Lecture 21: 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:
  $$\text{cgen}(\&A[E], t_0):$$
  $$\text{cgen}(\&A, t_1)$$
  $$\text{cgen}(E, t_2)$$
  $$\Rightarrow t_3 := t_2 \cdot S$$
  $$\Rightarrow t_0 := t_1 + t_3$$

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 $M \times N$ array as a 1D array of 1D rows (not pointers to rows): row-major order...
- Or, as in FORTRAN, a 1D array of 1D columns: 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:
  $$\&A[i][j] = \&A + i \cdot S + j \cdot S$$
  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

$$\text{cgen}(\&e1[e2,e3], t):$$
$$\text{cgen}(e1, t_1); \text{cgen}(e2, t_2); \text{cgen}(e3, t_3)$$
$$\text{cgen}(N, t_4) \# (N need not be constant)$$
$$\Rightarrow t_5 := t_4 + t_2$$
$$\Rightarrow t_6 := t_5 + t_3$$
$$\Rightarrow t_7 := t_6 + S$$
$$\Rightarrow t := t_7 + t_1$$
**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:

```
&[0][0]  S1  S
```

**Array Descriptors (II)**

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

```plaintext
cgen(&e1[e2,e3], t):
cgen(e1, t1); 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
```

**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

$$VO' = &e[1,1] - S1 - S2 = &e[0,0] \text{ (if it existed).}$$

• So with the descriptor

```
VO'  S1  S
```

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