15-411: Compiler Design Spring 2026

Recitation 7: Memory & Dynamic Semantics Solutions 12 March

`X86_64` Operand Sizes

In previous assignments, we have only worked with 32-bit types. Now, you'll need to modify your compiler to support 64-bit pointers. Most of the `x86-64` instructions specify the sizes of their operands (e.g. `movl` vs. `movq`), so you'll need to take operand sizes into account when generating assembly.

The assembler will gladly let you use 32-bit instructions with your 64-bit data, but be careful: most 32-bit instructions will **zero the upper 32 bits of your registers**. Additionally, you need to be careful when reading or writing to memory. You can either allocate 8 bytes for every temp you spill, or worry about a non-linear mapping from temp number to memory offset.

Compiling the `==` and `!=` operations might get more tricky because they can be used with integers, boolean values, *and* pointers.

Structs and Alignment: a 213 Review

Structs store a collection of fields. In Lab 4, our fields will either be primitive types, arrays, or other structs. Each primitive field of a struct must be aligned by the size of its type[1], and each struct/array field must be aligned to the size of its largest field. These rules can introduce padding if a larger field follows a smaller one. The size of the overall struct needs to be aligned with its largest field.

Most of the time, L4 programs will reference fields via the arrow notation (`->`), as we cannot place structs on the stack or manipulate a struct directly. That said, we do need the dot notation (`.`) for arrays of structs and expressions of the form `(*S).f`.

Checkpoint 0

Compute the byte offsets for each of the fields in the two struct definitions below. Assume that we are *not* padding `int` and `bool` fields unless it is required by the alignment rules.

```
struct A {
   int* b;
   int c;
   struct A* d;
};

struct B {
   int i;
   struct A a;
   int k;
};

struct C {
  int x;
  int y;
  int z;
};

struct D {
  struct C c;
  int w;
};
```

---

[1] To satisfy this requirement, one simplification you may want to make (to start with) is adding padding to `int` and `bool` fields to make them 8 bytes

Solution:

```
struct A (size: 24):
  b: 0    c: 8    d: 16
struct B (size: 40):
  i: 0    a: 8    k: 32
struct C (size: 12):
  x: 0    y: 4    z: 8
struct D (size: 16):
  c: 0    z: 12
```

Checkpoint 1

In Lab 4, struct accesses can get messy. You and your partner should carefully discuss your design. To make your logic cleaner, here are some tips:

- There are two methods for accessing structs (`.` and `->`). Can you design your elaboration such that there's only one way to access structs?
- For struct and array accesses, split it into first computing the address and then reading/writing to the address. Compute the address of the struct field and store it in a temp. Along the way, you might need to generate intermediate temps. This will be discussed more in the next section.

Compute the addresses (relative to `x` and `b`) for each struct access in the following program. What information will we need to know about the structs for code generation? What about for typechecking?

```
int foo(struct B* x) {
   x->a.d = alloc(struct A);
   return x->a.d->c;
}

int main() {
   struct B* b = alloc(struct B);
   b->a.c += 15411;
   return foo(b);
}
```

Solution:

```
&(x->a.d) = x + 8 + 16
&(x->a.d->c) = &((*(&(x->a.d)))->c) = *(&(x->a.d)) + 8
&(b->a.c) = b + 8 + 8
```

For typechecking, we'll need to know the type of each field of the struct. For code generation, we'll only need to know the size. Depending on your design, two C0 types could be compiled as the same size (for example, you could decide to use four bytes to store booleans).

`sizeof` in L4

During code generation, you will need to know the sizes of the types your program uses. For pointers, this is pretty easy; for arrays, this is only slightly harder. Figuring out offsets for structs, however, can be non-trivial. You will want to write a recursive function for computing the sizes of types, since structs may be nested. With these two parts, you'll be able to compute struct offsets needed in code generation.

```
fun sizeof ty =
  case ty of
    Int => 4
  | Bool => 4
  | Ptr _ => 8
  | Struct id => ???
```

Addressing Schemes

There are many ways to describe memory locations in x86 assembly:

Form	Address
`(%rsp)`	`%rsp`
`a(%rsp)`	`%rsp + a`
`(%rsp, %rax)`	`%rsp + %rax`
`a(%rsp, %rax)`	`%rsp + %rax + a`
`a(%rsp,%rax,s)`	`%rsp + %rax*s + a`

In the table above, `%rsp` and `%rax` represent registers (note that you *aren't* required to use `%rax`), a is an arbitrary constant, and s can be 1, 2, 4, or 8. The last form is useful for array accesses, and the form `a(%rsp)` is useful for accessing fields of a struct. There are other forms, but they probably aren't necessary to know to implement your compiler.

Also, recall that when you use one of the above forms, the `mov` instruction dereferences the memory location while the `lea` instruction loads the address of the memory location. Both of these instructions will be useful in Lab 4.

Code Generation for Structs and Arrays

Once we have some sort of environment that maps struct fields to offsets, we can go about with actual code generation. From our example, alloc is straight forward, so let's focus on line 19.

We do not want to elaborate `b->a.c += 15411` to `b->a.c = b->a.c + 15411`, as in general, the left hand side may have side effects we cannot repeat[2]. Instead, we must:

(a) Compute the *address of* `b->a.c` (called `addr`).
(b) Elaborate the right hand side to include `*addr`.
(c) Compute the right side of the assignment (in this case, `*addr + 15411`).
(d) Check that the address is not `NULL`[3].
(e) Set the memory at the address to the result.

Note that the semantics for arrays are slightly different than this. Make sure to double check the lecture notes for specifics.

Checkpoint 2

Assuming `b` is in the register t_0, write assembly pseudocode that executes the statement `b->a.c += 15411`. Remember that C0 is a memory-safe language, so your executable should *never* dereference a `NULL` pointer. You'll need to manually insert `NULL` checks (the C standard library's `raise` function will be of interest). For Lab 4, we expect you to raise `SIGUSR2` when a C0 program dereferences a `NULL` pointer.

---

[2] Importantly, the **15-122** reference compiler does not correctly implement these semantics. The **15-411** reference compiler **does** (Thanks Iván!)

[3] Yes, this is in theory redundant. Though, remember to check if the original struct pointer is `NULL`!

Solution:

```
  lea 8(t0), t1  // &(b->a)
  lea 8(t1), t1  // &(b->a.c) [could also combine these two offsets]
  t2 <- 15411
  if t0 == 0 then goto L1 else L2 // null-check on original struct pointer
L1:
  // C standard library functions can be directly called from assembly
  raise(SIGUSR2)
L2:
  mov (t1), t3
  t3 <- t3 + t2
  mov t3, (t1)
```

Dynamic Semantics for Mutable Memory

Having memory beyond the stack, which can carry between functions means that our dynamic semantics needs to be extended. We can do this but adding a new variable, H, to the context of our rules. This represents an infinite heap of memory that our program can use.

All the rules that we've currently discussed don't modify the heap, so our heap remains the same there. So, we introduce some new rules for the constructs added in L4 that do modify the heap:

$$
\begin{array}{llll}
H; S; \eta \vdash \texttt{null} \rhd K & \to & H; S; \eta \vdash 0 \rhd K & \\
H; S; \eta \vdash \texttt{alloc}(\tau) \rhd K & \to & H[a \mapsto \texttt{default}(\tau), \mapsto a + |\tau|]; S; \eta \vdash a \rhd K & \\
 & & a = H() & \\
H; S; \eta \vdash *e \rhd K & \to & H; S; \eta \vdash e \rhd (*\_, K) & \\
H; S; \eta \vdash a \rhd (*\_, K) & \to & H; S; \eta \vdash H(a) \rhd K & (a \neq 0) \\
H; S; \eta \vdash a \rhd (*\_, K) & \to & \texttt{exception}(\texttt{mem}) & (a = 0) \\
H; S; \eta \vdash \texttt{assign}(*d, e) \blacktriangleright K & \to & H; S; \eta \vdash d \rhd (\texttt{assign}(*\_, e), K) & \\
H; S; \eta \vdash a \rhd (\texttt{assign}(*\_, e), K) & \to & H; S; \eta \vdash e \rhd (\texttt{assign}(*a, \_), K) & \\
H; S; \eta \vdash c \rhd (\texttt{assign}(*a, \_), K) & \to & H[a \mapsto c]; S; \eta \vdash \texttt{nop} \blacktriangleright K \qquad (a \neq 0) & \\
H; S; \eta \vdash c \rhd (\texttt{assign}(*a, \_), K) & \to & \texttt{exception}(\texttt{mem}) \qquad (a = 0) &
\end{array}
$$

Importantly, `default` is just the default value of τ. More rules are in the lecture notes.

Checkpoint 3

Write a trace of the dynamic semantics for the following program:

```
int *p = NULL;
*p = 1 / 0;
```

Solution:

$$\begin{array}{ll}
H; \cdot; \cdot \vdash \mathsf{declare}(\mathsf{p}, \mathsf{int}{*}, \mathsf{seq}(\mathsf{assign}(\mathsf{p}, \mathtt{NULL}), \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0)))) \blacktriangleright \cdot & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto \mathsf{nothing}] \vdash \mathsf{seq}(\mathsf{assign}(\mathsf{p}, \mathtt{NULL}), \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0))) \blacktriangleright \cdot & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto \mathsf{nothing}] \vdash \mathsf{assign}(\mathsf{p}, \mathtt{NULL}) \blacktriangleright \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0)) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto \mathsf{nothing}] \vdash \mathtt{NULL} \rhd (\mathsf{assign}(p, \_), \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0))) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto \mathsf{nothing}] \vdash 0 \rhd (\mathsf{assign}(p, \_), \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0))) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash \mathsf{nop} \rhd \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0)) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash \mathsf{assign}({*}\mathsf{p}, \mathtt{binop}(1, \mathtt{div}, 0)) \blacktriangleright \cdot & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash \mathsf{p} \rhd \mathsf{assign}({*}\_, \mathtt{binop}(1, \mathtt{div}, 0)) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash 0 \rhd \mathsf{assign}({*}\_, \mathtt{binop}(1, \mathtt{div}, 0)) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash \mathtt{binop}(1, \mathtt{div}, 0) \rhd \mathsf{assign}({*}0, \_) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash 1 \rhd \mathtt{binop}(\_, \mathtt{div}, 0), \mathsf{assign}({*}0, \_) & \to \\
H; \cdot; \eta[\mathsf{p} \mapsto 0] \vdash 0 \rhd \mathtt{binop}(1, \mathtt{div}, \_), \mathsf{assign}({*}0, \_) & \to \\
\mathtt{exception(arith)} &
\end{array}$$