ D_V, st) := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st)))$$

$$\text{update}(\varepsilon, st) := st$$

$$\text{update}(\text{const } h_1 := c_1; \dots; l_n := c_n; , st)$$

$$:= st[h_1 \mapsto (\text{const}, c_1), \dots, l_n \mapsto (\text{const}, c_n)]$$

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

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

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

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

$$\text{update}(\text{type } l := \text{array}[z_1 \dots z_2] \text{ of } J; D'_T, st)$$

$$:= \text{update}(\text{type } D'_T,$$

$$st[l \mapsto (\text{type}, \text{array}, z_1, z_2, J, k \cdot n)])$$

$$\text{if } st(J) = (\text{type}, \dots, n) \text{ and } k = z_2 - z_1 + 1$$

$$\text{update}(\text{type } l := \text{record } h_1 : J_1; \dots; l_l : J_l \text{ end}; D'_T, st)$$

$$:= \text{update}(\text{type } D'_T, st[l \mapsto$$

$$(\text{type}, \text{record}, h_1, J_1, 0, h_2, J_2, n_1, \dots,$$

$$l_l, J_l, \sum_{i=1}^{l-1} n_i, \sum_{i=1}^l n_i)])$$

$$\text{if } st(J_i) = (\text{type}, \dots, n_i) \text{ for } i \in [l]$$

$$\text{update}(\text{var } h_1 : J_1; \dots; l_n : J_n; , st) := st[h_1 \mapsto (\text{var}, J_1, 0), h_2 \mapsto (\text{var}, J_2, n_1), \dots,$$

$$l_n \mapsto (\text{var}, J_n, \sum_{i=1}^{n-1} n_i)]$$

$$\text{if } st(J_i) = (\text{type}, \dots, n_i) \text{ for } i \in [l]$$

Example 17.2 (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;
```

Example 17.2 (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

```
update( $D, st$ ) = st[ Bool  $\mapsto$  (type, bool, 1),
                    Int  $\mapsto$  (type, int, 1),
                    Array  $\mapsto$  (type, array, 1, 20, Bool, 20),
                    Record  $\mapsto$  (type, record, S, Array, 0, T, Int, 20, 21),
                    x  $\mapsto$  (var, Int, 0),
                    y  $\mapsto$  (var, Array, 1),
                    z  $\mapsto$  (var, Record, 21)]
```

Translation of Variables I

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

Definition 17.3 (vtype function)

The mapping

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

is given by

$$\text{vtype}(I, \text{st}) := J \\ \text{if } \text{st}(I) = (\text{var}, J, n)$$

$$\text{vtype}(V[E], \text{st}) := J \\ \text{if } \text{vtype}(V, \text{st}) = I \\ \text{and } \text{st}(I) = (\text{type}, \text{array}, z_1, z_2, J, n)$$

$$\text{vtype}(V.I, \text{st}) := J \\ \text{if } \text{vtype}(V, \text{st}) = I' \text{ and } \text{st}(I'.I) = (J, o)$$

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 17.4 (Translation of variables)

The mapping

$$vt : Var \times Tab \dashrightarrow AM$$

is given by

```
vt(l, st) := LIT(n) ;
           if st(l) = (var, J, n)
vt(V[E], st) := vt(V, st)      % address of V
                et(E, st)       % array index
                CAB(z1, z2) ; % bounds checking
                LIT(z1) ; SUB ; % index difference
                LIT(n) ; MULT ; % relative address
                ADD ;             % address of V[E]
                if vtype(V, st) = l and st(l) = (type, array, z1, z2, J, m)
                and st(J) = (type, ..., n)
vt(V.l, st) := vt(V, st)      % address of V
                LIT(o) ;       % offset
                ADD ;           % address of V.l
                if vtype(V, st) = l' and st(l'.l) = (J, o)
```


Definition 17.5 (Translation of expressions)

The mapping

$$et : Exp \times Tab \dashrightarrow AM$$

is given by

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

Definition 17.6 (Translation of commands)

For the mapping

$$ct : Cmd \times Tab \dashrightarrow AM$$

only the handling of assignments needs to be adapted:

```
ct( $V := E$ , st) := vt( $V$ , st)  % address of left-hand side
                  et( $E$ , st)  % value of right-hand side
                  STORE;
```

Definition 17.6 (Translation of commands)

For the mapping

$$ct : Cmd \times Tab \dashrightarrow AM$$

only the handling of assignments needs to be adapted:

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

Definition 17.7 (Translation of programs)

The mapping

$$trans : Pgm \dashrightarrow AM$$

is defined by

$$trans(D \ C) := ct(C, update(D, st_{\emptyset}))$$

- 1 Recap: Syntax of EPL
- 2 Implementation of Data Structures
- 3 Static Data Structures
- 4 Modifying the Abstract Machine
- 5 Translation of Static Data Structures into AM Programs
- 6 A Translation Example**

Example 17.8

```
 $P = \left. \begin{array}{l} \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \\ \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} D$ 
```

C

Example 17.8

```
 $P = \left. \begin{array}{l} \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \\ \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} D$ 
```

C

$\text{trans}(P) = \text{ct}(C, \text{update}(D, \text{st}_0))$

Example 17.8

$$P = \left. \begin{array}{l} \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \\ \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} D$$
$$\left. \begin{array}{l} \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} C$$
$$\begin{aligned} \text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\ \text{st} &:= \text{update}(D, \text{st}_\emptyset) \\ &= \text{st}_\emptyset[\text{Int} \mapsto (\text{type}, \text{int}, 1), \\ &\quad \text{Array} \mapsto (\text{type}, \text{array}, 1, 10, \text{Int}, 10), \\ &\quad \text{a} \mapsto (\text{var}, \text{Array}, 0), \\ &\quad \text{i} \mapsto (\text{var}, \text{Int}, 10)] \end{aligned}$$

Example 17.8

$$P = \left. \begin{array}{l} \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \end{array} \right\} D$$
$$\left. \begin{array}{l} \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} C$$
$$\begin{aligned} \text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\ \text{st} &:= \text{update}(D, \text{st}_\emptyset) \\ &= \text{st}_\emptyset [\text{Int} \mapsto (\text{type}, \text{int}, 1), \\ &\quad \text{Array} \mapsto (\text{type}, \text{array}, 1, 10, \text{Int}, 10), \\ &\quad \text{a} \mapsto (\text{var}, \text{Array}, 0), \\ &\quad \text{i} \mapsto (\text{var}, \text{Int}, 10)] \\ \text{ct}(C, \text{st}) &= \text{ct}(\text{i:=1}, \text{st}) \\ &\quad \text{ct}(\text{while i<=10 do a[i]:=i; i:=i+1}, \text{st}) \end{aligned}$$

Example 17.8

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

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
  = a : et(i<=10,st)
        JFALSE(a');
        ct(a[i]:=i; i:=i+1,st)
        JMP(a);
  a' :
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a: et(i<=10,st)
        JFALSE(a');
        ct(a[i]:=i; i:=i+1,st)
        JMP(a);
    a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a: et(i<=10,st)
        JFALSE(a');
        ct(a[i]:=i; i:=i+1,st)
        JMP(a);
    a' :
    et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
    ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a: et(i<=10,st)
        JFALSE(a');
        ct(a[i]:=i; i:=i+1,st)
        JMP(a);
    a' :
    et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st) % adr(a[i])
                et(i,st) % val(i)
                STORE;
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st) % adr(a[i])
                et(i,st) % val(i)
                STORE;
vt(a[i],st) = vt(a,st) % adr(a)
              et(i,st) % val(i)
              CAB(1,10); % bounds checking
              LIT(1); SUB; % index diff.
              LIT(1); MULT; % rel. address
              ADD; % adr(a[i])
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st) % adr(a[i])
                et(i,st) % val(i)
                STORE;
vt(a[i],st) = vt(a,st) % adr(a)
              et(i,st) % val(i)
              CAB(1,10); % bounds checking
              LIT(1); SUB; % index diff.
              LIT(1); MULT; % rel. address
              ADD; % adr(a[i])
vt(a,st) = LIT(0);
```

Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st) % adr(a[i])
                et(i,st) % val(i)
                STORE;
vt(a[i],st) = vt(a,st) % adr(a)
              et(i,st) % val(i)
              CAB(1,10); % bounds checking
              LIT(1); SUB; % index diff.
              LIT(1); MULT; % rel. address
              ADD; % adr(a[i])
vt(a,st) = LIT(0);
et(i,st) = LIT(10); LOAD;
```


Example 17.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st) % adr(a[i])
                et(i,st) % val(i)
                STORE;
vt(a[i],st) = vt(a,st) % adr(a)
              et(i,st) % val(i)
              CAB(1,10); % bounds checking
              LIT(1); SUB; % index diff.
              LIT(1); MULT; % rel. address
              ADD; % adr(a[i])
vt(a,st) = LIT(0);
et(i,st) = LIT(10); LOAD;
ct(i:=i+1,st) = LIT(10); LIT(10); LOAD; LIT(1); ADD; STORE;
```