Today's Topics

Previously
• Modelling variables, scopes and visibility using the Run Stack and Display with (LL,ON) addressing

Today
• Modelling storage layout of arrays and records (including classes)
Recall: (LL,ON) Address Calculation

- At run time, the address of a variable in the Run Stack is computed from its (LL,ON).
- RS [Display [LL] + ON] is the storage for variable (LL,ON).
- Display [LL] is the base of the storage for the scope, and ON is the displacement of the particular variable within that storage.
- But what happens if a variable is non-scalar (e.g. array)? => Needs more than one RS slot!
Storage Layout

- When variables are scalar (primitive, unstructured) values, they take only one element of the Run Stack (of course, in practice char and integer scalars may be different sizes, but both still fit in a "word")
- Non-scalar types such as arrays and records (structs, classes) will take more than one element of the stack
- And we get even more complexity due to the fact that elements of an array may be records, and elements of records may be arrays

```plaintext
var r:
  record
    size: integer
    data: array [1..10] of record
      x: integer
      y: integer
    end record
  end record
end
```

- Variables of these types require a contiguous segment of the RS
Storage Layout - Arrays of Scalars

• An array of scalar values is laid out as a sequence of \( N \) contiguous scalars in a row, where \( N \) is the number of elements in the array

\[
\text{var } a: \text{ array } [1..10] \text{ of integer;}
\]

\[
\text{var } y: \text{ integer;}
\]

• If array \( a \) is at lexical level \( LL \) and order number \( ON \), then its storage layout would look like this on the Run Stack

• The order number of the following variable \( y \) is shifted by the number of elements in the array

[Diagram showing storage layout]
Storage Layout - Arrays of Scalars

- Given this representation, the \((\text{LL,ON})\) addresses of \(y\) and \(a\) are:
  \[
  \begin{align*}
  a & \quad \text{(LL,ON)} \\
  y & \quad \text{(LL,ON + 10)}
  \end{align*}
  \]

- The \((\text{LL,ON})\) addresses of the elements of \(a\) are:
  \[
  \begin{align*}
  a[1] & \quad \text{(LL,ON)} \\
  a[2] & \quad \text{(LL,ON + 1)} \\
  a[3] & \quad \text{(LL,ON + 2)} \\
  \vdots & \quad \\
  a[10] & \quad \text{(LL,ON + 9)}
  \end{align*}
  \]

- Or in general:
  \[
  a[i] \quad \text{(LL,ON + i - 1)}
  \]
Storage Layout - Arrays of Scalars

• For example, suppose LL = 1, Display[1] = 22, and ON = 11 then:

ON_y = ON_a + 10 = 21
ON_a[i] = ON_a + i - 1
ON_a[3] = ON_a + 3 - 1

a RS[22 + 11] RS[33]
y RS[22 + 11 + 10] RS[43]
a[1] RS[22 + 11 + 1 - 1] RS[33]
Storage Layout - General 1D Arrays of Scalars

• The general form of a one dimensional array of scalars is:

```
var a: array [lower..upper] of integer;
```

• The size of this array (number of elements) is \((upper - lower + 1)\)
  - this is the number of slots in the RS required for it

• Given: \(\text{address}(a) = (LL, ON)\)
  then: \(\text{address}(a[i]) = \text{address}(a) + i - lower\)
  \(= (LL, ON + i - lower)\)

• So a reference to element \(a[i]\) can be implemented in our abstract machine as:
  
  pushaddress \((LLa, ONa)\) \(\text{i.e., pushaddress } a\)
  push \((LLi, ONi)\) \(\text{i.e., push } i\)
  push lower \(\text{lower bound is a constant}\)
  subtract \(i - lower\)
  add \(\text{address}(a) + i - lower\)
  evaluate \(\text{fetch value of element}\)
Variable Addressing - New Instructions

• **evaluate** is a new abstract machine instruction that pops the top value \( v \) from the Expression Stack and pushes the value of the location in the Run Stack that it is the address of, i.e., \( \text{RunStack}[v] \)

  \[
  \text{push } x \quad \text{pushaddress } x \quad \text{evaluate}
  \]

• Once we have the **pushaddress** instruction, the old **pop** instruction that took a variable as operand is no longer needed - a new and more general instruction called **assign** is used instead

  \[
  x := 1 \quad \text{push } 1 \quad \text{pushaddress } x \\
  \text{pop } x \quad \text{push } 1 \\
  \text{assign}
  \]

• **assign** pops the top two values from the ES, a value (the top element) and an address (second from top), and assigns the value to \( \text{RunStack}[\text{address}] \)
Variable Addressing - New Instructions

- *assign* and *evaluate* allow us to treat computed addresses of array elements and record fields as variables

  \[
  a[i] := 1 \\
  \text{pushaddress } a \\
  \text{push } i \\
  \text{push lower} \\
  \text{subtract} \\
  \text{add} \\
  \text{push } 1 \\
  \text{assign}
  \]

- The *assign* instruction is more general than the *pop* instruction so we remove the *pop* instruction from our model and use *pushaddress*, *evaluate* and *assign* from now on

- Note that *evaluate* and *assign* are also needed for reference parameters - otherwise we could not get or set their values
Multi-Dimensional Arrays of Scalars

• If an array has more than dimension (e.g., a matrix), then we can think of it as an array of arrays

• Example: the matrix

```haskell
var a: array [1..5, 2..10] of integer;
```

is equivalent (in memory storage) to

```haskell
var a: array [1..5] of array [2..10] of integer;
```

• The element (sub)arrays are simply laid out one after the other in the RS
Example: 2-Dimensional Array (Matrix) of Scalars

```
var a:
  array [1..5, 2..10] of integer;

(LLa, ONa) = a[1,*]
```

Subarrays:
- Subarray 1
- Subarray 2
- Subarrays 3 ... 5
Example: 2-Dimensional Array (Matrix) of Scalars

\texttt{var a: array[1..5, 2..10] of integer;}

\begin{itemize}
  \item The address of element \texttt{a[i,j]} is calculated as
    \begin{align*}
    \text{address(a[i,j])} &= \text{address(a[i,*])} + j - 2 \\
    &= \text{address(a)} + (i-1) \times 9 + j - 2
    \end{align*}
  
  where: 2 is the \textit{lower bound} of each \textit{inner} subarray
  
  9 is the \textit{size} of the inner subarray (elements 2 through 10)
  
  1 is the \textit{lower bound} of the \textit{outer} array

  \item In general, for
    \begin{align*}
    \text{address(a[i,j])} &= \text{address(a[i,*])} + j - l_2 \\
    &= \text{address(a)} + (i-l_1) \times (u_2-l_2+1) + j - l_2
    \end{align*}

  (u_2 - l_2+1) is the \textit{size} of the inner array (i.e. the size of each of the
  subarray \textit{elements} of the outer array) - normally a \texttt{constant},
  except when we have a dynamically sized array in languages that allow them
Multi-Dimensional Arrays of Scalars

• The completely general case of a multi-dimensional array is:

```pascal
var a:
    array [l1..u1, l2..u2, ..., ln .. un] of integer
```

• In this case:

```
address (a [i1, i2, ... i_n]) = address (a) + \sum_{k=1}^{n} (i_k - l_k) * s_k
```

where $s_k$ is the size of the subarray elements of the subarray $a [i_1, ..., i_k, *, ..., *]$

• In **Pascal** and **C** this is a compile time constant defined as :

```
S_k = S_{k+1} * (u_{k+1} - l_{k+1} + 1)
S_n = 1
```

which can be calculated at compile time as:

```
S_k = \prod_{j=k}^{n-1} u_{j+1} - l_{j+1} + 1
```
Records

- Records (or structs in C) are similar to classes, but without methods
- Records are allocated in consecutive locations for the each field

```ruby
var r:
  record
    x: integer;
    y: array [1..5] of integer;
    z: real;
  end record
```

![Diagram of record layout]

- `r.y` (LL, ON + 1)
- `r.z` (LL, ON + 6)
- `r, r.x` (LL, ON)
Record Fields

- The address of a field is computed as an offset from the beginning of the record (i.e., the ON within the record scope)

\[
\text{address}(r.z) = \text{address}(r) + \text{offset}(z) = \text{address}(r) + 6
\]

\[
i := r.z \quad \text{pushaddress (LL}_i,\text{ON}_i) \quad \text{address of } i
\]

\[
\text{pushaddress (LL}_r,\text{ON}_r) \quad \text{address of } r
\]

\[
\text{pushliteral 6} \quad \text{offset of } z
\]

\[
\text{add} \quad \text{address of } r.z
\]

\[
\text{evaluate} \quad \text{value of } r.z
\]

\[
\text{assign}
\]

- Sometimes the address of the record is known at compile time, as in our example - other times it is not, for example,

Arrays of records – record address depends on the subscript

Dynamic records (objects) - record address depends on pointer

- When the address of the record is known, the add and pushliteral can be optimized out, to give:

\[
\text{pushaddress (LL}_r,\text{ON}_r+6)
\]
Arrays as Record Fields

```pascal
var r:
  record
    x: integer;
    y: array [1..5] of integer;
    z: real;
  end record

i := r.y[j]

address(r.y[j]) = address(r.y) + j - 1
= address(r) + offset(y) + j - 1
= address(r) + 1 + j - 1
= address(r) + j
```
Arrays as Record Fields

\textbf{var} \ r: \ \
\textbf{record} \ \
x: integer; \ 
y: array [1..5] of integer; \ 
z: real; \ 
\textbf{end record} \ 

\[ i := r.y[j] \]

\textbf{pushaddress} \ i \quad \textit{address of i} \ 
\textbf{pushaddress} \ r \quad \textit{address of r} \ 
\textbf{pushliteral} \ 1 \quad \textit{offset of y in r} \ 
\textbf{add} \quad \textit{address of r.y} \ 
\textbf{pushaddress} \ j \quad \textit{value of j} \ 
\textbf{evaluate} \ 
\textbf{pushliteral} \ 1 \quad \textit{lower bound of y} \ 
\textbf{subtract} \ 
\textbf{add} \quad i - \textit{lower} \ 
\textbf{evaluate} \ 
\textbf{assign} \quad \textit{address of r.y[j]} \ 
\textbf{value of r.y[j]} \ 
i := r.y[j] \]
var a: array [2..10] of record
  x: integer;
  y: integer;
end record

Arrays of Records
Arrays of Records

- Indexing of arrays of records must be scaled by the size of the records in the array

```pascal
var a: array [2..10] of record
    x: integer;
    y: integer;
end record

address(a[i].y) = address(a[i]) + offset(y)
= address(a) + (i-2) * recsize + offset(y)
= address(a) + (i - 2) * 2 + 1
```
Arrays of Records

• In general, the array subscripting formulas do not change, so the formula still holds:

\[
\text{address (a} [i_1, i_2, \ldots, i_n]\text{)} = \text{address (a)} + \sum_{k=1}^{n} (i_k - l_k) \times s_k
\]

except that

\[
s_k = s_{k+1} \times (u_{k+1} - l_{k+1} + 1)
\]

\[
s_n = \text{recsize} \quad \text{(instead of 1)}
\]

or more compactly,

\[
s_k = \left[ \prod_{j=k}^{n-1} u_{j+1} - l_{j+1} + 1 \right] \times \text{recsize}
\]
Summary

Arrays and Records

• The storage model for records and arrays treats each like a consecutive sequence of variables for each of the elements or fields.

• Elements are addressed by computing the sum of the base address of the record or array and the offset of the element or field within it.

• Can often optimize by doing constant computations at compile time instead of using abstract machine instructions to compute them at run time.

Next

• Begin Semantic Analysis