Compile Our Language to C/ASM

By Deniz Celik and Jacob Riedel

GOAL: Take compiler code and run on C or ASM

We wanted to compile the register code given by Riccardo in Lectures 8 and 9 to Assembly.

We thought C might be a good mid-point to compile to and gauge how much of a challenge it would be.

We would not change any of the functionality of the current code, meaning full implementation of ENV, ARGS, ADDR, and RESULT registers.

Then from C, we would go further to implement ASM, time permitting.

Our Process

- Write equivalent C commands for each code from Riccardo’s Compile
  - Make these work together for the example function provided in the lecture notes
    - (let ((sum (function s (n result) (if (zero? n) result (s (- n 1) (+ result n)))))) (sum 200000 0))
    - With Riccardo’s optimized code (V3), runs in 3.4s on Deniz’s Dell and 5.2s on my MacAir
  - Ensure answer is correct, then optimize runtime:
    - Answer must be 20000100000
    - Time will hopefully be <3.4s
  - Use compiler code V3 as a jumping off point
    - Replaced the assembler to write out to ‘.c’ file
    - Manually compile and run the C file that is automatically generated
    - Function timing based off of total compile and run time
- Cost/Benefit of ASM
  - Arguably faster to remain in C
  - Structure is already very ASM-like

Converting Python to C

Issues

1. Handling lack of type declarations
   - Result -> int or VClosureAddr
   - Env -> list of Results
   - Args -> list of Results
2. Handling lack of dynamic arrays
   - No fixed size for Env, Args
   - Reallocation can be time consuming
3. Jumping to addresses no longer uses PC

Converting Python to C

Solutions

1. Create a struct that can act as an int or VClosureAddr
2. Pre-allocate memory slightly larger than needed
3. Use goto command to jump and void* to store the labels

```
typedef struct tClosure{
        long long val;
        void* addr;
        int envLen;
        struct tClosure* p_env;
} tClosure;

tClosure env[array_size];

addr = &&PL_DONE; goto *addr;
```

Code Layout

1. Setup registers:

```
void* addr;

tClosure result;

tClosure args[array_size];
int args_index = 0;

tClosure env[array_size];
int env_index = 0;
```

2. Setup Prim-Calls:

```
long long oper_plus(tClosure clo[ ]){
    return clo[0].val+clo[1].val;}
long long oper_minus(tClosure clo[ ]){
    return clo[0].val-clo[1].val;}
long long oper_times(tClosure clo[ ]){
    return clo[0].val*clo[1].val;}
long long oper_zero(tClosure clo[ ]){
    return clo[0].val==0;}
```

3. Initialize Result and Env:

```
result.val = -1;
result.envLen = 0;
result.p_env = (tClosure*)malloc(sizeof(tClosure)*array_size);
tClosure addr1;
addr1.addr = &&PL_start0;
addr1.envLen = 0;
addr1.p_env = (tClosure*)malloc(sizeof(tClosure)*array_size);
env[0] = addr1;
env_index++
```

4. Creating Jumps:

```
PL_LOOP:
…
addr = &&PL_LOOP;
...
goto *addr;
```

5. Loading Args:

```
args[args_index] = result;
args_index++;

for (i=0;i<args_index;i++){
    env[env_index] = args[i];
    env_index++;
}
```

6. Copying to and Copying from Env:

```
addr = result.addr;
memcpy( env, result.p_env, result.envLen * sizeof(tClosure) );
env_index = result.envLen;

result.addr = addr;
memcpy( result.p_env, env, env_index * sizeof(tClosure) );
result.envLen = env_index;
```

Compilation

Compiling C code:

```
gcc -O3 -fexpensive-optimizations -fomit-frame-pointer -o auto_build.out auto_build.c
```

Optimization turned as high as possible: Compile time will take hit

Results :’(:o :| :) :D (From sadness, to happiness)

It works!

And its fast:

Running on my Mac, average runtime of .6s for sample function

HOORAY, IT WORKS! And Really quickly too!

DEMO

Questions?

Thank you.