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.

![Diagram]

PC = B + S[SP];
SP--;
Simplification

We only regard switch statements of the following form:

\[
    s = \text{switch}(r) \begin{cases}
        \text{case } 0: & s_{0}, \text{ break; } \\
        \text{case } 1: & s_{1}, \text{ break; } \\
        \quad \vdots \\
        \text{case } k-1: & s_{k-1}, \text{ break; } \\
        \text{default: } & s_{k}\ \text{break;}
    \end{cases}
\]

$s$ is then translated into the instruction sequence:

\[
    \text{check } 0 \ k \ B = \begin{cases}
        \text{dup} & \text{dup} & \text{jumpi } B \\
        \text{loadc } 0 \ & \text{loadc } k \ & \text{A}, \text{ pop} \\
        \text{geq} \ & \text{le} \ & \text{loadc } k \\
        \text{jumpz } A \ & \text{jumpz } A \ & \text{jumpi } B
    \end{cases}
\]
check 0 k B  =  dup  dup  jumpi B
    load 0  load k  A:  pop
    geq  le  load k
    jump A  jump A  jumpi B

- The R-value of $\epsilon$ 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 $\epsilon$ is replaced by $k$ before the indexed jump is executed if it is less than 0 or greater than $k$.

\[
\text{code } s \rho = \begin{array}{lll}
\text{code } s \rho \quad C_0: & \text{code } s \rho \quad B: & \text{jump } C_0 \\
\text{check } 0 k B & \text{jump } D & \ldots \\
\quad \ldots & \text{jump } C_2 & \\
\end{array}
\]

- The Macro `check 0 k B` checks whether the R-value of $\epsilon$ 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.

Simplification

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

\[
s = \begin{array}{l}
\text{switch } (\epsilon) \\
\text{case 0: } s_0 \text{ break;}
\text{case 1: } s_1 \text{ break;}
\ldots
\text{case } k - 1 : s_{k-1} \text{ break;}
\text{default: } s_k
\end{array}
\]

$s$ is then translated into the instruction sequence:

Remark

- 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 $u$ instead of 0, we have to decrease the R-value of $\epsilon$ by $u$ before using it as an index.
- If all potential values of $\epsilon$ are definitely in the interval $[0, k]$, the macro `check` is not needed.
code $s \rho = \text{code}_e e \rho$
check 0 $k$ B
\begin{align*}
C_0: & \text{code } s_0 \rho \\
B: & \text{jump } C_0 \\
& \text{jump } D \\
& \cdots \\
& \text{jump } C_k \\
C_k: & \text{code } s_k \rho \\
D: & \cdots
\end{align*}

- The Macro 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.

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 $d \equiv t_1 \ x_1 ; \ldots ; t_k \ x_k$ (ti basic type) the address environment $\rho$ such that

\[\rho x_i = i, \quad i = 1, \ldots, k\]

Remark
- 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 $u$ instead of 0, we have to decrease the R-value of e by $u$ 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.

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 $d \equiv t_1 \ x_1 ; \ldots ; t_k \ x_k$ (ti basic type) the address environment $\rho$ such that

\[\rho x_i = i, \quad i = 1, \ldots, k\]
5.1 Arrays

Example

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

\[
\begin{array}{c}
\text{a[10]} \\
\vdots \\
\text{a[0]}
\end{array}
\]

---

Task

Extend \( \text{code}_1 \) and \( \text{code}_2 \) to expressions with accesses to array components.

Be \( \Gamma[c] \quad a \); the declaration of an array \( a \).

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

In consequence:

\[
\text{code}_1 \ a[i] \quad \rho = \quad \text{load} \ (\rho a) \\
\text{ code}_2 \ e \quad \rho \\
\text{ load} \ |i| \\
\text{mul} \\
\text{add}
\]

... or more general:

---

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

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

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

\[
\begin{align*}
\rho x_1 &= 1 \\
\rho x_i &= \rho x_{i-1} + |t_{i-1}| & \text{for } i > 1
\end{align*}
\]

Since \( |\cdot| \) can be computed at compile time, also \( \rho \) can be computed at compile time.

---

Task

Extend \( \text{code}_1 \) and \( \text{code}_2 \) to expressions with accesses to array components.

Be \( \Gamma[c] \quad a \); the declaration of an array \( a \).

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

In consequence:

\[
\text{code}_1 \ a[i] \quad \rho = \quad \text{load} \ (\rho a) \\
\text{ code}_2 \ e \quad \rho \\
\text{ load} \ |i| \\
\text{mul} \\
\text{add}
\]

... or more general:
We need a function `sizeof` (notation: `|·|`), computing the space requirement of a type:

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

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

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

Since `|·|` can be computed at compile time, also \( \rho \) can be computed at compile time.

\[\text{code}_2 e_1[e_2] \rho = \text{code}_2 e_1 \rho \\
\text{code}_2 e_2 \rho \\
\text{loadc}(\rho e) \\
mul \\
\text{add}\]

\[\text{code}_2 e \rho = \text{loadc}(\rho e) \\
\text{loadc}|e| \\
mul \\
\text{add}\]

\[\text{... or more general:}\]

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

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

- \( x \) 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 \( a \mapsto 0, b \mapsto 1 \).
Let \( t \equiv \text{struct } \{ t_1 \; c_1; \ldots \; t_k \; c_k \} \). We have

\[
|t| = \sum_{i=1}^{k} |t_i|
\]

\[\rho c_1 = 0 \quad \text{and} \]
\[\rho c_i = \rho c_{i-1} + |t_{i-1}| \quad \text{for } i > 1\]

We thus obtain:

\[
\text{code}_L (e.c) \ \rho = \text{code}_L e \ \rho \\
\qquad \text{loadc } (\rho c) \\
\qquad \text{add}
\]

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:

\[
\text{code}_L (x.b) \ \rho = \text{loadc } 13 \\
\qquad \text{loadc } 1 \\
\qquad \text{add}
\]

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.

\[
\begin{array}{c}
\text{S} \\
0 \\
\text{H} \\
MAX \\
\end{array}
\]

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

\( \text{EP} \) \( = \) 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.

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

   \[
   \text{code}_R \texttt{malloc}(e) \rho \rightarrow \text{code}_R \ e \rho \quad \text{new}
   \]

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

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