COS320: Compiling Techniques

Zak Kincaid

March 9, 2019
Compiling data types
Structures

```c
struct Point { long x; long y; };

struct Rect { struct Point tl, br; };

struct Rect mk_square(struct Point top_left, long len) {
    struct Rect square;
    square.tl = top_left;
    square.br.x = top_left.x + len;
    square.br.y = top_left.y - len;
    return square;
}
```

*How do we compile these structures?*
struct Rect mk_square(struct Point top_left, long len)

- cdecl convention:
  - Parameter 1 in rdi
  - Parameter 2 in rsi
  - Return in rax

Parameter 1 doesn't fit into rdi, and return doesn't fit into rax.

Straightforward solution: pass & return pointers to values that don't fit into registers (Java, OCaml)

C has copy-in/copy out semantics ("call by value")

If we call mk_square(p, 5) and mk_square writes to top_left.x, the value of p.x does not change from the perspective of the caller.
struct Rect mk_square(struct Point top_left, long len)

- cdecl convention:
  - Parameter 1 in rdi
  - Parameter 2 in rsi
  - Return in rax

- Parameter 1 doesn't fit into rdi, and return doesn't fit into rax
struct Rect mk_square(struct Point top_left, long len)

- cdecl convention:
  - Parameter 1 in rdi
  - Parameter 2 in rsi
  - Return in rax

- Parameter 1 doesn’t fit into rdi, and return doesn’t fit into rax
- Straightforward solution: pass & return pointers to values that don’t fit into registers (Java, OCaml)
- C has copy-in/copy out semantics (“call by value”)
  - If we call mk_square(p, 5) and mk_square writes to top_left.x, the value of p.x does not change from the perspective of the caller
Copy-in/Copy-out

- Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```
Copy-in/Copy-out

• Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```

• Solution for return:

```c
struct Rect* mk_square(long top_left_x, long top_right_x, long len) {
    struct Rect result;
    ...
    return &result;
}
```
Copy-in/Copy-out

- Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```

- Solution for return:

```c
struct Rect* mk_square(long top_left_x, long top_right_x, long len) {
    struct Rect result;
    ...
    return &result;
}
```

- Unsafe!
Copy-in/Copy-out

• Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```

• Solution for return:

```c
struct Rect* mk_square(long top_left_x, long top_right_x, long len) {
    struct Rect *result = malloc(sizeof(struct Rect));
    ...
    return result;
}
```
Copy-in/Copy-out

- Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```

- Solution for return:

```c
struct Rect* mk_square(long top_left_x, long top_right_x, long len) {
    struct Rect *result = malloc(sizeof(struct Rect));
    ...
    return result;
}
```

- Protocol: caller must de-allocate space
- Heap allocation is slow
Copy-in/Copy-out

- Solution: use additional parameters for structs

```c
struct Rect mk_square(long top_left_x, long top_right_y, long len)
```

- Solution for return:

```c
void mk_square(struct Rect *result,
               long top_left_x, long top_right_x, long len) {
    ...
    return;
}
```

- Callee is responsible for allocating space for return value
Copy-in/Copy-out

- Solution: use additional parameters for structs

  ```c
  struct Rect mk_square(long top_left_x, long top_right_y, long len)
  ```

- Solution for return:

  ```c
  void mk_square(struct Rect *result,
                 long top_left_x, long top_right_x, long len) {
    ...
    return;
  }
  ```

- Callee is responsible for allocating space for return value
Structures in memory

• What *is* a pointer to a structure?
Structures in memory

- **What is a pointer to a structure?**

  - Address of the start of a block of memory large enough to store the struct

```c
struct Point { long x, y; };
struct Point* p = malloc(sizeof(struct Point));
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>
```

- **Nested structs:**

```c
struct Rect { struct Point tl, br; };
struct Rect* r = malloc(sizeof(struct Rect));
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tl.x</td>
<td>tl.y</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>br.x</td>
<td>br.y</td>
</tr>
</tbody>
</table>
```

- **Compiler needs to know:**
  - Size of the struct so that it can allocate storage
  - Shape of the struct so that it can index into the structure
Structures in memory

• What is a pointer to a structure?
  • Address of the start of a block of memory large enough to store the struct
  • Nested structs: `struct Rect { struct Point tl, br; };`  
    `struct Rect* r = malloc(sizeof(struct Rect));`

```
<table>
<thead>
<tr>
<th>tl.x</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl.y</td>
<td></td>
</tr>
<tr>
<td>br.x</td>
<td></td>
</tr>
<tr>
<td>br.y</td>
<td></td>
</tr>
</tbody>
</table>
```
Structures in memory

• What is a pointer to a structure?
  • Address of the start of a block of memory large enough to store the struct
  • Nested structs: struct Rect { struct Point tl, br; }
    struct Rect* r = malloc(sizeof(struct Rect));

  \begin{center}
  \begin{tabular}{c c c c}
    \hline
    tl.x & tl.y & br.x & br.y \\
    \hline
  \end{tabular}
  \end{center}

• Compiler needs to know:
  • Size of the struct so that it can allocate storage
  • Shape of the struct so that it can index into the structure
Padding & Alignment

• Memory accesses need to be aligned
  • E.g., in x86lite, memory addresses are divisible by 8
  • Need to insert padding: unused space so that pointers align with addressable boundaries

• How do we lay out storage?

```
struct Example {
    int x;
    char a;
    char b;
    int y;
};
```
Structures in LLVM

%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

define void @mk_square(%Rect* noalias sret %result, i64 %top_left_x, i64 %top_left_y, i64 %len) {
    %square = alloc %Rect
    ; %square.tl = top_left
    %square_t1_x = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 0
    %square_t1_y = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 1
    store i64 %top_left_x, i64* %square_t1_x
    store i64 %top_left_y, i64* %square_t1_y

    ; %square.br.x = top_left + len
    %square_br_x = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 0
    %t1 = add i64 %top_left_x, %len
    store i64 %t1, i64* %square_br_x

    ; %square.br.y = top_left - len
    %square_br_y = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 1
    %t2 = sub i64 %top_left_y, %len
    store i64 %t2, i64* %square_br_y

    ; return square
    %result_t1_x = getelementptr %Rect, %Rect* %result, i32 0, i32 0, i32 0
    %result_t1_y = getelementptr %Rect, %Rect* %result, i32 0, i32 0, i32 1 ...
    %t3 = load i64, i64* %square_t1_x
    %t4 = load i64, i64* %square_t1_y ...
    store i64 %t3, i64* %result_t1_x
    store i64 %t4, i64* %result_t1_y ...
    ret void}
getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types.
  - Given a type, a pointer `p` of that type, and a `path` `q` consisting of a sequence of indices, `getelementpointer` computes the address of `p->q`.
- Does *not* access memory (like x86 `lea`).

%Point = type { i64, i64 }
%Rect = type { %Point, %Point }
The `getelementpointer` instruction handles indexing into tuple, array, and pointer types.

- Given a type, a pointer `p` of that type, and a path `q` consisting of a sequence of indices, `getelementpointer` computes the address of `p->q`.

- Does not access memory (like x86 `lea`).

```assembly
%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

%square_tl_x = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 0
&(%square[0])
&(%square[0].tl)
&(%square[0].tl.x)

computes %square + 0*sizeof(struct Rect) + 0 + 0
```
getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types.
  - Given a type, a pointer \( p \) of that type, and a path \( q \) consisting of a sequence of indices, `getelementpointer` computes the address of \( p->q \).

- Does not access memory (like x86 `lea`)

```assembly
%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

%square_tl_y = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 1
  &(%square[0])
    &(%square[0].tl)
      &(%square[0].tl.y)

computes %square + 0*sizeof(struct Rect) + 0 + sizeof(i64)
```
The `getelementpointer` instruction handles indexing into tuple, array, and pointer types.

- Given a type, a pointer `p` of that type, and a path `q` consisting of a sequence of indices, `getelementpointer` computes the address of `p->q`.

- It does not access memory (like x86 `lea`).

```mlir
%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

%square_br_y = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 1
&(%square[0])
&(%square[0].br)
&(%square[0].br.y)

computes %square + 0*sizeof(struct Rect) + sizeof(struct Point) + sizeof(i64)
```
The `getelementpointer` instruction handles indexing into tuple, array, and pointer types.

- Given a type, a pointer \( p \) of that type, and a path \( q \) consisting of a sequence of indices, `getelementpointer` computes the address of \( p->q \).

This instruction does **not** access memory (like x86 `lea`).

```c
%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

%square6_br_y = getelementptr %Rect, %Rect* %square, i32 6, i32 1, i32 1
	&(%square[0])
	&(%square[6].tl)
	&(%square[0].tl.y)
```

This computes \%square + 6*sizeof(struct Rect) + sizeof(struct Point) + sizeof(i64).
Arrays
Single-dimensional arrays

- In C: essentially the same as tuples
  - Array is stored as a contiguous chunk of memory
  - Index into position of \( i \) of an array \( a \) of \( ts \) with \( a + \text{sizeof}(t) \times i \)
Single-dimensional arrays

• In C: essentially the same as tuples
  • Array is stored as a contiguous chunk of memory
  • Index into position of $i$ of an array $a$ of $t$s with $a + \text{sizeof}(t)\times i$

• Memory-safe languages (e.g., OCaml & Java) must check that an array access is within bounds before accessing
  • Compiler must generate array access checking code
  • Store array length before array contents, or in a pair
    type bytes = char array \rightarrow \%\text{bytes} = \text{type} \{ \text{i}64, [0 \times \text{i}8] \}*
    or \%\text{bytes} = \text{type} \{ \text{i}64, \text{i}8* \}*

Example: suppose we want to load $a[i]$ into %rax; suppose %rbx holds a pointer to $a$ and %rcx holds an index.

```assembly
movq (%rbx) %rdx // load size into rdx
cmpq %rdx %rcx // compare index to bound
jl __ok // jump if $i < a$.size
callq __err_oob // test failed, call the error handler
__ok :
movq 8(%rbx, %rcx, 8) %rax // load $a[i]$
```
Single-dimensional arrays

- In C: essentially the same as tuples
  - Array is stored as a contiguous chunk of memory
  - Index into position of i of an array a of ts with a + sizeof(t)*i
- Memory-safe languages (e.g, OCaml & Java) must check that an array access is within bounds before accessing
  - Compiler must generate array access checking code
  - Store array length before array contents, or in a pair
    type bytes = char array → %bytes = type { i64, [0 x i8] }*
    or %bytes = type { i64, i8* }*
  - Example: suppose we want to load a[i] into %rax; suppose %rbx holds a pointer to a and %rcx holds an index.

```assembly
movq (%rbx) %rdx      // load size into rdx
cmpq %rdx %rcx        // compare index to bound
j l __ok              // jump if i < a.size
callq __err_oob       // test failed, call the error handler
__ok:
movq 8(%rbx, %rcx, 8) %rax  // load a[i]
```
Multi-dimensional arrays

• In C: row-major order
  • 3x2 array: m[0][0], m[0][1], m[1][0], m[1][1], m[2][0], m[2][1]

• In Fortran: column-major order
  • 3x2 array: m[0][0], m[1][0], m[2][0], m[0][1], m[1][1], m[2][1]

• In OCaml & Java: no multi-dimensional arrays
  • 2-dimensional array is an array of arrays
    type mat = int array array → ℓmat = type { i64, { i64, i64* }* }
Strings

- Null-terminated arrays of characters
- String constants are kept in the text segment
  - LLVM: @str = constant [18 x i8] c"Factorial is %ld\0A\00"
  - X86: str: .string "Factorial is %d\n"
  - In the text segment ⇒ immutable
Variant types
Enumerations

- type color = Red | Green | Blue \(\rightarrow\) i8
  - Red \(\rightarrow\) 0
  - Green \(\rightarrow\) 1
  - Blue \(\rightarrow\) 2
Enumerations

- type color = Red | Green | Blue → i8
  - Red → 0
  - Green → 1
  - Blue → 2
- Compiling switch:
  1. Nested if statements
  2. Jump tables (for dense switches):

```
switch (color) {
  case Red:
    ...
  case Green:
    ...
  case Blue:
    ...
}
```

```
# color in %rax
jmp (table, %rax, 8)
LabelRed:
  ...
LabelGreen:
  ...
LabelBlue:
  ...
```

```
table:
  .quad LabelRed, LabelGreen, LabelBlue
```
Algebraic data types

- Algebraic data types hold data, and can pattern match on constructor
- type expr = Add of expr * expr | Var of string
  - Easy way: quadword tag + payload. Must store a pointer if more space is needed.
    - type %expr = { i64, i64* }
    - (use bitcast to convert i64* pointer to { %expr*, %expr* }* or { i64, [0 x i8] }* after pattern matching)
  - More complicated way: tack a quadword tag in front of payload
Algebraic data types

- Algebraic data types hold data, and can pattern match on constructor
- type expr = Add of expr * expr | Var of string
  - Easy way: quadword tag + payload. Must store a pointer if more space is needed.
    - type %expr = { i64, i64* }
    - (use bitcast to convert i64* pointer to { %expr*, %expr* }* or { i64, [0 x i8] }* after pattern matching)
  - More complicated way: tack a quadword tag in front of payload
- Nested pattern matching → unnested pattern matching at AST level