Script generated by TTT

Title: Petter: Virtual Machines (06.05.2019)

Date: Mon May 06 10:16:19 CEST 2019

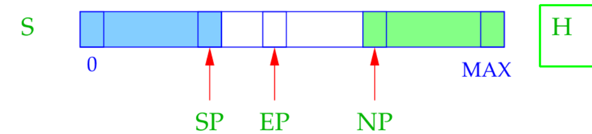
Duration: 92:40 min

Pages: 12

6 Pointers and Dynamic Storage Management

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

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



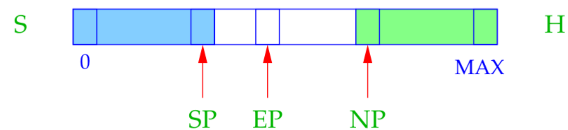
NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

6 Pointers and Dynamic Storage Management

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

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



NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ 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 a 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:

$$\text{code}_R\ \textbf{malloc}\,(e)\ \rho \;=\; \text{code}_R\ e\ \rho$$
$$\text{new}$$

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

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

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 a two ways to set a pointer:

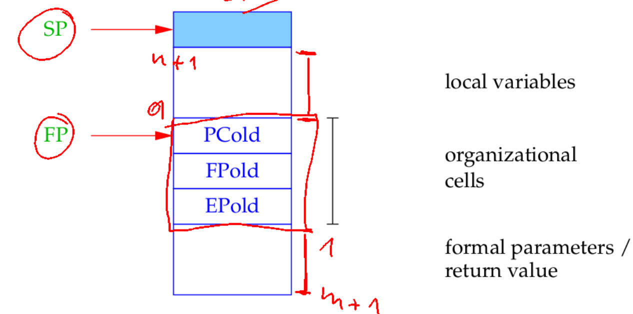
(1) A call **malloc** (e) reserves a heap area of the size of the value of e and returns a pointer to this area:

$$\text{code}_R\ \textbf{malloc}\,(e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{new}$$

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

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

9.1 Memory Organization for Functions



FP ≙ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

Caveat

- The local variables receive relative addresses $+1, +2, \ldots$.
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP.
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for `printf`.
- The memory block of parameters is recycled for storing the return value of the function.

  Simplification: The return value fits into a single memory cell.

Tasks of a Translator for Functions

- Generate code for the body of the function!
- Generate code for calls!

9.2 Determining Address Environments

We distinguish two kinds of variables:

1. global/extern that are defined outside of functions;
2. local/intern/automatic (inkluding formal parameters) which are defined inside functions.

The address environment ρ maps names onto pairs $(tag, a) \in \{G, L\} \times \mathbb{Z}$.

Caveat

- In general, there are further refined grades of visibility of variables.
- Different parts of a program may be translated relative to different address environments!

Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

2 Inside of main:

$$\begin{array}{llll} \rho_2 : & i & \mapsto & (G,1) \\ & l & \mapsto & (G,2) \\ & k & \mapsto & (L,1) \\ & \text{ith} & \mapsto & (G,\text{_ith}) \\ & \text{main} & \mapsto & (G,\text{_main}) \\ & & \ldots & \end{array}$$

Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

2 Inside of main:

$$\begin{array}{llll} \rho_2 : & i & \mapsto & (G,1) \\ & l & \mapsto & (G,2) \\ & k & \mapsto & (L,1) \\ & \text{ith} & \mapsto & (G,\text{_ith}) \\ & \text{main} & \mapsto & (G,\text{_main}) \\ & & \ldots & \end{array}$$

9.3 Calling/Entering and Exiting/Leaving Functions

Assume that f is the current function, i.e., the caller, and f calls the function g, i.e., the callee.

The code for the call must be distributed between the caller and the callee.

The distribution can only be such that the code depending on information of the caller must be generated for the caller and likewise for the callee.

Caveat

The space requirements of the actual parameters is only known to the caller ...

Remark

- Of every expression which is passed as a parameter, we determine the R-value
  $\Longrightarrow$ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, whose R-value provids the start address of the called function ...

Remark

- Of every expression which is passed as a parameter, we determine the R-value
  $\Longrightarrow$ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, whose R-value provids the start address of the called function ...