15-411: Compiler Design — Spring 2024

Recitation 7: Memory Solutions — 15 March

X86_64 Operand Sizes

In previous assignments, we have only worked with 32-bit types. Now, you will need to modify your compiler to account for 64-bit types. Most of the x86-64 instructions specify the sizes of their operands, so you will want to use the ___Q (rather than ___L) variants.

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.

The == and != operations are particularly tricky in Lab 4 because they can be used with integers, Boolean values, and pointers. You'll notice that because pointers are 64 bits wide, you'll need to take the operand sizes into account when generating code for these operations.

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.

```
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;
};
```

Solution:

[1] One simplification you may want to make (to start with) is to add 4 bytes of padding to int's and bool's

```
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

Compute the byte offsets for each struct access in the following program. What information do we need about the structs for typechecking? For code generation?

```
12 int foo(struct B* x) {
13    x->a.d = alloc(struct A);
14    return x->a.d->c;
15 }
16
17 int main() {
18    struct B* b = alloc(struct B);
19    b->a.c += 15411;
20    return foo(b);
21 }
```

Solution:

```
[l.13 lhs] &(x->a.d)    x+8+16
[l.14]     x->a.d->c    *(*(x+8+16)+8)
[l.19 rhs] b->a.c       *(b+8+8)
[l.19 lhs] &(b->a.c)    b+8+8
```

`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, 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 righ hand side to include `*addr`.

(c) Compute the right side of the assignment (in this case, `*addr + 15411`. To use the value `*addr`, we must first complete the null checks on any memory accesses needed to get to that value. Here, `b` needs to be dereferenced to find the value stored at the address `addr`.).

(d) Check that the address (`addr`) 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 that executes the statement `b->a.c += 15411`.

Solution: (this is admittedly a weird amalgamation of abstract and concrete assembly, but I think it is useful to see it this way)

```
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
L1:
raise(SIGUSR2)
L2:
mov (t1), t3
t3 <- t3 + t2
if t1 == 0 then goto L3 else goto L4
L3:
raise(SIGUSR2)
L4:
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.

[2] Importantly, the reference compiler does not correctly implement these semantics. Your compiler should follow the dynamic semantics directly, not the reference's buggy implementation.

[3] Yes, this is in theory redundant

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:

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

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:

$$
\begin{array}{ll}
H;\cdot;\cdot \vdash \texttt{declare}(\texttt{p}, \texttt{int*}, \texttt{NULL}, \texttt{assign}(\texttt{*p}, \texttt{binop}(1, \texttt{div}, 0))) \blacktriangleright \cdot & \rightarrow \\
H;\cdot;\cdot \vdash \texttt{NULL} \lhd \texttt{declare}(\texttt{p}, \texttt{int*}, \_, \texttt{assign}(\texttt{*p}, \texttt{binop}(1, \texttt{div}, 0))) & \rightarrow \\
H;\cdot;\cdot \vdash 0 \lhd \texttt{declare}(\texttt{p}, \texttt{int*}, \_, \texttt{assign}(\texttt{*p}, \texttt{binop}(1, \texttt{div}, 0))) & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash \texttt{assign}(\texttt{*p}, \texttt{binop}(1, \texttt{div}, 0)) \blacktriangleright \cdot & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash \texttt{p} \lhd \texttt{assign}(*\_, \texttt{binop}(1, \texttt{div}, 0)) & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash 0 \lhd \texttt{assign}(*\_, \texttt{binop}(1, \texttt{div}, 0)) & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash \texttt{binop}(1, \texttt{div}, 0) \lhd \texttt{assign}(*0, \_) & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash 1 \lhd \texttt{binop}(\_, \texttt{div}, 0), \texttt{assign}(*0, \_) & \rightarrow \\
H;\cdot;[\texttt{p} \mapsto 0] \vdash 0 \lhd \texttt{binop}(1, \texttt{div}, \_), \texttt{assign}(*0, \_) & \rightarrow \\
\texttt{exception(arith)} &
\end{array}
$$