Compilation

Exercise 4, Due 5/1/2020 before 14:00

Introduction

For the first time ever we are going to compile Poseidon programs to LLVM bitcode. LLVM is an open source compiler infraructure and toolchain that supports multiple source languages (C, CPP, C#, Go, etc.) and multiple destination targets (x86, ARM, MIPS, x86_64, sparc, etc.). Its bitcode (or intermediate representation) was designed as a language-independent, high-level portable assembley. Bitcode files come in two flavours, or formats: machine readable and human readable. Machine readable (*.bc) bitcode files can be transformed into human readable ones with llvm-dis. Human readable (*.ll) bitcode files can be transformed into machine readable ones with llvm-as. In this exercise you will translate the input Poseidon program into a human readable (*.ll) bitcode file. Then, in order to check you work, the human readable bitcode file will be translated into a proper machine readable bitcode file, and linked with clang to a native executable.

Install LLVM

To complete this exercise you need a working LLVM + clang. The formal LLVM + clang version for our course can be downloaded and installed with this build-llvm-script. Note that this is a debug build, so allow at least one hour for it to finish. In addition you will need to have root privliges for the final install step of the script. To check your installation go to the source code folder and run `make everything`. You should see the prime numbers from 2 to 100 printed to stdout.

Poseidon Semantics

Until now used the term semantics to describe legal and illegal programs. But a programming language semantics also describes the way a program is meant to be executed. What is the order of evaluation when a mathematical expression is computed? What is the explicit underlying mechanism that controls execution of while loops? etc. The following sections describe the Poseidon semantics with a multitude of running examples.

If and While Statements

If statements behave similar to (practically) all programming languages: before executing their body, their condition is evaluated. If it equals 0, the body is ignored, and control is transferred to the statement immediately after the body. Otherwise, the body is executed exactly once, then control is transferred to the statement immediately after the body.

```
// Expected Output: 8 2 2
void foo(int i)
{
    if (i = 6)
    {
        PrintInt(8);
    }
    PrintInt(2);
}
void main(){ foo(6); foo(3); }
```

While statements behave similar to (practically) all programming languages: before executing their body, their condition is evaluated. If it equals 0, the body is ignored, and control is transferred to the statement immediately after the body. Otherwise, the body is executed, then the condition is evaluated again, and so forth.

```
// Expected Output: 6 7 8
void foo(int i)
{
    while (i < 9)
    {
        PrintInt(i);
        i := i + 1;
    }
}
void main(){ foo(6); foo(9); }
```

Binary Operations

Integers in Poseidon are artificially bounded between -215 and $215 - 1$. The semantics of integer binary operations in Poseidon is therefore somewhat different than that of standard programming languages. It is presented in the following table, and to distinguish Poseidon operators from the usual arithmetic signs, we shall use a Poseidon subscript inside brackets: (`+[Poseidon]`, `-[Poseidon]` etc.)

Binop	Condition	Value
a +[Poseidon] b	32767 <= a+b -32768 <= a+b < 32767 a+b < -32768	32767 a+b -32768
a -[Poseidon] b	32767 <= a-b -32768 <= a-b < 32767 a=b < -32768	32767 a-b -32768
a *[Poseidon] b	32767 <= a*b -32768 <= a*b < 32767 a*b < -32768	32767 a*b -32768
a /[Poseidon] b	32767 <= a/b -32768 <= a/b < 32767 a/b < -32768	32767 a/b -32768

Strings can be concatenated with binary operation `+`, and tested for (contents) equality with binary operator `=`. When concatenating two (null terminated) strings s1 and s2, the resulting string s1s2 is allocated on the heap, and should be null terminated. The result of testing contents equality is either `1` when they are equal, or `0` otherwise.

Order of Evaluation

When calling a function evaluation order matters. Sent parameters should be evaluated from left to right. For instance, the following code should output `32766`:

```
int x=32767;
int foo(int i,int j)
{
    PrintInt(x);
}
int inc(){ x := x + 1; return x; }
int dec(){ x := x - 1; return x; }
void main(){ foo(inc(),dec()); }
```

Global Variables should be evaluated according to their order of appearence in the original program. For example, the following code should output `32767`:

```
int x=32767;
int inc(){ x := x + 1; return x; }
int dec(){ x := x - 1; return x; }
int y := dec();
int z := inc();
void main(){ PrintInt(z); }
```

Assignments are evaluated according to a left before right paradigm. So the following code should output `6`:

```
int x := 4;
array IntArray = int[]
IntArray A := new int[9];
int inc(){ x := x + 1; return x; }
void main(){ A[inc()] := inc(); PrintInt(A[5]); }
```

Binary Expressions

Runtime Checks

Poseidon enforces three kinds of runtime checks:

- Division by zero
- Invalid pointer dereference
- Out of bounds array access.

Division by zero should be handled by printing "Division By Zero", and then exit gracefully by using an exit system call. The following code will result in such behaviour:

```
void main()
{
    int i:= 6;
    while (i+17)
    {
        int j := 8/i;
        i := i-1;
    }
}
```

Invalid pointer dereference can occur when trying to access data members of an uninitialized class variable. For example, here:

```
class Father { int i; int j; }
Father f;
int i := f.i;
```

And the same behaviour is expected here too:

```
class Father { int i; int j; }
Father f;
int i := f.i;
```

When an invalid pointer dereference occurs, the program should print "Invalid Pointer Dereference", and then then exit gracefully by using an exit system call.

Out of bounds array access happens when an array is accessed beyond its allocated size. The following code demonstrates a possible scenario:

```
array IntArray = int[]
IntArray A := new int[6];
int i := A[18];
```

In this case "Access Violation" should be printed, and then exit gracefully by using an exit system call. The same behaviour is expected here too:

```
array IntArray = int[]
IntArray A := NIL;
int i := A[13];
```

Poseidon Syntax

To avoid an overly complex exercise, we will exclude class methods from it. This means that our classes are like structures from the C programming language. Here is the grammar for Poseidon without class methods:

```
Program  ::= dec+

dec      ::= varDec | funcDec | classDec | arrayDec

varDec   ::= ID ID [ ASSIGN exp ] ';'
         ::= ID ID ASSIGN newExp ';'
funcDec  ::= ID ID ( [ ID ID [ ',' ID ID ]* ] ) { stmt [ stmt ]* }
classDec ::= CLASS ID [ EXTENDS ID ] { cField [ cField ]* }
arrayDec ::= ARRAY ID = ID '[' ']'

exp      ::= var
         ::= ( exp )
         ::= exp BINOP exp
         ::= [ var '.' ] ID ( [ exp [ ',' exp ]* ] )
         ::= -INT | NIL | STRING

var      ::= ID
         ::= var '.' ID
         ::= var '[' exp ']'

stmt     ::= varDec
         ::= var ASSIGN exp ';'
         ::= var ASSIGN newExp ';'
         ::= RETURN [ exp ] ';'
         ::= IF    ( exp ) { stmt [ stmt ]* }
         ::= WHILE ( exp ) { stmt [ stmt ]* }
         ::= [ var '.' ] ID ( [ exp [ ',' exp ]* ] ) ';'

newExp   ::= new ID | new ID '[' exp ']'

cField   ::= varDec

BINOP    ::= + | - | * | / | < | > | =

INT      ::= [1 - 9][0 - 9]* | 0
```

Input

The input for this exercise is a single text file: a semantically valid Posiedon program without class methods.

Output

The output of this exercise is a single human readable (*.ll) bitcode file according to the standard of LLVM `6.0.0`.

Submission Guidelines

The skeleton code for this exercise resides (as usual) in subdirectory EX4 of the course repository. COMPILATION/EX4 should contain a makefile building your source files to a runnable jar file called COMPILER (note the lack of the .jar suffix). Feel free to use the makefile supplied in the course repository, or write a new one if you want to. Before you submit, make sure that your exercise compiles and runs on Ubuntu 18.04.1 LTS. This is the formal running environment of the course.

Execution parameters

COMPILER receives 2 input file names:

- InputPoseidonProgram.txt
- OutputBitcodeFile.ll