Compiler Construction

Lecture 23: Code Generation IV (Implementation of 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/cc10/

Winter semester 2010/11

Outline

Syntax of EPL

Definition (Syntax of EPL)

The syntax of EPL is defined as follows:

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

Outline

Implementation of Data Structures

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

$\Longrightarrow$ structured state space, variables with components

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

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 ($\Longrightarrow$ address computation)

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

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 ($\Longrightarrow$ 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)

Modified Syntax of EPL

Definition 23.1 (Modified syntax of EPL)

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

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

Static Semantics I

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

Static Semantics I

- All identifiers in a declaration D have to be different.
- In $T =$ `record` I_1`:`T_1`;` $\ldots$ `;`I_n`:`T_n `end`, all selectors I_j must be different.

Static Semantics I

- All identifiers in a declaration D have to be different.
- In $T =$ `record` I_1`:`T_1`;` $\ldots$ `;`I_n`:`T_n `end`, all selectors I_j must be different.
- In $T =$ `array[`z_1`..`z_2`] of` T, $z_1 \leq z_2$.

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 type identifier I occurs in T_j, then $I \in \{I_1, \ldots, I_{j-1}\}$.

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 type 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.

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 type 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).

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 type 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`.

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.

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

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)

  $\implies$ type casts

  weak typing: implicit casting by compiler (`2.5 + 1`, `1 + "42"`)
  $\implies$ risc of undetected "real" errors;
  for programming-in-the-small (script languages)

  strong typing: explicit casting by programmer
  $\implies$ enhanced software reliability;
  for programming-in-the-large
- Instantiation of operators/functions/procedures/... for different parameter types: polymorphism or overloading

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

Outline

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

Definition 23.2 (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 $DS := \mathbb{R}^*$, and
- the main storage $MS := \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{R}\}$.

New AM Instructions

Definition 23.3 (New AM instructions)

- Procedure instructions are no longer needed.

New AM Instructions

Definition 23.3 (New AM instructions)

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

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

Definition 23.3 (New AM instructions)

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

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

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

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

Outline

Modifying the Symbol Table

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

Remarks:

- Variable descriptor $(\texttt{var}, I, n)$: 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 $\mathrm{update}(D, \mathrm{st})$ which specifies the update of symbol table st according to declaration D.

The symbol table is again maintained by the function $\text{update}(D, \text{st})$ 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 $D_T = \texttt{type}\ I_1\texttt{:=}T_1\texttt{;}\ldots\texttt{;}I_n\texttt{:=}T_n\texttt{;}$, 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 $j_1, \ldots, j_k \in [n]$

Definition 23.1 (Modified update function)

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

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

Definition 23.1 (Modified update function)

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

$$
\begin{aligned}
\text{update}(D_C\ D_T\ D_V, \text{st}) &:= \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, \text{st}))) \\
\text{update}(\varepsilon, \text{st}) &:= \text{st} \\
\text{update}(\texttt{const}\ I_1 \texttt{:=} c_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} c_n \texttt{;}, \text{st}) & \\
&:= \text{st}[I_1 \mapsto (\texttt{const}, c_1), \ldots, I_n \mapsto (\texttt{const}, c_n)]
\end{aligned}
$$

Maintaining the Symbol Table II

Definition 23.1 (Modified update function)

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

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

Definition 23.1 (Modified update function)

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

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

Maintaining the Symbol Table III

Example 23.2 (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;
```

Maintaining the Symbol Table III

Example 23.2 (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

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

Translation of Variables I

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

Definition 23.3 (vtype function)

The mapping

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

is given by

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

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 23.4 (Translation of variables)

The mapping

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

is given by

$$
\begin{array}{rll}
\mathrm{vt}(I, \mathrm{st}) := & \texttt{LIT(}n\texttt{);} & \\
& \text{if } \mathrm{st}(I) = (\texttt{var}, J, n) & \\
\mathrm{vt}(V\texttt{[}E\texttt{]}, \mathrm{st}) := & \mathrm{vt}(V, \mathrm{st}) & \%\ \text{address of } V \\
& \mathrm{et}(E, \mathrm{st}) & \%\ \text{array index} \\
& \texttt{CAB(}z_1\texttt{,}z_2\texttt{);} & \%\ \text{bounds checking} \\
& \texttt{LIT(}z_1\texttt{); SUB;} & \%\ \text{index difference} \\
& \texttt{LIT(}n\texttt{); MULT;} & \%\ \text{relative address} \\
& \texttt{ADD;} & \%\ \text{address of } V\texttt{[}E\texttt{]} \\
& \multicolumn{2}{l}{\text{if } \mathrm{vtype}(V, \mathrm{st}) = I \text{ and } \mathrm{st}(I) = (\texttt{type}, \texttt{array}, z_1, z_2, J, m)} \\
& \multicolumn{2}{l}{\text{and } \mathrm{st}(J) = (\texttt{type}, \ldots, n)} \\
\mathrm{vt}(V\texttt{.}I, \mathrm{st}) := & \mathrm{vt}(V, \mathrm{st}) & \%\ \text{address of } V \\
& \texttt{LIT(}o\texttt{);} & \%\ \text{offset} \\
& \texttt{ADD;} & \%\ \text{address of } V\texttt{.}I \\
& \multicolumn{2}{l}{\text{if } \mathrm{vtype}(V, \mathrm{st}) = I' \text{ and } \mathrm{st}(I'.I) = (J, o)}
\end{array}
$$

Definition 23.5 (Translation of expressions)

The mapping

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

is given by

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

Translation of Commands and Programs

Definition 23.6 (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:

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

Translation of Commands and Programs

Definition 23.6 (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:

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

Definition 23.7 (Translation of programs)

The mapping

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

is defined by

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

Outline

Translation Example I

Example 23.8

$P =$
```
type Int=int; Array=array[1..10] of Int;
var a:Array; i:Int;
i:=1;
while i<=10 do
  a[i]:=i; i:=i+1;
```
(lines 1–2: D; lines 3–5: C)

Translation Example I

Example 23.8

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

$$\mathrm{trans}(P) \;=\; \mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}_\emptyset))$$

Translation Example I

Example 23.8

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

$$\begin{aligned}
\mathrm{trans}(P) &= \mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}_\emptyset))\\
\mathrm{st} &:= \mathrm{update}(D, \mathrm{st}_\emptyset)\\
&= \mathrm{st}_\emptyset[\ \begin{array}{rl}\texttt{Int} \mapsto & (\texttt{type}, \texttt{int}, 1),\\ \texttt{Array} \mapsto & (\texttt{type}, \texttt{array}, 1, 10, \texttt{Int}, 10),\\ \texttt{a} \mapsto & (\texttt{var}, \texttt{Array}, 0),\\ \texttt{i} \mapsto & (\texttt{var}, \texttt{Int}, 10)]\end{array}
\end{aligned}$$

Translation Example I

Example 23.8

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

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

Translation Example I

Example 23.8

$$P = \left.\begin{array}{l}\texttt{type Int=int; Array=array[1..10] of Int;}\\ \texttt{var a:Array; i:Int;}\end{array}\right\} D$$

$$\left.\begin{array}{l}\texttt{i:=1;}\\ \texttt{while i<=10 do}\\ \quad\texttt{a[i]:=i; i:=i+1;}\end{array}\right\} C$$

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

Translation Example II

Example 23.8 (continued)

$$
\begin{aligned}
\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st})
&= a: \ \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
&\texttt{JFALSE(}a'\texttt{);} \\
&\mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\texttt{JMP(}a\texttt{);} \\
& a':
\end{aligned}
$$

Example 23.8 (continued)

$$
\begin{aligned}
\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st}) \\
= a :\ & \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
& \texttt{JFALSE(}a'\texttt{);} \\
& \mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
& \texttt{JMP(}a\texttt{);} \\
a' :\ & \\
\mathrm{et}(\texttt{i<=10}, \mathrm{st}) = {} & \texttt{LIT(10); LOAD; LIT(10); LE;}
\end{aligned}
$$

Translation Example II

Example 23.8 (continued)

$$
\begin{aligned}
\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st}) \\
&= a: \ \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
&\qquad \texttt{JFALSE(}a'\texttt{);} \\
&\qquad \mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad \texttt{JMP(}a\texttt{);} \\
&\quad\ a': \\
\mathrm{et}(\texttt{i<=10}, \mathrm{st}) &= \texttt{LIT(10); LOAD; LIT(10); LE;} \\
\mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) &= \mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st})\ \mathrm{ct}(\texttt{i:=i+1}, \mathrm{st})
\end{aligned}
$$

Translation Example II

Example 23.8 (continued)

$$
\begin{aligned}
&\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad = a:\ \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
&\qquad\qquad \texttt{JFALSE(}a'\texttt{);} \\
&\qquad\qquad \mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad\qquad \texttt{JMP(}a\texttt{);} \\
&\qquad\quad a':
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{et}(\texttt{i<=10}, \mathrm{st}) &= \texttt{LIT(10); LOAD; LIT(10); LE;} \\
\mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) &= \mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st})\ \mathrm{ct}(\texttt{i:=i+1}, \mathrm{st}) \\
\mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st}) &= \mathrm{vt}(\texttt{a[i]}, \mathrm{st}) \quad \%\ \mathrm{adr}(\texttt{a[i]}) \\
&\ \mathrm{et}(\texttt{i}, \mathrm{st}) \qquad\quad \%\ \mathrm{val}(\texttt{i}) \\
&\ \texttt{STORE;}
\end{aligned}
$$

Translation Example II

Example 23.8 (continued)

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

Translation Example II

Example 23.8 (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 23.8 (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;
```

Translation Example II

Example 23.8 (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;
```