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.

\[
\begin{align*}
PC &= B + S[SP]; \\
SP &\rightarrow; \\
\end{align*}
\]

Simplification:

We only regard switch-statements of the following form:

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

$s$ is then translated into the instruction sequence:

\[
\begin{align*}
code s \rho &= \text{code } e \rho \\
C_0: & \text{code } s_0 \rho \\
\text{check } 0 \ k \ B \\
jump D \\
\vdots \\
\text{jump } C_d \\
C_k: & \text{code } s_k \rho \\
jump D \\
\end{align*}
\]

- The Macro $\text{check } 0 \ k \ B$ 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.
code s \rho = \text{code} s \rho \qquad C_0: \text{code} s \tau \rho \qquad B: \text{jump} C_0
\begin{align*}
\text{check } 0 & k B \\
& \text{jump } D \\
& \ldots \\
& \text{jump } C_d \\
C_d: & \text{code } s \tau \rho \\
& \text{jump } D
\end{align*}

- The Macro \text{check } 0 k B 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 = \text{dup} \quad \text{dup} \quad \text{jumpi} B
\begin{align*}
\text{load} 0 & \\
\text{load} k & \\
\text{A: pop} \\
\text{geq} & \\
\text{le} & \\
\text{load} k \\
\text{jumpz} A & \\
\text{jumpz} A & \\
& \text{jumpi} B
\end{align*}

- 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 \text{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 \( k \).

Note:
- The jump table could be placed directly after the code for the Macro \text{check}. This would save a few unconditional jumps. However, it may require to search the switch-statement twice.
- If the table starts with \( k \) instead of 0, we have to decrease the R-value of \( e \) by \( k \) before using it as an index.
- If all potential values of \( e \) are definitely in the interval \([0, k]\), the macro \text{check} is not needed.
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. \( \text{int} \), … 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 \( d = t_1 x_1; \ldots; t_k x_k; \) (\( t_i \) basic type) the address environment \( \rho \) such that \( \rho x_i = i_1, \ldots, k \).

5.1 Arrays

Example: \( \text{int}[11] a; \)
The array \( a \) consists of 11 components and therefore needs 11 cells.
\( \rho a \) is the address of the component \( a[0] \).

We need a function \( \text{sizeof} \) (notation: \( \cdot \cdot \cdot \)), computing the space requirement of a type:

\[
|t| = \begin{cases} 
1 & \text{if } t \text{ basic} \\
1 + |t'| & \text{if } t = t'[k] 
\end{cases}
\]

Accordingly, we obtain for the declaration \( d \equiv t_1 x_1; \ldots; t_k x_k; \)

\[
\rho x_1 = 1 \\
\rho x_i = \rho x_{i-1} + |t_{i-1}| 
\]

for \( i > 1 \).

Since \( |t| \) can be computed at compile time, also \( \rho \) can be computed at compile time.
Task:

Extend code₁ and code₂ to expressions with accesses to array components.

Be \( \tau[\ell] a; \) the declaration of an array \( a \).

To determine the start address of a component \( a[i] \), we compute
\( \rho a + |i| \times (\text{R-value of } i) \).

In consequence:

\[
\text{code₁} \; a[i] \; \rho = \begin{array}{ll}
\text{loadc} \; (\rho a) \\
\text{code₂} \; e \; \rho \\
\text{loadc} \; |i| \\
\text{mul} \\
\text{add}
\end{array}
\]

... or more general:

\[
\text{code₁} \; e_1[e_2] \; \rho = \begin{array}{ll}
\text{code₂} \; e_1 \; \rho \\
\text{loadc} \; |i| \\
\text{mul} \\
\text{add}
\end{array}
\]

Remark:

- In C, an array is a pointer. A declared array \( a \) is a pointer-constant, whose R-value is the start address of the array.
- Formally, we define for an array \( e \):
  \( \text{code₂} \; e \; \rho = \text{code₁} \; e \; \rho \)
- In C, the following are equivalent (as L-values):
\[
2[a[2]] \; a + 2
\]

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.

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 \( \rho_s \) for each structure type \( st \).

Be \( \text{struct} \{ \text{int } a; \text{int } b; \} \; x; \) 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 \( a \rightarrow 0, b \rightarrow 1 \).
Let \( t \equiv \text{struct}\{t_1, t_2, \ldots t_k\} \). We have

\[
|t| = \sum_{i=1}^{k} |t_i| \\
\rho_{t_1} = 0 \quad \text{and} \\
\rho_{t_i} = \rho_{t_{i-1}} + |t_{i-1}| \quad \text{for} \ i > 1
\]

We thus obtain:

\[
\text{code}_1((c, c) \, \rho) = \text{code}_1(c \, \rho) \\
\text{load}(\rho \, c) \\
\text{add}
\]

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 \( \rho_e \) for each structure type \( s_t \).

Be \( \text{struct}\{\text{int}\, a; \text{int}\, b; \} \, x; \) \; 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 \( a \rightarrow 0, b \rightarrow 1 \).

Example:

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

This yields:

\[
\text{code}_1((x, b) \, \rho) = \\
\text{load} \, 13 \\
\text{load} \, 1 \\
\text{add}
\]
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.

We need another potentially unbounded storage area \( H \) – the Heap.

\[
\begin{array}{c}
| & S & & H \\
\hline
0 & & \text{SP} & \text{EP} & \text{NP} \\
\hline
\end{array}
\]

\( \text{NP} \equiv \text{New Pointer}; \) 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).

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.

\[
\begin{array}{c}
| & S & & H \\
\hline
0 & & \text{SP} & \text{EP} & \text{NP} \\
\hline
\end{array}
\]

\( \text{NP} \equiv \text{New Pointer}; \) 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).
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.

We need another potentially unbounded storage area H – the Heap.

NP \equiv New\ Pointer; \ points\ to\ the\ lowest\ occupied\ heap\ cell.
EP \equiv 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:
   \[
   \text{code}_{\text{malloc}}(e) \quad \rho = \text{code}_{\text{new}}(e) \quad \rho
   \]

2. The application of the address operator `&` to a variable returns a pointer to this variable, i.e. its address (\(\equiv\) L-value). Therefore:
   \[
   \text{code}_{\text{&}}(e) \quad \rho = \text{code}_{\text{e}} \quad \rho
   \]

```c
NP -> [n]  

if (NP - S[SP] \leq EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

- 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.
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:

   ```
   \text{code}_E \text{malloc}(e) \rho = \text{code}_E e \rho
   
   \text{new}
   ```

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

   ```
   \text{code}_E (\&e) \rho = \text{code}_E e \rho
   ```

- 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.