Lecture 35: IL for Arrays

One-dimensional Arrays

- How do we process retrieval from and assignment to `x[i]`, for an array `x`?
- We assume that all items of the array have fixed size—S bytes—and are arranged sequentially in memory (the usual representation).
- Easy to see that the address of `x[i]` must be

  $$\&x + S \cdot i,$$

  where `&x` is intended to denote the address of the beginning of `x`.
- Generically, we call such formulae for getting an element of a data structure *access algorithms.*
- The IL might look like this:

$\texttt{cgen(\&A[E],}\ t_0\texttt{):}$

$\quad\texttt{cgen(\&A,}\ t_1\texttt{)}$

$\quad\texttt{cgen(E,}\ t_2\texttt{)}$

$\quad\Rightarrow t_3\ \texttt{:=}\ t_2\ \texttt{*}\ \texttt{S}$

$\quad\Rightarrow t_0\ \texttt{:=}\ t_1\ \texttt{+}\ t_3$

Multi-dimensional Arrays

- A 2D array is a 1D array of 1D arrays.
- Java uses arrays of pointers to arrays for >1D arrays.
- But if row size constant, for faster access and compactness, may prefer to represent an MxN array as a 1D array of M 1D rows of length N (not pointers to rows): *row-major order*...
- Or, as in FORTRAN, a 1D array of N 1D columns of length M: *column-major order.*
- So apply the formula for 1D arrays repeatedly—first to compute the beginning of a row and then to compute the column within that row:

  $$\&A[i][j] = \&A + i \cdot S \cdot N + j \cdot S$$

  for an `M`-row by `N`-column array, where `S`, again, is the size of an individual element.

IL for $M \times N$ 2D array

```
cgen(&e1[e2,e3], t):
    # Compute e1, e2, e3, and N:
    cgen(e1, t1);
    cgen(e2,t2);
    cgen(e3,t3)
    cgen(N, t4) # (N need not be constant)
    ⇒ t5 := t4 * t2
    ⇒ t6 := t5 + t3
    ⇒ t7 := t6 * S
    ⇒ t := t7 + t1
```

Array Descriptors

- Calculation of element address `&e1[e2,e3]` has the form

  $VO + S1 \times e2 + S2 \times e3$

  , where

  – VO (`&e1[0,0]`) is the *virtual origin*.

  – `S1` and `S2` are ***strides***.

  – All three of these are constant throughout the lifetime of the array (assuming arrays of constant size).

- Therefore, we can package these up into an *array descriptor,* which can be passed in lieu of the array itself, as a kind of "*fat pointer*" to the array:

`&e1[0][0]`	S×N	S

Array Descriptors (II)

- Assuming that `e1` now evaluates to the address of a 2D array descriptor, the IL code becomes:

```
cgen(&e1[e2,e3], t):
    cgen(e1, t1);      # Yields a pointer to a descriptor.
    cgen(e2,t2);
    cgen(e3,t3)
    ⇒ t4 := *t1;       # The VO
    ⇒ t5 := *(t1+4)    # Stride #1
    ⇒ t6 := *(t1+8)    # Stride #2
    ⇒ t7 := t5 * t2
    ⇒ t8 := t6 * t3
    ⇒ t9 := t4 + t7
    ⇒ t10:= t9 + t8
```

(Here, we assume 32-bit quantities. Adjust the constants appropriately for 64-bit pointers and/or integers.)

Array Descriptors (III)

- By judicious choice of descriptor values, can make the same formula work for different kinds of array.
- For example, if lower bounds of indices are 1 rather than 0, must compute address

  `&e[1,1] + S1` $\times$ `(e2-1) + S2` $\times$ `(e3-1)`

- But some algebra puts this into the form

  `VO' + S1` $\times$ `e2 + S2` $\times$ `e3`

  where

  `VO' = &e[1,1] - S1 - S2 = &e[0,0]` *(if it existed).*

- So with the descriptor

VO'	S×N	S

  we can use the same code as on the last slide.

- By passing descriptors as array parameters, we can have functions that adapt to many different array layouts automatically.

Other Uses for Descriptors

- No reason to stop with strides and virtual origins: can include other data.
- By adding upper and lower index bounds to a descriptor, can easily implement bounds checking.
- This also allows for runtime queries of array sizes and bounds.
- Descriptors also allow *views* of arrays: nothing prevents multiple descriptors from pointing to the same data.
- This allows effects such as slicing, array reversal, or array transposition without copying data.

Examples

- Consider a simple base array (in C):

```
int data[12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

and descriptor types (including lengths):

```
struct Desc1 { int* V0, int S1, int len1 };
struct Desc2 { int* V0, int S1, int len1, int S2, int len2 };
```

- Here are some views:

```
Desc1 v0 = { data, 4, 12 }; /* All of data. */
Desc1 v1 = { &data[3], 4, 3 }; /* data[3:6]: [4, 5, 6]. */
/* Every other element of data: [1, 3, ...] */
Desc1 v2 = { data, 8, 6 };
Desc1 v3 = { &data[11], -4, 12 }; /* Reversed: [12, 11, ...] */

/* As a 2D 4x3 array: [ [ 1, 2, 3 ], [ 4, 5, 6 ], ... ] */
Desc2 v4 = { data, 12, 4, 4, 3 };
/* As row 2 of v4: [7, 8, 9] */
Desc1 v5 = { &data[6], 4, 3 }
```