15-122: Principles of Imperative Computation

Recitation 23 — Josh Zimmerman

Demystifying the C0 VM – the operand stack

In claclab, we had a stack of integer values that we manipulated. We have a similar stack for the C0 VM, but it is much more advanced. For the VM, we take advantage of the `void *` type in C to store values of any type on the stack.

We've seen how we can use `void *` to support functions that can take arguments that are pointers to a variety of types, but we can also cast things that aren't even pointers to `void *`. This is useful for the VM, since we can just cast the values we want to store on the stack to `void *`.

Casting values to `void *` allows us to store any kind of value on the stack, meaning that we actually only need to have a single stack that can store all of our variables!

Note: from here on, I'll refer to `void *` as `c0_value`, since that's how we refer to it in the code and it helps us remember what the actual meaning of the type is.

There are two inline functions we've given you that are important here: `INT` and `VAL`. The `VAL` function takes a 32-bit signed `int` and converts it to a `c0_value`, and the `INT` function takes in a `c0_value` and converts it to a 32-bit signed `int`. It's very very important to note that the argument to `INT` must come from a call to `VAL`. Otherwise, the result is unpredictable.

For any signed 32-bit integer `x`, we guarantee that `INT(VAL(x)) == x`.

Other casts for the stack (of pointers to `c0_value` are much more straightforward).

Demystifying the C0 VM – the program counter

For the C0 VM, we keep an array of bytecode to execute. We use an index into this array to keep track of the current instruction we're executing. Every time we want to execute an instruction we look at `P[pc]`, where `P` is the array that contains the bytecode and `pc` is the program counter. The number at `P[pc]` tells us which instruction to execute, and there may be arguments to the instruction that come after the instruction itself.

When we're done executing a particular instruction, we change the `pc` so that it points to the next instruction we want to execute. How exactly we do this change depends on the instruction. For some instructions, we simply increment `pc` past the arguments to the instruction we just executed, but for others we need to do more complex operations. For other instructions (the control flow operations) we change `pc` depending on the arguments to the instruction.

This can be very tricky, since we need to support changing `pc` by a negative amount, but the arguments we get are two *unsigned* bytes. An challenging part of the VM is figuring out how to get a signed offset from these two unsigned bytes.

Demystifying the C0 VM – the call stack

In C0 , we keep track of function calls with a *call stack*.

Each function has its own frame, which is defined as:

```
1 struct frame {
2  c0_value *V; /* local variables */
3  stack S;     /* operand stack */
```

```
4   ubyte *P;    /* function body */
5   size_t pc;   /* return address */
6 };
```

Each time we call a function, we need to save its frame onto the call stack so that when the function we're calling returns we can restore the state of the current function.

If you think back to clac, this is somewhat like the stack of queues we used when calling functions, except that for the VM, we need to store more information than just the function definition — we also have a separate array of local variables and a separate operand stack for each function.

Also, since we're keeping the instructions in an array rather than a queue, we need to keep track of the instruction we should execute after a function returns.

Note that we keep instructions in an array to allow us to freely jump around from an instruction to an earlier instruction (this is useful in, for example, a loop).

Testing

While working on the VM, it's important to remember the ideas of testing that we've used throughout the semester. In particular, it's important to remember to break the code you write up into small, byte-sized chunks that you can easily test.

For example, the control flow instructions all use the operation of combining two unsigned bytes and getting back a signed 16-bit (two byte) int.

Since that code involves casting (which is hard to think about) and is present in many different cases, it makes sense to have a helper function to do this for us.

Having helper functions is also great since it helps us write test cases — we can write test cases in C to specifically test just that function. We can also write C0 programs and compile them to test, but that makes it much harder to get useful information about what a problem is: If a C0 program that you run returns an incorrect result, that doesn't tell you much about what the bug is.

However, if you're testing one very specific function and it fails, it's much easier to figure out what's wrong since you've narrowed it down already.

Just like what we did with tests throughout the semester, testing lots of different kinds of cases is very important. Here, some important cases are a jump that's 0, one that's small and positive, one that's large and positive, one that's small and negative, and so on.

```
1 int16_t pack(ubyte o1, ubyte o2) {
2   // ...
3 }
4 int main() {
5   size_t start = 1000000;
6   printf("No jump: %zu\n", start + pack(0x00, 0x00));
7   printf("Small positive jump: %zu\n", start + pack(0x00, 0x01));
8   printf("Large positive jump: %zu\n", start + pack(0x01, 0xFF));
9   printf("Huge positive jump: %zu\n", start + pack(0x7F, 0xFF));
10  printf("Small negative jump: %zu\n", start + pack(0xFF, 0xF1));
11  printf("Large negative jump: %zu\n", start + pack(0xFE, 0x00));
12  printf("Huge negative jump: %zu\n", start + pack(0x80, 0x00));
13 }
```

Print statements also remain useful, and we've already put a basic one in `c0vm.c` that you may find useful and may wish to extend.