A compiler

High level programming language

Front-end

- Values encoding
- Checking memory accesses
- Basic block generation

Middle-end

- Explicit control flow and data types
- Instruction selection
- Register allocation

Back-end

- Legalizer
- Peephole optimizations
- Instruction scheduling

Machine code
Outline

• LA

• Encoding values

• Checking array accesses

• Generating basic blocks and variable definitions
\[ p ::= f^+ \]
\[ f ::= T \text{name} (\text{type name})* \{ \text{i}^* \} \]
\[ i ::= \text{type name} | \text{name} <- t | \text{name} <- t \text{op} t | \]
\[ \text{label} | \text{br} \text{label} | \text{br} t \text{label} \text{label} | \text{return} | \text{return} t \]
\[ \text{name} <- \text{name}([t]^+ | \text{name}([t]^+) <- t | \text{name} <- \text{length} \text{name} t | \text{name}(\text{args?}) | \text{name} <- \text{name}(\text{args?}) | \text{name} <- \text{new Array} (\text{args}) | \text{name} <- \text{new Tuple} (t) \]

\[ T ::= \text{type} | \text{void} \]
\[ \text{type} ::= \text{int64}([[]]^* | \text{tuple} | \text{code} \]
\[ \text{args} ::= t | t (, t)^* \]
\[ t ::= \text{name} | N \]
\[ N ::= (+|-)? [0-9]^+ \]
\[ \text{op} ::= + | - | * | \& | << | >> | < | <= | = | >= | > \]
\[ \text{name} ::= [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ \text{label} ::= :\text{name} \]

You cannot have invalid memory accesses in the underlying architecture.

All values are decoded!
LA standard library

• int64 input (void)

• void print (int64)

• void print (int64[])

• void print (int64[][])

• void print (int64[][][])

• ...

5
LA standard library

• int64 input (void)

• void print (int64([])*)

• void print (tuple)

No tensor-error
IR

define int64 @test (){
  :entry
  int64 %myRes
  int64 %v1
  int64 %v2
  %myRes <- call @myF(5)
  %v1 <- %myRes * 7
  %v2 <- %myRes + %v1
  return %v2 }

define int64 @myF (int64 %p1){
  :myLabel
  int64 %p1
  int64 %p2
  %p2 <- %p1 + 3
  return %p2 }

LA

int64 test (){
  int64 myRes
  int64 v1
  int64 v2
  myRes <- myF(2)
  v1 <- myRes * 3
  v2 <- myRes + v1
  return v2 }

int64 myF (int64 p1){
  int64 p1
  int64 p2
  p2 <- p1 + 1
  return p2 }

int64 test (){ 
    int64[] v 
    v <- new Array(2) 
    int64 s 
    s <- v[100] 
    return 0 
}
LA and variable definitions

• LA variables must be defined before being used

• The scope of LA’s variables is the function
  • They can be defined anywhere in the function
  • There is no semantic difference between defining a variable in the middle of the function and defining it at the beginning of that function
Final notes

• Implicit return instruction if missing

• LA integer variables are implicitly initialized to zero of the LA language
  • One in IR, L3, L2, L1, x86_64

• LA code variables are initialized to 0
  • Reading a 0 in a code variable is a bug

• Having all variables implicitly initialized and having made impossible invalid memory access leaves integer overflow as the only other undefined behavior

```c
int64 test (){  
    int64 v  
    print(v)  
}
```
Now that you know LA, rewrite your IR programs in LA
Outline

• LA

• Encoding values

• Checking array accesses

• Generating basic blocks and variable definitions
Value encoding

• Values in LA **are not** encoded

• Values in IR
  (beside array/tuple indices, and the 2\textsuperscript{nd} parameter of length) **are** encoded

• The LA compiler needs to encode the values
  • Solution 1: (this class)
    • Everything is encoded
    • When needed, decode just before uses
  • Solution 2: (advanced)
    • Everything is decoded
    • Encode just before invoking the runtime
Value encoding example

LA

```plaintext
int64 v1
...
br v1 :L1 :L2
...
```

IR

```plaintext
in64 %v1
int64 %v1_new
%v1 <- 1
...
%v1_new <- v1 >> 1
br %v1_new :L1 :L2
...
```

Integer variables are initialized to “zero”, which encoded is 1
Value encoding example 2

LA

Int64[] v1
...
br v1 :L1 :L2
...

IR

In64[] %v1
...
br %v1 :L1 :L2
...
Value encoding example 3

LA

int64 v1
int64 v2
int64 v3
...
v3 <- v1 + v2
...

IR

int64 %v1
int64 %v2
int64 %v3
int64 %v1_new
int64 %v2_new
...
%v1_new <- %v1 >> 1
%v2_new <- %v2 >> 1
%v3 <- %v1_new + %v2_new
%v3 <- %v3 << 1
%v3 <- %v3 + 1
...
Value encoding: toDecode(i)

1. toDecode(br t label label) = t
2. toDecode(var1 <- length var2 var3) = var3
3. toDecode(var1 <- var2([vari])⁺ ) = ([vari])⁺
4. toDecode(var1([vari])⁺ <- s) = ([vari])⁺
5. toDecode(var <- t1 op t2 ) = t1, t2
6. toDecode(everything else) =
Value encoding: toEncode(i)

1. toEncode(var <- t op t) = var
2. toEncode(everything else) =
Value encoding algorithm

1. Encode all constants not used for array/tuple indices and 2\textsuperscript{nd} parameter of length

2. For each instruction $i$
   A. For every variable or number $t$ in $\text{toDecode}(i)$
      I. Create a new variable $v'$ and place its definition to the first basic block
      II. Store the decoded value of $t$ in $v'$
      III. Change $i$ to use $v'$ instead of $t$
   B. For every variable $v$ in $\text{toEncode}(i)$
      I. Encode $v$ just after $i$
Outline

• LA

• Encoding values

• Checking array accesses

• Generating basic blocks and variable definitions
p ::= f^+
f ::= T name ( (type name)* ) { i* }
i ::= type name | name <- t | name <- t op t |
    label | br label | br t label label | return | return t
    name <- name([t])^+ | name([t])^+ <- t |
    name <- length name t |
    name( args? ) | name <- name( args? ) |
    name <- new Array(args) | name <- new Tuple(t)

T ::= type | void

type ::= int64([])* | tuple | code

args ::= t | t (, t)*

t ::= name | N

N ::= (+|-)? [0-9]^*

op ::= + | - | * | & | << | >> | < | <= | = | >= | >

name ::= [a-zA-Z_][a-zA-Z_0-9]*

label ::= := name
Checking array/tuple accesses

To avoid invalid memory accesses, we need to generate code to

1. Check if an array or tuple has been allocated before their use

2. Check if the indexes used to access an array are within the allowed ranges
Checking array/tuple allocation

To check if an array or tuple has been allocated before their use

1. Initialize an array/tuple variable to 0 at their declaration

2. Before every array/tuple access:
   I. Check if the array/tuple variable $v$ is not 0
   II. If it is 0, then call tensor-error(int64 line)
       passing the line number of the LA file
       of the incorrect instruction
Checking allocation example

LA

41 int64[] v1
42 ...
43 v3 ← v1[0]

IR

static void apply( const Input & in,
        Program & p){
    auto ln = in.position().line;
}

Line numbers
of the LA file
Checking allocation example

LA

```
int64[] v1
...
v3 <- v1[0]
```

Line numbers of the LA file

IR

```
int64 %LineNumber
in64[] %v1
%v1 <- 0
...
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
...
%v3 <- %v1[0]
```

Checking whether the memory has been allocated or not
Checking allocation example

LA

Int64[] v1
...
v3 <- v1[0]

41
42
43

Line numbers of the LA file

IR

int64 %LineNumber
in64[] %v1
%v1 <- 0
...
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
...
%v3 <- %v1[0]

Error reporting
Checking whether the memory has been allocated or not
Checking allocation example

LA

Int64[] v1
...
v3 <- v1[0]

41 42 43

Line numbers of the LA file

IR

int64 %LineNumber
in64[] %v1
%v1 <- 0
...
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
...
%v3 <- %v1[0]

Error reporting
Checking whether the memory has been allocated or not

Checking whether the memory offset is within the object allocated

27
Checking single-dimension array accesses

Check if the index used to access a single-dimension array are within the allowed range

1. Before every array access (and hence after the above check)
   A. For the index i used (e.g., ar[i])
      I. Load the length of the relative dimension (e.g., l_i <- length ar 0)
      II. Check if i is less than that length (e.g., i < l_i)
      III. If it isn’t, then call tensor-error(int64 line, int64 length, int64 index)

You can assume indexes are not negative
Checking tensor accesses

Check if the indexes used to access a tensor are within the allowed ranges

1. Before every tensor access (and hence after the above check)
   A. For every index i (e.g., t[k][i][j])
      I. Load the length of the relative dimension (e.g., l_i <- length t 1)
      II. Check if i is less than that length (e.g., i < l_i)
      III. If it isn’t, then call tensor-error(int64 line, int64 d, int64 length, int64 index)

You can assume indexes are not negative
Outline

• LA

• Encoding values

• Checking array accesses

• Generating basic blocks and variable definitions
Instructions without basic blocks

Instructions organized in basic blocks

```
Inst = F.entryPoint() ; newInsts = new List() ; startBB = true ;
While (Inst){
    if (startBB){ // Next instruction must be a label
        if (Inst is not Label) {
            L = new Label() ; newInsts.append(L) ;
        }
        startBB = false;
    } else if (Inst is Label){ // L1 i+ L2 ...
        g = new Goto(Inst); newInsts.append(g) ; // L1 i+ goto L2 ...
    }
    newInsts.append(Inst) ;
    if (Inst is Terminator) { // Next instruction must be a label
        startBB = true;
    }
    Inst = F.nextInst(Inst);
}
```

See next slide
If (!startBB){
  if (Function’s return type is void){
    r = new Return();
  } else {
    r = new Return(0);
  }
  newInsts.append(r);
}
Variable definitions

• Collect all LA variable definitions

• Generate IR variable definitions (in any order) at the beginning of the first basic block

• Append to the first basic block the initialization of code variables to 0
The LA compiler (LAc)

LA program

Your work

prog.IR

IRc

a.out

• To build LAc:
  translate an LA program to an equivalent IR one

• We need to
  • Encode values
  • Generate code to check memory accesses
  • Create basic blocks
Homework #6

Write a compiler that translates an LA program (.a) to an IR one

- You need to generate prog.IR
- You need to pass all tests in the framework
Always have faith in your ability

Success will come your way eventually

Best of luck!