EXERCISE #9

ABSTRACT INTERPRETATION REVIEW

Write your name and answer the following on a piece of paper

- Describe the purpose of using an abstract domain for dataflow analysis

ADMINISTRIVIA AND ANNOUNCEMENTS

Exam 1 is Wednesday

- Topic list linked and updated on https://analysis.cool

Carmack lecture is Friday

- Video linked on https://analysis.cool

Please sign up for Piazza if you haven't already!

- Video linked on https://analysis.cool

LLVM BITCODE

EECS 677: Software Security Evaluation

Drew Davidson

CLASS PROGRESS

EXPLORING STATIC ANALYSIS

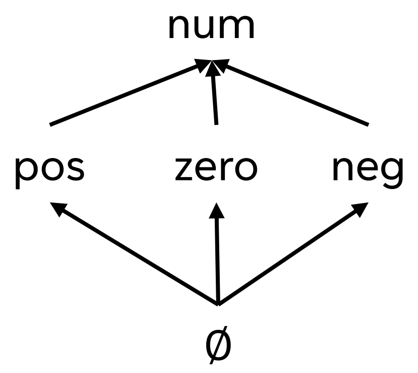
- FINISHED ENOUGH INTUITION THAT WE CAN PERFORM A BASIC ANALYSIS
- TIME TO EXPLORE OUR ANALYSIS TARGET FORMAT

LAST TIME: ABSTRACT INTERPRETATION

REVIEW: LAST LECTURE

PRECISION / EFFICIENCY TRADEOFF

- Overapproximate the domain
- Rebuild the transfer functions

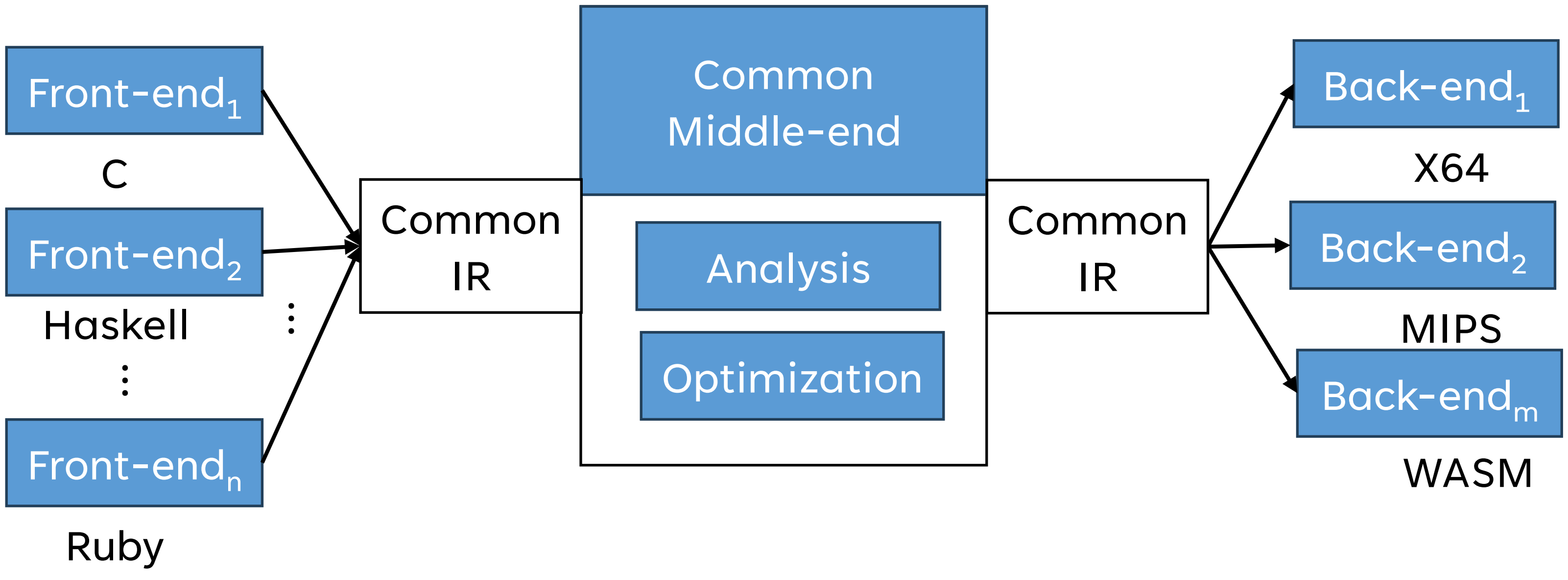


LAST TIME: LLVM

REVIEW: LAST LECTURE

A SET OF PROGRAM MANIPULATION TOOLS BUILT AROUND A "MID-LEVEL" ABSTRACT INSTRUCTION SET

- Called an intermediate representation (IR) because it sits between source code and executable
- High level enough to avoid architecture lock-in
- Low level enough to optimize / provide explicit operational details



LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- SSA Format

LLVM'S "UNIVERSAL IR"

LLVM BITCODE

Bit-Code Language Design Goals

An in-memory compiler IR

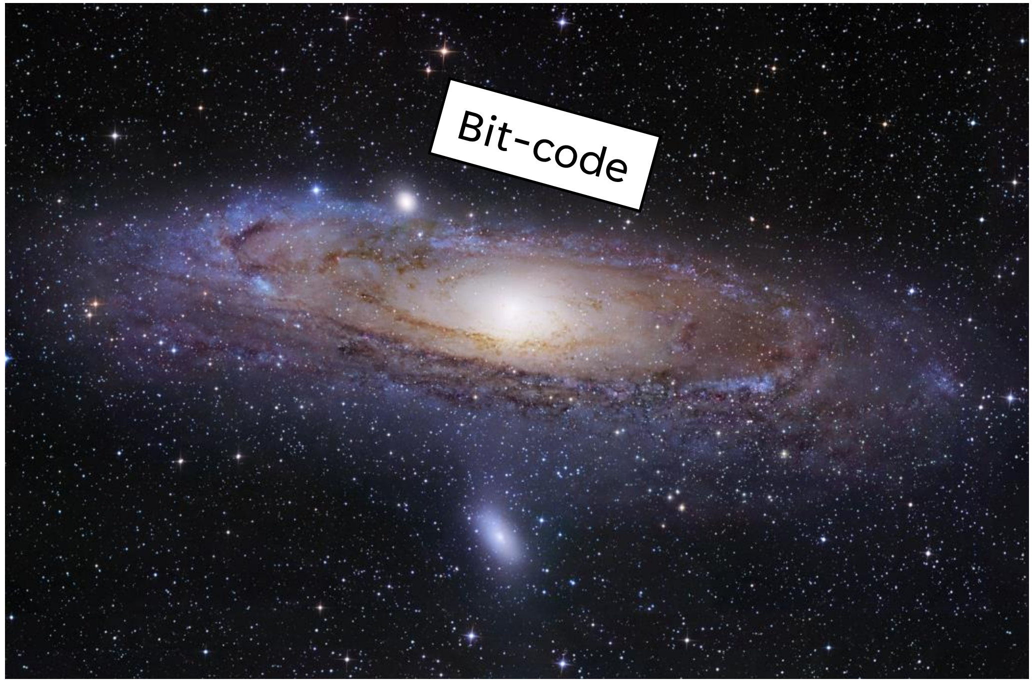
An on-disk program representation

A human readable assembly language

A Compiler's Representation

Relatively generic

Relatively easy to analyze



BITCODE STRUCTURE

LLVM BITCODE

Nested Structure

Modules

Individual translation unit (can be a whole program)

Functions

Invokable execution units

Global variables (globals)

Regions of statically-allocated memory

Local variables

Regions of dynamically-allocated memory

Instructions

Data transformers

Registers

~~*Data transformers*~~ value holders

AN ABSTRACT COMPUTER

LLVM BITCODE

No real computer runs bitcode natively*

Abstract representation of memory

Highly-explicit instructions

*Without some additional translation software

LLVM'S ABSTRACT MEMORY

LLVM BITCODE

Named Memory Objects

No explicit layout between objects

Sized field within the object

Highly-explicit instructions

Abstract registers

Infinite number of registers

EXAMPLE-DRIVEN LEARNING

LLVM BITCODE

Before we get too lost in the details, let's explore bit-code with an example

LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- SSA Format

AN EXAMPLE PROGRAM

LLVM BITCODE

Source code

```
int main(){
        return 7;
}
```

Basically-equivalent bit-code

```
define i32 @main() #0 {
  ret i32 7
}
```

AN EXAMPLE PROGRAM - MATH

LLVM BITCODE

Source code

```
int main(int argc){
        return argc + 5;
}
```

Basically-equivalent bit-code

```
define i32 @main(i32 %argc) {
        %val = add i32 %argc, 5
        ret i32 %val
}
```

% precedes a register name

No nested operations!

AN EXAMPLE PROGRAM - JUMPS

LLVM BITCODE

Source code

```
int main(int argc){
        if (argc == 1){
                return 1;
        } else {
                return 2;
        }
}
```

Basically-equivalent bit-code

```
define i32 @main(i32 %argc) {
lbl_head:
        %noArgs = icmp eq i32 %argc, 1
        br i1 %noArgs, label %lbl_t, label %lbl_f
lbl_t:
        ret i32 1
lbl_f:
        ret i32 2
}
```

All blocks must end in a terminator instruction

SIMPLE INSTRUCTION SET

LLVM BITCODE – VERY SIMPLE EXAMPLES

Math

The `add` instruction for addition
The `mul` instruction for multiplication
The `sub` instruction for subtraction
The `div` instruction for division

Control Flow

The `br` instruction for branching

- Predicate + multiple targets for conditional branch
- No predicate + 1 target for unconditional branch

Comparison

The `icmp <kind>` for integer comparison
Where kind is...

`eq:` equal
`ne:` not equal
`ugt:` unsigned greater than
`uge:` unsigned greater or equal
`ult:` unsigned less than
`ule:` unsigned less or equal
`sgt:` signed greater than
`sge:` signed greater or equal
`slt:` signed less than
`sle:` signed less or equal

RUNNING BITCODE PROGRAMS

LLVM BITCODE – VERY SIMPLE EXAMPLES

LLI – A RUNTIME ENVIRONMENT FOR BIT-CODE PROGRAMS!

SECTION SUMMARY

LLVM BITCODE – VERY SIMPLE EXAMPLES

WE CAN WRITE SIMPLE PROGRAMS USING THE INSTRUCTIONS GIVEN

WE CAN WRITE RUN SIMPLE PROGRAMS USING LLI

LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- Format Constraints - SSA

AN INCORRECT PROGRAM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

THIS PROGRAM IS INVALID!

```
define i32 @main(i32 %0) {
        %reg = add i32 %0, 5
        %reg = add i32 %0, 5
        ret i32 %2
}
```

THE REGISTER %REG IS NOT

IS NOT IN **<u>SSA FORM</u>**

```
lli: badSSA.ll:3:2: error: multiple def
inition of local value named 'reg'
        %reg = add i32 %0, 5
        ^
```

SSA FORM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

IN STATIC SINGLE ASSIGNMENT FORM, A VARIABLE (HERE, REGISTER) MAY BE ASSIGNED IN AT MOST ONE PROGRAM POINT

```
define i32 @main(i32 %argc) {
        %v1 = add i32 %argc, 1
        %v1 = mul i32 %v1, 7
        %v1 = sub i32 %v1, 2
        ret i32 %v1
}
```

```
define i32 @main(i32 %argc) {
        %v1 = add i32 %argc, 1
        %v2 = mul i32 %v1, 7
        %v3 = sub i32 %v2, 2
        ret i32 %v3
}
```

SSA FORM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

IN STATIC SINGLE ASSIGNMENT FORM, A VARIABLE (HERE, REGISTER) MAY BE ASSIGNED IN AT MOST ONE PROGRAM POINT

Is this program in SSA form?

Yes!

```
define i32 @main(i32 %argc) {
loop:
        %v1 = add i32 %argc, 1
        br label %loop
}
```

Is this program in SSA form?

no!

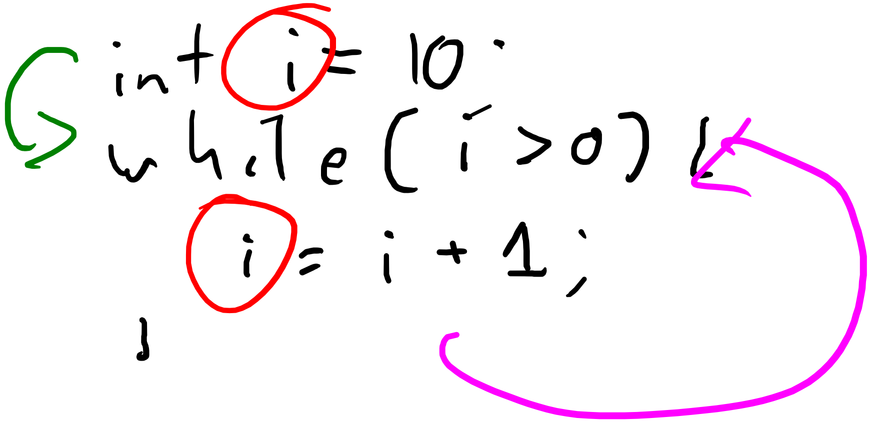
```
define i32 @main(i32 %argc) {
lbl_head: %noArgs = icmp eq i32 %argc, 1
          br i1 %noArgs, label %lbl_t, label %lbl_f
lbl_t: %var = add i32 1, 0
          br label %end
lbl_f: %var = add i32 2, 0
          br label %end
end: ret i32 %var
}
```

PHI FUNCTIONS

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

THE CONCEPTS WE HAVE SO FAR PREVENT SOME BASIC PROGRAMS FROM BEING WRITTEN

```
int main(int argc){
        while (argc > 0){
                argc = argc - 1;
        }
        return 0;
}
```



Fortunately, there is an instruction for exactly these cases:

%res = phi <type> [val_1, bbl_1], [val_2, bbl_2], … [val_n, bbl_n]

Set %res to val_i if the block was entered from bbl_i

SECTION SUMMARY

STATIC ANALYSIS

LLVM Constrains how values can be set

One solution is to use PHI instructions to unify disparate values

WRAP-UP

HOMEWORK 1
DUE FRIDAY, 9/15

WRITE AN LLVM PROGRAM THAT ITERATIVELY COMPUTES THE K^{TH} FIBONACCI NUMBER WHERE K IS THE ARG COUNT TO THE PROGRAM

NEXT TIME

LOOK AT SOME MORE COMPLEX LLVM EXAMPLES

START LOOKING AT MANIPULATING MEMORY:

- POINTERS / REF+DEREF
- STRUCTURES / ARRAYS