Function Arguments, Local Variables

Lecture #13

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

Previously, we updated how we handle leftvalues so that we could do dereference assignments:

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/test10
int main() {
    int a; int b; int c;
    a = b = c = 3;
    print a; print b; print c;

    int x; int *y;
    y = &x; *y = 5;
    print x;

    int n;
    n = 50;
    int **z; z = &y; **z = n;
    print x;

    // We will eventually un-comment this field
    // when we add local variables
    // int **q; q = &y;
    // print **q;
    // *q = &a;
    // *y = 7;
    // print a; // 7
}
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/test10
-----RUN-----
3
3
3
5
50
```

The Goal

We have put off functions for too long. Let's finally add support for function arguments!

```
int fred(int x) {
    print x;
    x = x + 1;
    print x;
    return 20 + x;
}

int bob(int x, int y) {
    return x * y;
}

int main() {
    // Should print 6, 6, 7, 6, 7, 42, 162
    int a; a = 6; print a;
    fred(a);
    print fred(a) + 15;
    print bob(9, 18);
}
```

The Plan

FYI: This was the most fun I had adding a feature to the compiler, primarily due to the deceptive simplicity of the changes we have to make.

1. Convert our global Symbol Table into a stack of local symbol tables (with the GST on the bottom)
2. Differentiate between local and global variables. For now we will only have 1 layer of "local-ness": variables are either local to subroutines (funtion arguments) or are globals defined in the function body. We will differentiate in `SymbolTableEntries`
3. Generate code using local variables

The reason we're doing both function args and local variables is that function args **should** be local variables. If they were globals we wouldn't need args at all.

Revisiting the Global Symbol Table

We are still going to use our existing SymbolTable code, but now we are going to use a stack of them to represent scopes. Consider:

```
int main(int argc, char* argv[]) {
    int x = argc;
    if (argc == 3) {
        char* y = argv[0];
        if (y == '/0') {
            long z = 5L;
        }
    }
}
```

The `x`, `y`, and `z` variables should be in different scopes. If we added a function outside of `main`, its variables should be in a different scope. Scoping prevents the internals of one function/block modifying the contents of another. Be careful, we still want to be able to access `x` from the inside of the second `if` statement!

The Symbol Table Stack

- Acts like a standard stack (push, pop, peek operations)
- Has accessor functions that traverse the stack from the bottom-up to find SymbolTableEntries and insert/update them in the appropriate SymbolTables
- Will replace most of our `GLOBAL_SYMBOL_TABLE` accesses (functions should still be global, and we don't un-global variables defined in function bodies yet)
- When we start parsing a function body, push a new Symbol Table to the stack. After we've **generated code** for it, pop that Symbol Table. Don't pop it after parsing, we still need to access scope data!

Function Arguments LLVM

Before we continue, we need to see what our end goal is:

```
int fred(int x, int y) {
    ...
}
```

should become

```
define i32 fred(i32 %x, i32 %y) {
    ...
}
```

(see how nice LLVM is??)

Ramifications

We actually don't need to call function args by the names programmers give them (we could use %0 and %1 in the last example) but I think it adds to readability and debug-ability. We need to do two things to support this:

1. Allow `LLVMValue.VIRTUAL_REGISTER`s to use strings as names (easy)
2. Tell our code generator that a value already exists for a given identifier, and that we don't need to do any generation (slightly harder)

Function argument values already exist

To solve this issue, I propose that:

1. When generting a function preamble, create LLVMValues for each function argument
2. Assign these LLVMValues to a new field, "`latest_llvmvalue`", in `SymbolTableEntry`
3. Any time we need GST variable data, use its `latest_llvmvalue` if it exists

Code Generation

Finally, we need to do some minor updates to our code generator everywhere we were dealing with printing registers. We replace a bunch of calls to the GST with the STS. We need to generate code for passing function arguments. And ta-da! Our example works. We even get recursion for free!

```
int recursive_fact(int x) {
    if (x <= 0) {
        return 1;
    }

    return x * recursive_fact(x - 1);
}

int main() {
    print recursive_fact(5);
}
```

One Issue

What will be wrong with the code generated for this function?

```
int iterative_fact(int x) {
    int y;
    y = x - 1;

    while (y > 0) {
        x = x * y;
        y = y - 1;
    }

    return x;
}
```

(Hint: assignment to x)

One Issue

The generated LLVM:

```
define dso_local i32 @iterative_fact(i32 %x) #0 {
    %1 = sub nsw i32 %x, 1
    store i32 %1, i32* @y
    br label %L1
    L1:
    %2 = load i32, i32* @y
    %3 = icmp sgt i32 %2, 0
    br i1 %3, label %L3, label %L2
    L3:
    %4 = load i32, i32* @y
    %5 = mul nsw i32 %x, %4
    %6 = load i32, i32* @y
    %7 = sub nsw i32 %6, 1
    store i32 %7, i32* @y
    br label %L1
    L2:
    ret i32 %5
    ret i32 0
}
```

One Issue

The `x` assignment tries to use `%x` every time! But VRs are static, so it gets the same value every time. In a future lecture, we will fix this by allocating stack space and storing into it, similar to how we store into global variables.

However, this isn't necessary! Still Turing complete if our function args are read-only, and plenty of languages have readonly function arguments (mostly functional languages though).