Lecture #23: Code Generation, Part II

Parameter Passing Semantics: Value vs. Reference

- So far, our examples have dealt only with *value parameters*, which are the only kind found in C, Java, and Python

  Ignorant comments from numerous textbook authors, bloggers, and slovenly hackers notwithstanding [End Rant].

- Pushing a parameter's value on the stack creates a copy that essentially acts as a local variable of the called function.
- C++ (and Pascal) have *reference parameters*, where assignments to the formal are assignments to the actual.

```
void incr(int& x) {                y = 4;
    x += 1;                        incr(y); // Now y == 5.
}
```

- The distinction is clear from this fact in Java:

```
var temp = x;    /* x a local variable. */
f(x);
/* At this point, x == temp, regardless of the body of f. */
```

Implementation of Reference Parameters

- Implementation of reference parameters is simple:
  - Push the address of the argument, not its value, and
  - To fetch from or store to the parameter, do an extra indirection.

```
void incr(int& x) {
    x += 1;
}

incr:
   # Prologue goes here
   lw t0, 0(fp)
   lw t1, 0(t0)
   addi t1, t1, 1
   sw t1, 0(t0)
   # Epilogue goes here
```

```
y = 4;
incr(y);

# Assume y at -12(fp)
li t0, 4
sw t0, -12(fp)
addi t0, fp, -12   # &y
addi sp, sp -4
sw t0, 0(sp)
jal incr
addi sp, sp, 4
```

Copy-in, Copy-out Parameters

- Some languages, such as Fortran and Ada, have a variation on this: *copy-in, copy-out*. As for call by value, the value of the parameter is *copied into* the parameter, but also the final value of the parameter is *copied out* to the original location of the actual parameter after function returns.
  - "Original location" because of cases like `f(A[k])`, where `k` might change during execution of `f`. In that case, we want the final value of the parameter copied back to `A[k0]`, where `k0` is the original value of `k` before the call.
  - Question: can you give an example where call by reference and copy-in, copy-out give different results?

Implementation of Copy-in/Copy-out Parameters

- We can implement copy-in/copy-out as a variation of the by-reference implementation.

```
void incr(int& x) {
    x += 1; etc.
}
```

```
y = 4;
incr(y);
```

```
incr:
   # Prologue goes here.
   # Allocate local at -12(fp) for x
   lw t0, 0(fp)
   lw t0, 0(t0)
   sw t0, -12(fp)  # Copy in
   lw t0, -12(fp)
   addi t0, t0, 1
   sw t0, -12(fp)
   # etc.  (modify -12(fp) only)
   lw t0, 0(fp)
   lw t1, -12(fp)
   sw t1, 0(t0)    # Copy out
   # Epilogue goes here
```

```
# Assume y at -12(fp)
li t0, 4
sw t0, -12(fp)
addi t0, fp, -12   # &y
addi sp, sp, -4
sw t0, 0(sp)
jal incr
addi sp, sp, 4
```

Parameter Passing Semantics: Call by Name

- Algol 60's definition says that the effect of a call $P(E)$ is as if the body of P were substituted for the call (dynamically, so that recursion works) and E were substituted for the corresponding formal parameter in the body (changing names to avoid clashes).
- It's a simple description that, for simple cases, is just like call by reference:

```
procedure F(x)
   integer x;
begin
   x := 42;
end F;
```

```
    F(aVar);
```

becomes

```
    aVar := 42;
```

- But the (unintended?) consequences were "interesting".

Call By Name: Jensen's Device

- Consider:

```
procedure DoIt (i, L, U, x, x0, E)
    integer i, L, U; real x, x0, E;
begin
    x := x0;
    for i := L step 1 until U do
        x := E;
end DoIt;
```

- To set y to the sum of the values in array A[1:N],

```
integer k;
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

- To set z to the Nth harmonic number:

```
DoIt(k, 1, N, z, 0.0, z+1.0/k);
```

- Now how are we going to make this work?

Call By Name: Implementation

- Basic idea: Convert call-by-name parameters into parameterless functions (traditionally called *thunks*.)
- To allow assignment, these functions can return the addresses of their results.
- So the call

```
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

  becomes something like (please pardon highly illegal notation):

```
integer t1;  real t2, t3, t4;
t2 := 1.0; t3 := 0.0;
DoIt(lambda: &k, lambda: &t2, lambda: &N, lambda: &y,
     lambda: &t3, lambda: (t4 := y+A[k], &t4));
```

- Later languages have abandoned this particular parameter-passing mode.

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:

t_0 = cgen(&A[E], t_0):
  t_1 = cgen(&A)
  t_2 = cgen(E)
  $\Rightarrow$ t_3 := t_2 * 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 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 stored in row-major order.
- Where does this come from? Assuming `S`, again, is the size of an individual element, the size of a row of N elements will be $S \cdot N$.

IL for $M \times N$ 2D array

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

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 a pointer to 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:

```
t = cgen(&e1[e2,e3]):
    t1 = cgen(e1);      # Yields a pointer to a descriptor.
    t2 = cgen(e2;
    t3 = cgen(e3)
    ⇒ 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 × (e2-1) + S2 × (e3-1)
```

- But some algebra puts this into the form

```
V0' + S1 × e2 + S2 × e3
```

  where

  `V0' = &e[1,1] - S1 - S2 = &e[0,0]` *(if it existed).*

- So with the descriptor

V0'	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 }
```

Caveats

- Unfortunately, TANSTAAFL (There Ain't No Such Thing As A Free Lunch):
- Use of descriptors is nifty, but it costs:
  - For 1-D arrays, multiplication by a stride can be somewhat faster if the stride is known and is a power of 2 than when the stride is unknown due to differencec in cost of multiplication vs. shift.
  - Fetching the VO from memory can also cost cycles relative to computing address of array on the stack or in static memory.
  - And fetching strides from memory is more expensive than using immediates.
  - Also, when stride is unknown can be hard to use vectorizing operations.