

Compiler Construction

Lecture 18: Code Generation III (Translation to Intermediate Code)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University
`noll@cs.rwth-aachen.de`

`http://www-i2.informatik.rwth-aachen.de/i2/cc08/`

Summer semester 2008

- 1 Repetition: Intermediate Code for EPL
- 2 The Procedure Stack
- 3 Translation of EPL into AM Programs

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 $d = d.r : \dots : d.1 \in DS$, and
- a procedure stack $p = p.1 : \dots : p.t \in PS$.

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)` ($z \in \mathbb{Z}$)

Definition (Semantics of AM instructions (1st part))

The semantics of an AM instruction O

$$\llbracket O \rrbracket : S \dashrightarrow S$$

is defined as follows:

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

- 1 Repetition: Intermediate Code for EPL
- 2 The Procedure Stack
- 3 Translation of EPL into AM Programs

Structure of Procedure Stack I

The semantics of procedure and transfer instructions requires a particular structure of the procedure stack $p \in PS$: it must be composed of **frames** (or: **activation records**) of the form

$$sl : dl : ra : v_1 : \dots : v_k$$

where

static link sl : points to frame of surrounding declaration environment
 $\implies$ used to access non-local variables

dynamic link dl : points to previous frame (i.e., of calling procedure)
 $\implies$ used to remove topmost frame after termination of procedure call

return address ra : program label after termination of procedure call
 $\implies$ used to continue program execution after termination of procedure call

local variables v_i : values of locally declared variables

- Frames are **created** whenever a procedure call is performed
- Two **special frames**:
 - I/O frame**: for keeping values of **in/out** variables
($sl = dl = ra = 0$)
 - MAIN frame**: for keeping values of top-level block
($sl = dl = \text{I/O frame}$)

Example 18.1 (cf. Example 16.8)

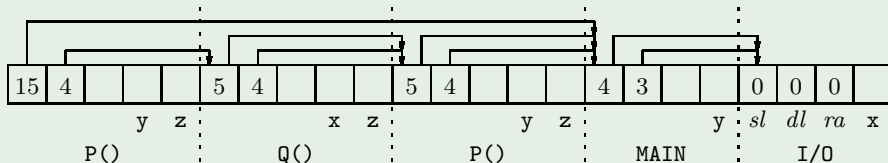
```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... Q() ...]  
proc R;  
  [... P() ...]  
[... P() ...].
```

Structure of Procedure Stack III

Example 18.1 (cf. Example 16.8)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... Q() ...]  
proc R;  
  [... P() ...]  
  [... P() ...].
```

Procedure stack after second call of P:



Structure of Procedure Stack IV

Observation:

- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

Structure of Procedure Stack IV

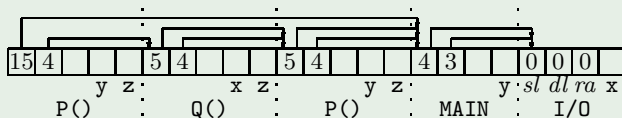
Observation:

- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

Example 18.2 (cf. Example 18.1)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... x ... y ... Q() ...]  
  proc R;  
    [... P() ...]  
  [... P() ...].
```

Procedure stack after second call of P:



Structure of Procedure Stack IV

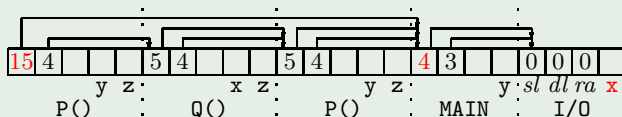
Observation:

- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

Example 18.2 (cf. Example 18.1)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... x ... y ... Q() ...]  
proc R;  
  [... P() ...]  
  [... P() ...].
```

Procedure stack after second call of P:



P uses x $\implies dif = 2$

Structure of Procedure Stack IV

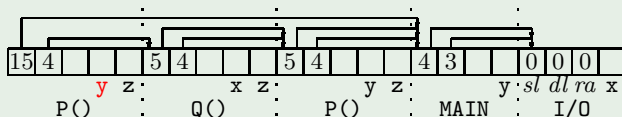
Observation:

- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

Example 18.2 (cf. Example 18.1)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... x ... y ... Q() ...]  
proc R;  
  [... P() ...]  
  [... P() ...].
```

Procedure stack after second call of P:



P uses y $\Rightarrow$ *dif* = 0

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned}\text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)\end{aligned}$$

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned}\text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)\end{aligned}$$

Example 18.4 (cf. Example 18.2)

In the second call of P (from Q): $dif = 2$
 $\text{base}(p, 0) = 1$

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned}\text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)\end{aligned}$$

Example 18.4 (cf. Example 18.2)

In the second call of P (from Q): $dif = 2$

$$\begin{aligned}\text{base}(p, 0) &= 1 \\ \implies \text{base}(p, 1) &= 1 + p.1 = 6\end{aligned}$$

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned}\text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)\end{aligned}$$

Example 18.4 (cf. Example 18.2)

In the second call of P (from Q): $dif = 2$

$$\begin{aligned}\text{base}(p, 0) &= 1 \\ \implies \text{base}(p, 1) &= 1 + p.1 = 6 \\ \implies \text{base}(p, 2) &= 6 + p.6 = 11\end{aligned}$$

The base Function

Upon procedure call, the static link information is computed by the following auxiliary function which, given a procedure stack and a level difference, determines the begin of the corresponding frame.

Definition 18.3 (base function)

The function

$$\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$$

is given by

$$\begin{aligned}\text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)\end{aligned}$$

Example 18.4 (cf. Example 18.2)

In the second call of P (from Q): $dif = 2$

$$\begin{aligned}\text{base}(p, 0) &= 1 \\ \implies \text{base}(p, 1) &= 1 + p.1 = 6 \\ \implies \text{base}(p, 2) &= 6 + p.6 = 11 \\ \implies sl &= \text{base}(p, 2) + \underbrace{2}_{y,z} + \underbrace{2}_{ra,dl} = 15\end{aligned}$$

Semantics of Procedure Instructions

- $\text{CALL}(ca, dif, loc)$ with
 - **code address** $ca \in PC$
 - **level difference** $dif \in \mathbb{N}$
 - **number of local variables** $loc \in \mathbb{N}$

creates the new frame and **jumps** to the given address
(= starting address of procedure)
- RET **removes** the topmost frame and returns to the calling site

Semantics of Procedure Instructions

- $\text{CALL}(ca, dif, loc)$ with
 - **code address** $ca \in PC$
 - **level difference** $dif \in \mathbb{N}$
 - **number of local variables** $loc \in \mathbb{N}$

creates the new frame and **jumps** to the given address
(= starting address of procedure)

- **RET removes** the topmost frame and returns to the calling site

Definition 18.5 (Semantics of procedure instructions)

The **semantics of a procedure instruction** O , $\llbracket O \rrbracket : S \dashrightarrow S$, is defined as follows:

$$\begin{aligned} \llbracket \text{CALL}(ca, dif, loc) \rrbracket(l, d, p) \\ &:= (ca, d, \underbrace{(\text{base}(p, dif) + loc + 2)}_{sl} : \underbrace{(loc + 2)}_{dl} : \underbrace{(l + 1)}_{ra} : \underbrace{0 : \dots : 0}_{loc \text{ times}} : p) \\ \llbracket \text{RET} \rrbracket(l, d, p.1 : \dots : p.t) \\ &:= (\underbrace{p.3}_{ra}, d, p.(\underbrace{p.2}_{dl} + 2) : \dots : p.t) \quad \text{if } t \geq p.2 + 2 \end{aligned}$$

Semantics of Transfer Instructions

- $\text{LOAD}(dif, off)$ and $\text{STORE}(dif, off)$ with
 - **level difference** $dif \in \mathbb{N}$
 - **variable offset** $off \in \mathbb{N}$

respectively load and store variable values between data and procedure stack, following a chain of dif static links

- $\text{LIT}(z)$ loads the literal constant $z \in \mathbb{Z}$

Semantics of Transfer Instructions

- $\text{LOAD}(dif, off)$ and $\text{STORE}(dif, off)$ with
 - **level difference** $dif \in \mathbb{N}$
 - **variable offset** $off \in \mathbb{N}$

respectively load and store variable values between data and procedure stack, following a chain of dif static links

- $\text{LIT}(z)$ loads the literal constant $z \in \mathbb{Z}$

Definition 18.6 (Semantics of transfer instructions)

The **semantics of a transfer instruction** O , $\llbracket O \rrbracket : S \dashrightarrow S$, is defined as follows:

$$\begin{aligned}\llbracket \text{LOAD}(dif, off) \rrbracket(l, d, p) &:= (l + 1, d : p.(\text{base}(p, dif) + off + 2), p) \\ \llbracket \text{STORE}(dif, off) \rrbracket(l, d : z, p) &:= (l + 1, d, p[\text{base}(p, dif) + off + 2 \mapsto z]) \\ \llbracket \text{LIT}(z) \rrbracket(l, d, p) &:= (l + 1, d : z, p)\end{aligned}$$

Definition 18.7 (Semantics of AM programs)

An **AM program** is a sequence of $k \geq 1$ labeled AM instructions:

$$P = a_1 : O_{a_1}; \dots; a_k : O_{a_k}$$

where $a_i \in PC$ with $a_i = a_1 + i - 1$ for every $i \in [k]$. The set of all AM programs is denoted by AM .

Definition 18.7 (Semantics of AM programs)

An **AM program** is a sequence of $k \geq 1$ labeled AM instructions:

$$P = a_1 : O_{a_1}; \dots; a_k : O_{a_k}$$

where $a_i \in PC$ with $a_i = a_1 + i - 1$ for every $i \in [k]$. The set of all AM programs is denoted by AM .

The **semantics of AM programs** is determined by

$$\llbracket . \rrbracket : AM \times S \dashrightarrow S$$

with

$$\llbracket P \rrbracket(l, d, p) := \begin{cases} \llbracket P \rrbracket(\llbracket O_l \rrbracket(l, d, p)) & \text{if } l \in \{a_1, \dots, a_k\} \\ (l, d, p) & \text{otherwise} \end{cases}$$

- 1 Repetition: Intermediate Code for EPL
- 2 The Procedure Stack
- 3 Translation of EPL into AM Programs

Translation of EPL into AM Programs

Goal: define translation mapping

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

Translation of EPL into AM Programs

Goal: define **translation mapping**

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

The translation employs a **symbol table**:

$$\begin{aligned} Tab := \{st \mid st : Ide \dashrightarrow (\{ \mathbf{const} \} \times \mathbb{Z}) \\ \cup (\{ \mathbf{var} \} \times Lev \times Off) \\ \cup (\{ \mathbf{proc} \} \times PC \times Lev \times Size) \} \end{aligned}$$

whose entries are created by declarations and are used for translating commands:

variable declarations: **declaration level** $lev \in Lev := \mathbb{N}$,

offset $off \in Off := \mathbb{N}$

(offset and difference between usage and declaration level
determine procedure stack entry)

procedure declarations: **code address** $ca \in PC$, **declaration level**

$lev \in Lev$, **number of local variables** $loc \in Size := \mathbb{N}$

Maintaining the Symbol Table

The symbol table is maintained by the function $\text{update}(D, \text{st}, l)$ which specifies the update of symbol table st according to declaration D (with respect to current level l):

Maintaining the Symbol Table

The symbol table is maintained by the function $\text{update}(D, \text{st}, l)$ which specifies the update of symbol table st according to declaration D (with respect to current level l):

Definition 18.8 (update function)

$$\text{update} : Dcl \times Tab \times Lev \dashrightarrow Tab$$

is defined by

$$\begin{aligned} & \text{update}(D_C \ D_V \ D_P, \text{st}, l) \\ & \quad := \text{update}(D_P, \text{update}(D_V, \text{update}(D_C, \text{st}, l), l), l) \\ & \quad \quad \text{if all identifiers in } D_C \ D_V \ D_P \text{ different} \\ & \text{update}(\varepsilon, \text{st}, l) \\ & \quad := \text{st} \\ & \text{update}(\text{const } I_1 := z_1, \dots, I_n := z_n; \text{st}, l) \\ & \quad := \text{st}[I_1 \mapsto (\text{const}, z_1), \dots, I_n \mapsto (\text{const}, z_n)] \\ & \text{update}(\text{var } I_1, \dots, I_n; \text{st}, l) \\ & \quad := \text{st}[I_1 \mapsto (\text{var}, l, 1), \dots, I_n \mapsto (\text{var}, l, n)] \\ & \text{update}(\text{proc } I_1; K_1; \dots; I_n; K_n; \text{st}, l) \\ & \quad := \text{st}[I_1 \mapsto (\text{proc}, a_1, l, \text{size}(K_1)), \dots, I_n \mapsto (\text{proc}, a_n, l, \text{size}(K_n))] \\ & \quad \quad \text{with “fresh” addresses } a_1, \dots, a_n \\ & \quad \quad \text{where } \text{size}(D_C \ \text{var } I_1, \dots, I_n; D_P C) := n \end{aligned}$$

The Initial Symbol Table

By Definition 17.9, an EPL program $P = \text{in/out } I_1, \dots, I_n; K. \in Pgm$ has the semantics $\mathfrak{M}[[P]] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

The Initial Symbol Table

By Definition 17.9, an EPL program $P = \text{in/out } I_1, \dots, I_n; K. \in Pgm$ has the **semantics** $\mathfrak{M}[[P]] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

Given $(z_1, \dots, z_n) \in \mathbb{Z}^n$, we choose the **initial state**

$$s := (1, \varepsilon, \underbrace{0 : 0 : 0 : z_1 : \dots : z_n}_{\text{I/O frame}}) \in S = PC \times DS \times PS$$

The Initial Symbol Table

By Definition 17.9, an EPL program $P = \text{in/out } I_1, \dots, I_n; K. \in Pgm$ has the **semantics** $\mathfrak{M}[[P]] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

Given $(z_1, \dots, z_n) \in \mathbb{Z}^n$, we choose the **initial state**

$$s := (1, \varepsilon, \underbrace{0 : 0 : 0 : z_1 : \dots : z_n}_{\text{I/O frame}}) \in S = PC \times DS \times PS$$

Thus the corresponding **initial symbol table** has n entries:

$$\text{st}_{\text{I/O}}(I_j) := (\text{var}, 0, j) \quad \text{for every } j \in [n]$$

Translation of `in/out  $I_1, \dots, I_n; D C$` :

- 1 Create MAIN frame for executing C
- 2 Stop program execution after return

Translation of `in/out  $I_1, \dots, I_n; D C$` :

- 1 Create MAIN frame for executing C
- 2 Stop program execution after return

Definition 18.9 (Translation of programs)

The mapping

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

is defined by

$$\begin{aligned} \text{trans}(\text{in/out } I_1, \dots, I_n; K.) &:= 1 : \text{CALL}(a, 0, \text{size}(K)); \\ &\quad 2 : \text{JMP}(0); \\ &\quad \text{kt}(K, \text{st}_{I/O}, a, 1) \end{aligned}$$

Translation of D C :

- 1 Update symbol table according to D
- 2 Create code for procedures declared in D
(using the updated symbol table – recursion!)
- 3 Create code for C (using the updated symbol table)

Translation of Blocks

Translation of $D \ C$:

- 1 Update symbol table according to D
- 2 Create code for procedures declared in D
(using the updated symbol table – recursion!)
- 3 Create code for C (using the updated symbol table)

Definition 18.10 (Translation of blocks)

The mapping

$$kt : Block \times Tab \times PC \times Lev \dashrightarrow AM$$

is defined by

$$\begin{aligned} kt(D \ C, st, a, l) := & \text{dt}(D, \text{update}(D, st, l), l) \\ & \text{ct}(C, \text{update}(D, st, l), a, l) \\ & a' : \text{RET}; \end{aligned}$$

Translation of Declarations

Translation of D :

- Generate code for the procedures declared in D

Translation of Declarations

Translation of D :

- Generate code for the procedures declared in D

Definition 18.11 (Translation of declarations)

The mapping

$$dt : Dcl \times Tab \times Lev \dashrightarrow AM$$

is defined by

$$dt(D_C \ D_V \ D_P, st, l)$$

$$:= dt(D_P, st, l)$$

$$dt(\varepsilon, st, l)$$

$$:= \varepsilon$$

$$dt(\text{proc } I_1; K_1; \dots; I_n; K_n, st, l)$$

$$:= kt(K_1, st, a_1, l + 1)$$

$$\vdots$$

$$kt(K_n, st, a_n, l + 1)$$

where $st(I_j) = (\text{proc}, a_j, \dots, \dots)$ for every $j \in [n]$

Definition 18.12 (Translation of commands)

The mapping

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

is defined by

$$\begin{aligned} ct(I := A, st, a, l) &:= at(A, st, a, l) \\ &\quad a' : STORE(l - lev, off); \\ &\quad \text{if } st(I) = (\text{var}, lev, off) \\ ct(I(), st, a, l) &:= a : CALL(ca, l - lev, loc); \\ &\quad \text{if } st(I) = (\text{proc}, ca, lev, loc) \\ ct(C_1; C_2, st, a, l) &:= ct(C_1, st, a, l) \\ &\quad ct(C_2, st, a', l) \\ ct(\text{if } B \text{ then } C_1 \text{ else } C_2, st, a, l) &:= bt(B, st, a, l) \\ &\quad a' : JFALSE(a''); \\ &\quad ct(C_1, st, a' + 1, l) \\ &\quad a'' - 1 : JMP(a'''); \\ &\quad ct(C_2, st, a'', l) \\ &\quad a''' : \\ ct(\text{while } B \text{ do } C, st, a, l) &:= bt(B, st, a, l) \\ &\quad a' : JFALSE(a'' + 1); \\ &\quad ct(C, st, a' + 1, l) \\ &\quad a'' : JMP(a); \end{aligned}$$

Definition 18.13 (Translation of Boolean expressions)

The mapping

$$\text{bt} : BExp \times Tab \times PC \times Lev \dashrightarrow AM$$

is defined by

$$\begin{aligned} \text{bt}(A_1 < A_2, \text{st}, a, l) &:= \text{at}(A_1, \text{st}, a, l) \\ &\quad \text{at}(A_2, \text{st}, a', l) \\ &\quad a'' : \text{LT}; \end{aligned}$$

$$\begin{aligned} \text{bt}(\text{not } B, \text{st}, a, l) &:= \text{bt}(B, \text{st}, a, l) \\ &\quad a' : \text{NOT}; \end{aligned}$$

$$\begin{aligned} \text{bt}(B_1 \text{ and } B_2, \text{st}, a, l) &:= \text{bt}(B_1, \text{st}, a, l) \\ &\quad \text{bt}(B_2, \text{st}, a', l) \\ &\quad a'' : \text{AND}; \end{aligned}$$

$$\begin{aligned} \text{bt}(B_1 \text{ or } B_2, \text{st}, a, l) &:= \text{bt}(B_1, \text{st}, a, l) \\ &\quad \text{bt}(B_2, \text{st}, a', l) \\ &\quad a'' : \text{OR}; \end{aligned}$$

Translation of Arithmetic Expressions

Definition 18.14 (Translation of arithmetic expressions)

The mapping

$$\text{at} : AExp \times Tab \times PC \times Lev \dashrightarrow AM$$

is defined by

$$\begin{aligned} \text{at}(z, \text{st}, a, l) &:= a : \text{LIT}(z); \\ \text{at}(I, \text{st}, a, l) &:= \begin{cases} a : \text{LIT}(z); & \text{if } \text{st}(I) = (\text{const}, z) \\ a : \text{LOAD}(l - lev, off); & \text{if } \text{st}(I) = (\text{var}, lev, off) \end{cases} \\ \text{at}(A_1 + A_2, \text{st}, a, l) &:= \begin{aligned} &\text{at}(A_1, \text{st}, a, l) \\ &\text{at}(A_2, \text{st}, a', l) \\ &a'' : \text{ADD}; \end{aligned} \end{aligned}$$