Extension 5.5: Compiling Amy to C

Group 11 - Violeta Vicente Cantero,
Alexander Mueller, Jonas Bonnaudet

Summary

1. Theoretical background
2. Amy to C transpiler
3. Program highlights
4. Changed phase: Code Printer
5. Possible further steps
6. Little demonstration

Theoretical background

- How the code skeleton for lab 5 works
- WebAssembly modules
- How WebAssembly locals get created for each function call
- C programming

Amy to C transpiler

- Our feature allows Amy to be transpiled directly to C
- Amy is now supported on everything C supports, such as an arduino!

```
val pipeline =
  Lexer andThen
  Parser andThen
  NameAnalyzer andThen
  TypeChecker andThen
  CodeGen andThen
  CodePrinter // modified
```

Amy to C transpiler

- CodeGen.scala hasn’t changed!
- We create a stack machine just like WebAssembly
- We only have to change one file (mostly) - Module Printer

```
int stack[MAX_STACK_SIZE];
int stack_pointer = 0;
int globals[MAX_STACK_SIZE];
char memory[MAX_STACK_SIZE];
```

```
#define push stack[stack_pointer++] =
#define pop stack[--stack_pointer]
#define getGlobal(i) push globals[i]
#define setGlobal(i) globals[i] = pop
#define getLocal(i) push locals[i]
#define setLocal(i) locals[i] = pop
```

Example: Factorial.amy

```
object Factorial
  fn fact(i: Int(32)): Int(32) = {
    if (i < 2) { 1 }
    else { i * fact(i-1) }
  }

  Std.printString("5! = "  ++
     Std.intToString(fact(5)));
  Std.printString("10! = " ++
     Std.intToString(fact(10)))
end Factorial
```

```
void Factorial_fact() {
  int locals[1] = {peek (1)};drop;
  getLocal(0);
  cnst 2;
  a = pop; push (pop < a);
  if (pop){
    cnst 1;
  } else {
    getLocal(0);
    getLocal(0);
    cnst 1;
    a = pop; push (pop - a);
    Factorial_fact();
    push (pop * pop);
  }
}
```

WebAssembly backend vs C backend

```
(func $Factorial_fact (param i32) (result i32)
  local.get 0
  i32.const 2
  i32.lt_s
  if (result i32)
    i32.const 1
  else
    local.get 0
    local.get 0
    i32.const 1
    i32.sub
    call $Factorial_fact
    i32.mul
  end
)
```

→

```
void Factorial_fact() {
  int locals[1] = {peek (1)}; drop;
  getLocal(0);
  cnst 2;
  a = pop; push (pop < a);
  if (pop){
    cnst 1;
  } else {
    getLocal(0);
    getLocal(0);
    cnst 1;
    a = pop; push (pop - a);
    Factorial_fact();
    push (pop * pop);
  }
}
```

Standard functions

Strings are represented the same way!

```
void Std_printString(){
 printf("%s\n", &memory[pop]);
 cnst 0;
}

void Std_printInt(){
 printf("%d\n", pop);
 cnst 0;
}
```

Changed phases: mkInstr():ModulePrinter.scala

```
case Const(value) => s"i32.const $value"    →    case Const(value) => s"cnst $value;"

case Add => "i32.add"                       →    case Add => "push (pop + pop);"

case Div => "i32.div_s"                     →    case Div => "a = pop; push (pop / a);"
```

Changed phases: mkFun():ModulePrinter.scala

```
Stacked(
  Lined(List(resultDoc, s" ${name}() {")), // void [name of function]
    Indented(Lined( // locals
      List(
        Raw(s"int locals[${fh.args + fh.locals}] = {"),
        Raw( // put arguments in locals
          (for i <- (1 to fh.args).reverse
          yield s"peek ($i)").mkString(", ") + (if fh.args == 0 then "" else ", ")
        ), Raw( // allocate memory for used locals in function
          (for i <- 1 to fh.locals
          yield "0").mkString(", ")
        ), Raw("};"), // drop arguments from the stack that are now in locals
        Raw((for i <- 1 to fh.args
          yield "drop").mkString(";") + ";")
      )
    )),
    Indented(Stacked(mkCode(fh.code))),
  "}"
)
```

Result of mkFun()

```
void example(){
  int locals[4] = {peek(2), peek(1), 0, 0};drop;drop;
  // peek instead of pop because of the order of the arguments is reversed
  // two 0's because the function will use 2 locals

  ... // rest of the function

}
```

Changed phases: mkMod():ModulePrinter.scala

```
Stacked(
   "#include <stdio.h>",
   "#include <stdlib.h>",
   "int a;",
   "#define push stack[stack_pointer++] = ",
   "#define pop stack[--stack_pointer]",
   ...,
   Stacked(mod.functions map decFun), // declare functions
   Lined(List()), // newline
   Stacked(mod.imports map mkImport),
   Stacked(mod.functions map mkFun)
)
```

Demonstration!

Possible Further Steps

- Dynamic reallocation of memory, globals, stack, as needed
- Finish implementing all standard functions from Std.amy
- Use the tests in lab5 to verify our implementation
- Make it seamless to generate code for arduino

Thank you!