4.5 The switch-Statement

Idea:
- Multi-target branching in constant time!
- Use a jump table, which contains at its i-th position the jump to the beginning of the i-th alternative.
- Realized by indexed jumps.

\[
\text{PC} = B + S[SP];
\]

\[
\text{SP}--;
\]

Simplification:

We only regard switch-statements of the following form:

\[
s \equiv \text{switch}(e) \{
\]
\[
\quad \text{case } 0: \text{break;}
\]
\[
\quad \text{case } 1: \text{break;}
\]
\[
\quad \cdots
\]
\[
\quad \text{case } k - 1: \text{break;}
\]
\[
\quad \text{default: } S_{S_{k}}
\]
\[
\}
\]

\[
s \text{ is then translated into the instruction sequence:}
\]

\[
\text{code } s \rho = \text{code}_0 \quad \rho \quad \text{check } 0 \quad k \quad B \quad \text{jump } D \quad \text{...}
\]

\[
\quad \text{check } k \quad B \quad \text{jump } C_k \quad \text{...}
\]

\[
\quad \text{check } k \quad D \quad \text{jump } D
\]

- The Macro check \([0, k]\) checks whether the R-value of \(e\) is in the interval \([0, k]\), and executes an indexed jump into the table \(B\).
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the switch-statement.
check 0 k B =
\[
\begin{align*}
\text{dup} & \quad \text{dup} & \quad \text{jumpi B} \\
\text{loadc 0} & \quad \text{loadc k} & \quad \text{A:} \quad \text{pop} \\
go\quad \leq & \quad \text{loadc k} & \\
jumpz A & \quad \text{jumpz A} & \quad \text{jumpi B}
\end{align*}
\]
\[
\text{jump (} B+2) \text{)}
\]

- The R-value of \( r \) is still needed for indexing after the comparison. It is therefore copied before the comparison.
- This is done by the instruction \text{dup}.
- The R-value of \( r \) is replaced by \( k \) before the indexed jump is executed if it is less than 0 or greater than \( k \).

\[
\begin{align*}
\text{dup} & \quad \text{dup} & \quad \text{jumpi B} \\
\text{loadc 0} & \quad \text{loadc k} & \quad \text{A:} \quad \text{pop} \\
go\quad \leq & \quad \text{loadc k} & \\
jumpz A & \quad \text{jumpz A} & \quad \text{jumpi B}
\end{align*}
\]

\[
\text{S}[\text{SP}+1] = \text{S}[\text{SP}] ; \\
\text{SP++ ;}
\]

\[
\begin{align*}
\text{code s \( \rho \) = } & \quad \text{code s \( \rho \)} & \quad \text{C_0:} & \quad \text{code s \( \rho \) B:} & \quad \text{jump C_0} \\
\text{check 0 \( \ell \) B } & \quad \text{jump D} & \quad \ldots & \quad \text{jump C_4} & \quad D: \quad \ldots \\
& \quad \text{C_4:} & \quad \text{code s \( \ell \) \( \rho \) } & \quad \text{jump D}
\end{align*}
\]

- The Macro \text{check 0 \( \ell \) B} checks whether the R-value of \( r \) is in the interval [0, k], and executes an indexed jump into the table \text{B}.
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the switch-statement.
Note:

- The jump table could be placed directly after the code for the Macro `check`. This would save a few unconditional jumps. However, it may require to search the `switch`-statement twice.
- If the table starts with \( n \) instead of 0, we have to decrease the R-value of \( e \) by \( n \) before using it as an index.
- If all potential values of \( e \) are `definitely` in the interval \([0, k]\), the macro `check` is not needed.

Simplification:

We only regard `switch`-statements of the following form:

\[
s \equiv \begin{array}{l}
\text{switch} \ (e) \{ \\
\quad \text{case} \ 0: \ s_0; \ \text{break}; \\
\quad \text{case} \ 1: \ s_1; \ \text{break}; \\
\quad \vdots \\
\quad \text{case} \ k-1: \ s_{k-1}; \ \text{break}; \\
\quad \text{default:} \ s_k \\
\}\end{array}
\]

\( s \) is then translated into the instruction sequence:

```
check 0 k B = dup dup jumpi B
      loadc 0 loadc A pop
      geq le loadc k
      jumpz A jumpz A jumpi B
```

Note:

- The R-value of \( e \) is still needed for indexing after the comparison. It is therefore copied before the comparison.
- This is done by the instruction `dup`.
- The R-value of \( e \) is replaced by \( k \) before the indexed jump is executed if it is less than 0 or greater than \( k \).
5 Storage Allocation for Variables

Goal:
Associate statically, i.e. at compile time, with each variable \( x \) a fixed (relative) address \( \rho x \).

Assumptions:
- variables of basic types, e.g. int, ... occupy one storage cell.
- variables are allocated in the store in the order, in which they are declared, starting at address 1.

Consequently, we obtain for the declaration \( \bar{d} \equiv t_1 x_1 ; \ldots ; t_k x_k \) (\( t_i \) basic type) the address environment \( \rho \) such that
\[
\rho x_i = i, \quad i = 1, \ldots, k
\]

\[\text{...}\]

5.1 Arrays

Example

The array \( a \) consists of 11 components and therefore needs 11 cells.
\( \rho a \) is the address of the component \( a[0] \).

\[a[10]\]
\[::\]
\[a[0]\]

\[\text{...}\]

Task:

Extend \( \text{code}_e \) and \( \text{code}_l \) to expressions with accesses to array components.

Be \( t[e] a \); the declaration of an array \( a \).
To determine the start address of a component \( a[i] \), we compute
\( \rho a + |t| \times (R-value of i) \).

In consequence:
\[
\text{code}_e e[e] \rho = \text{load} (\rho a) \\
\text{code}_e e[e] \rho = \text{load} (\rho a) \\
\text{load} (t) \\
\text{mul} \\
\text{add}
\]

... or more general.
5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere. Alternatively, one could manage a separate environment $\rho_{st}$ for each structure type $st$.

Be $\text{struct } \{ \text{int } a; \text{ int } b; \} x; \text{ part of a declaration list.}$

• $x$ has as relative address the address of the first cell allocated for the structure.

• The components have addresses relative to the start address of the structure. In the example, these are $a \mapsto 0$, $b \mapsto 1$.
Example

Be \( \text{struct} \{ \text{int} a; \text{int} b; \} x; \) such that \( \rho = \{ x \mapsto 13, a \mapsto 0, b \mapsto 1 \} \).

This yields:

\[
\text{code}_L \ (x,b) \ \rho \ =
\begin{align*}
\text{loadc} & \ 13 \\
\text{loadc} & \ 1 \\
\text{add} & 
\end{align*}
\]

6 Pointer and Dynamic Storage Management

Pointer allow the access to anonymous, dynamically generated objects, whose life time is not subject to the LIFO-principle.

\[\longrightarrow\quad \text{We need another potentially unbounded storage area } H \quad \text{the Heap.}\]

\[\begin{array}{cccc}
S & & & H \\
0 & & & \text{MAX} \\
SP & \text{EP} & \text{NP} & \\
\end{array}\]

\(\text{NP} \equiv \text{New Pointer;} \) points to the lowest occupied heap cell.

\(\text{EP} \equiv \text{Extreme Pointer;} \) points to the uppermost cell, to which \(SP\) can point (during execution of the actual function).
What can we do with pointers (pointer values)?

- **set** a pointer to a storage cell,
- **dereference** a pointer, access the value in a storage cell pointed to by a pointer.

There are two ways to set a pointer:

1. A call `malloc(e)` reserves a heap area of the size of the value of `e` and returns a pointer to this area:
   
   ```
   codeR malloc(e) = codeR e ρ
   new
   ```

2. The application of the address operator `&` to a variable returns a pointer to this variable, i.e. its address (≡ L-value). Therefore:
   
   ```
   codeR (&e) ρ = codeR e ρ
   ```

---

**Idea:**

- Stack and Heap grow toward each other in S, but must not collide. (Stack Overflow).
- A collision may be caused by an increment of `SP` or a decrement of `NP`.
- `EP` saves us the check for collision at the stack operations.
- The checks at heap allocations are still necessary.

---

Null is a special pointer constant, identified with the integer constant 0.
In the case of a collision of stack and heap the NULL-pointer is returned.
Dereferencing of Pointers:

The application of the operator \( * \) to the expression \( e \) returns the contents of the storage cell, whose address is the R-value of \( e \):

\[
\text{code}_L (e) \rho = \text{code}_R e \rho
\]

**Example**  
Given the declarations

\[
\begin{align*}
\text{struct } t &\{ \text{int } a[i]; \text{struct } t * b; \}; \\
\text{int } i, j;
\end{align*}
\]

and the expression \((pt \to b) \to a)[i + 1]\n
Because of \( e \to a = (\ast e).a \) holds:

\[
\begin{align*}
\text{code}_L (e \to a) \rho &= \text{code}_R e \rho \\
\text{loadc} (\rho a) &\quad \text{add}
\end{align*}
\]

What can we do with pointers (pointer values)?

- **set** a pointer to a storage cell,
  
  \[
  \text{code}_L (\text{set } x) = \text{code}_R (x)
  \]

- **dereference** a pointer, access the value in a storage cell pointed to by a pointer.
  
  \[
  \text{code}_L (x) = \text{code}_R (x)
  \]

There are two ways to set a pointer:

1. A call \texttt{malloc}(\(e\)) reserves a heap area of the size of the value of \(e\) and returns a pointer to this area:

\[
\text{code}_L \text{malloc}(\(e\)) \rho = \text{code}_R e \rho \\
\text{new}
\]

(2) The application of the address operator \& to a variable returns a pointer to this variable, i.e. its address (\(\equiv \text{L-value}\)). Therefore:

\[
\text{code}_L (\&e) \rho = \text{code}_R e \rho
\]

Dereferencing of Pointers:

The application of the operator \( * \) to the expression \( e \) returns the contents of the storage cell, whose address is the R-value of \( e \):

\[
\text{code}_L (e) \rho = \text{code}_R e \rho
\]

**Example**  
Given the declarations

\[
\begin{align*}
\text{struct } t &\{ \text{int } a[i]; \text{struct } t * b; \}; \\
\text{int } i, j;
\end{align*}
\]

and the expression \((pt \to b) \to a)[i + 1]\n
Because of \( e \to a = (\ast e).a \) holds:

\[
\begin{align*}
\text{code}_L (e \to a) \rho &= \text{code}_R e \rho \\
\text{loadc} (\rho a) &\quad \text{add}
\end{align*}
\]