Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• IR

• Explicit control flows

• Explicit data types
A compiler

High level programming language

Front-end

Middle-end

Back-end

Today: translating explicit control flow and data types

Instruction selection

Register allocation

Assembly generation

Machine code
define :main (){  
    %myRes <- call :myF(5)  
    %v1 <- %myRes * 4  
    %v2 <- %myRes + %v1  
    return  
}  
define :myF (%p1){  
    %p2 <- %p1 + 1  
    return %p2  
}
p ::= f^+
f ::= define label ( vars ) { i* }
i ::= var <- s | var <- t op t | var <- t cmp t |
    var <- load var | store var <- s |
    return | return t | label | br label | br t label |
    call callee ( args ) | var <- call callee ( args )
callee ::= u | print | allocate | input | tensor-error
vars ::= | var | var (, var)*
args ::= | t | t (, t)*
s ::= t | label
t ::= var | N
top ::= var | label
op ::= + | - | * | & | << | >>
cmp ::= < | <= | = | >= | >
N ::= (+|-)? [1-9][0-9]*
label ::= :name
var ::= %name
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
\textit{IR}

\begin{verbatim}
define int64 :myF (int64 %p1){
    :myLabel
    int64 %v1
    %v1 <- 1
    int64[][] %myMatrix
    int64[][][][][] %myTensor
tuple %myHeterogeneousArray
code %myFunctionPointer
%myFunctionPointer <- :myF
return %v1
}
\end{verbatim}
\[ p ::= f^+ \]
\[ f ::= \text{define } T \text{ label } ( \text{type } \text{var})^* \} \{ \text{bb}^+ \} \]
\[ \text{bb} ::= \text{label } i^* \text{ te} \]
\[ \text{te} ::= \text{br label } | \text{br t label label } | \text{return } | \text{return } t \]
\[ i ::= \text{type } \text{var } | \text{var } <- s \text{ var } <- t \text{ op } t \]
\[ \text{var } <- \text{var}(\text{t})^+ \text{ var}(\text{t})^+ <- s \text{ var } <- \text{length } \text{var } t \]
\[ \text{call callee } ( \text{args}?) \text{ var } <- \text{call callee } ( \text{args}?) \]
\[ \text{var } <- \text{new Array}(\text{args}) \text{ var } <- \text{new Tuple}(t) \]

\[ T ::= \text{type } | \text{void} \]
\[ \text{type} ::= \text{int64}([])^* \text{ tuple } | \text{code} \]
\[ \text{callee} ::= u | \text{print } | \text{input } | \text{tensor-error} \]
\[ \text{args} ::= t | t (, t)^* \]
\[ s ::= t | \text{label} \]
\[ t ::= \text{var } | N \]
\[ u ::= \text{var } | \text{label} \]
\[ N ::= (+|-)? [1-9][0-9]^* \]
\[ \text{op} ::= + | - | * | \& | << | >> | < | <= | = | >= | > \]
\[ \text{label} ::= [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ \text{var} ::= \text{sequence of chars matching } %[a-zA-Z_][a-zA-Z_0-9]^* \]

```plaintext
define int64 :myF (int64 %p1){
    :myLabel
    int64[] %v
    %v <- new Array(7)
    %v[1] <- 1
    return 0
}
```
IR

\[
p ::= f^* \\
f ::= \text{define } T \text{ label } (\text{type var})* \{ \text{bb}^* \} \\
\text{bb} ::= \text{label } i* \text{ te} \\
\text{te} ::= \text{br} \text{ label} | \text{br} \text{ t label label} | \text{return} | \text{return } t \\
i ::= \text{type } \text{var} | \text{var} <- s | \text{var} <- t \text{ op } t | \text{var} <- \text{var}([t])^* | \text{var}([t])^* <- s | \text{var} <- \text{length } \text{var } t | \text{call callee } (\text{args}?) | \text{var} <- \text{call callee } (\text{args}?) | \text{var} <- \text{new } \text{Array}(\text{args}) | \text{var} <- \text{new } \text{Tuple}(t) \\
T ::= \text{type} | \text{void} \\
\text{type} ::= \text{int64}([])^* | \text{tuple} | \text{code} \\
\text{callee} ::= u | \text{print} | \text{input} | \text{tensor-error} \\
\text{args} ::= t | t (, t)^* \\
\text{s} ::= t | \text{label} \\
\text{t} ::= \text{var} | N \\
\text{u} ::= \text{var} | \text{label} \\
N ::= ( +| -)? [1-9][0-9]^* \\
\text{op} ::= + | - | * | \& | << | >> | < | <= | = | >= | > \\
\text{label} ::= [a-zA-Z_][a-zA-Z_0-9]^* \\
\text{var} ::= \text{sequence of chars matching } %[a-zA-Z_][a-zA-Z_0-9]^* \\
\]

\[
\text{define int64 } :\text{myF } (\text{int64 } %p1)\{ \\
\quad :\text{myLabel} \\
\quad \text{int64}[][] %v \\
\quad %v <- \text{new } \text{Array}(7,7) \\
\quad %v[0][3] <- 1 \\
\quad \text{return } 0 \\
\} \\
\]
IR

define int64 :myF (int64 %p1) {
  :myLabel
  int64[][] %v
  %v <- new Array(7,7,7)
  %v[0][1][3] <- 1
  return 0
}

p ::= f*
f ::= define T label ( (type var)* ) { bb* }
bb ::= label i* te
te ::= br label | br t label label | return | return t
i ::= type var | var <- s | var <- t op t |
    var <- var([t]) | var([t]) <- s | var <- length var t |
    call callee ( args? ) | var <- call callee ( args? ) |
    var <- new Array(args) | var <- new Tuple(t)
T ::= type | void
type ::= int64([]) | tuple | code
callee ::= u | print | input | tensor-error
args ::= t | t (, t)*
s ::= t | label
t ::= var | N
u ::= var | label
N ::= (+|-)? [1-9][0-9]*
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
label ::= [a-zA-Z_][a-zA-Z_0-9]*
var ::= sequence of chars matching %[a-zA-Z_][a-zA-Z_0-9]*
\[
\begin{align*}
p & := f^+ \\
f & := \text{define } T \text{ label ( (type var)* ) } \{ \text{bb*} \} \\
\text{bb} & := \text{label } i^* \text{ te} \\
\text{te} & := \text{br label } \mid \text{br t label label } \mid \text{return } \mid \text{return t} \\
i & := \text{type var } \mid \text{var } <- \text{ s } \mid \text{var } <- \text{ t op } \text{ t} \\
& \quad \quad \quad \text{var } <- \text{ var([t])* } \mid \text{var([t])* } <- \text{ s } \mid \text{var } <- \text{ length var } \text{ t} \\
& \quad \quad \quad \text{call callee ( args? ) } \mid \text{var } <- \text{ call callee ( args? ) } \\
& \quad \quad \quad \text{var } <- \text{ new Array(args) } \mid \text{var } <- \text{ new Tuple(t)} \\
T & := \text{type } \mid \text{void} \\
\text{type} & := \text{int64([[]]* } \mid \text{tuple } \mid \text{code} \\
\text{callee} & := u \mid \text{print } \mid \text{input } \mid \text{tensor-error} \\
\text{args} & := t \mid t (, t)^* \\
\text{s} & := t \mid \text{label} \\
\text{t} & := \text{var } \mid \text{N} \\
\text{u} & := \text{var } \mid \text{label} \\
N & := (+|-)? [1-9][0-9]^* \\
\text{op} & := + \mid - \mid * \mid \& \mid << \mid >> \mid < \mid <\mid = \mid >\mid >\mid > \\
\text{label} & := [a-zA-Z_] [a-zA-Z_0-9]^* \\
\text{var} & := \text{sequence of chars matching } %[a-zA-Z_] [a-zA-Z_0-9]^*
\end{align*}
\]
Variable definition

• The code must defines (statically) a variable before using it

• In other words, the variable definition must appear in the function before all of its uses

```
int64 %d
%d <- 5
int64 %d
```
Final notes on IR

Same undefined behaviors as for L3
but without the one related to the last instruction of a function

L3

```c
define :myF (%p1){
    %p2 <- %p1 + 1
}
```

IR

```c
define void :myF (int64 %p1){
    int64 %p2
    %p2 <- %p1 + 1
    return
}
```
Now that you know the IR language

Rewrite your L3 programs in IR and

write a new IR program that uses tensors
with more than 40 instructions

don’t forget to add the *.in files if you call input()
Outline

• IR

• Explicit control flows

• Explicit data types
IR features

• Basic blocks and control Flow Graph (CFG)
  • The middle-end job: **analyze, analyze, analyze**, and transform
  • To help analyzing the IR: explicit control flow
  • Liveness analysis is an example of what the middle-end does
  • Your liveness analysis had to “learn” who were the successors of an instruction
  • Successor/predecessor of an instruction: control flows
  • If I have 1000 code analyses, do they all have to “learn” the control flows?
  • Control flows need to be explicit in the code to simplify the middle-end
Representing the control flow of the program

- Most instructions
- Jump instructions
- Branch instructions
Representing the control flow of the program

A graph where nodes are instructions
- Very large
- Lot of straight-line connections
- Can we simplify it?

Basic block
Sequence of instructions that is always entered at the beginning and exited at the end
Basic blocks

A basic block is a maximal sequence of instructions such that

- Only the first one can be reached from outside this basic block
- All* instructions within are executed consecutively if the first one get executed
  - Only the last instruction can be a branch/jump
  - Only the first instruction can be a label
- The storing sequence = execution order in a basic block
Basic blocks

- Automatically identified
- Algorithm:
  - Code changes trigger re-identification
  - Increase the compilation time
  - Enforced by design
  - Instruction exists only within the context of its basic block
- To define a function:
  - you define its basic blocks first
  - Then you define the instructions of each basic block

Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
  if Inst is Label && B \notin \emptyset { 
    B = new BasicBlock()
  }
  B.add(Inst)
  if Inst is Branch/Jump{
    B = new BasicBlock()
  }
  Inst = F.nextInst(Inst)
}
Add missing labels
Add explicit jumps
Delete empty basic blocks

What about calls?
- Program exits
- Exceptions
Control Flow Graph (CFG)

- A CFG is a graph $G = \langle \text{Nodes}, \text{Edges} \rangle$
- Nodes: Basic blocks
- Edges: $(x,y) \in \text{Edges}$ iff the first instruction in basic block $y$ might be executed just after the last instruction of the basic block $x$
Control Flow Graph (CFG)

• Entry node: block with the first instruction of the function
• Exit nodes: blocks with the return instruction
• All basic blocks beside the first can be stored in any order
\[
p \ ::= f^+ \\
 f \ ::= \text{define } T \text{ label } ( (\text{type var})^* ) \{ \text{bb}^* \} \\
 \text{bb} \ ::= \text{label } i^* \text{ te} \\
 \text{te} \ ::= \text{br label } | \text{br t label label } | \text{return } | \text{return t} \\
 i \ ::= \text{type var } | \text{var } \leftarrow \text{s } | \text{var } \leftarrow \text{t op t } | \\
 & \quad \text{var } \leftarrow \text{var}([t])^+ | \text{var}([t])^+ \leftarrow \text{s } | \text{var } \leftarrow \text{length var t } | \\
 & \quad \text{call callee ( args? ) } | \text{var } \leftarrow \text{call callee ( args? ) } | \\
 & \quad \text{var } \leftarrow \text{new Array(args) } | \text{var } \leftarrow \text{new Tuple(t) } \\
 T \ ::= \text{type } | \text{void} \\
 \text{type} \ ::= \text{int64}([])^* | \text{tuple } | \text{code} \\
 \text{callee} \ ::= \text{u } | \text{print } | \text{input } | \text{tensor-error} \\
 \text{vars} \ ::= \text{var } | \text{var } , \text{var }^* \\
 \text{args} \ ::= \text{t } | \text{t } (, \text{var})^* \\
 \text{s} \ ::= \text{t } | \text{label} \\
 \text{t} \ ::= \text{var } | \text{N} \\
 \text{u} \ ::= \text{var } | \text{label} \\
 \text{op} \ ::= + | - | * | \& | << | >> | < | <= | = | >= | > \\
 \text{label} \ ::= \text{[a-zA-Z][a-zA-Z0-9]^*} \\
 \text{var} \ ::= \text{sequence of chars matching } %\text{[a-zA-Z][a-zA-Z0-9]^*} \\
\]

**IR**

```plaintext
define void :main (){
    :entry
    call :myF(1, 2)
    return
}
```
IR

define void :main (){call print(1) return}
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3
• Any order will preserve the original semantics as long as the entry point BB is the first one (property of the CFG)
Naïve solution (not ok for your homework)

• Ignore the problem

• In other words:
  the sequence of basic blocks described in the IR program file is going to be the sequence chosen

• Translate a two labels IR branch into 2 branches in L3

\[
\begin{align*}
br & \%\text{cond} : \text{TRUE} : \text{FALSE} \\
\text{Your work} \\
\end{align*}
\]
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3
• Any order will preserve the original semantics as long as the entry point BB is the first one (property of the CFG)
• Different orders will have a different #branches
• We want to select the one with the lowest #branches
  • Run-time vs. compile-time
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3

No jump needed
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3

```assembly
... br %c :LABEL_D
:LABEL_C ...
:LABEL_D ...
```
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3

Let us assume the loop B-C is executed many times. How many branches do we execute per loop iteration? Is this the best we can do?
From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3
From CFG to a sequence of instructions

• CFG is a 2-dimension representation
• L3 is a 1-dimension representation
• We need to linearize CFG to generate L3

How many branches do we execute per loop iteration?
CFG linearization

• A trace is a sequence of basic blocks (instructions) that could be executed at run time
  • It can include conditional branches

• A program has many overlapping traces

• For our goal:
  • Find a set of traces that cover the whole function without any overlapping
    • Each basic block belongs to exactly 1 trace
  • Remove unconditional branches within the same trace
Finding the not overlapping traces

list <- all basic blocks

do{
    tr = new trace()
    bb = fetch_and_remove(list)
    while (bb is not marked){
        mark bb
        tr.append(bb)
        succs = successors(bb)
        if there is $c \in$ succs such that $c$ is unmarked and profitable(bb, c)
            bb = c
    }
} while (list is not empty)
Outline

• IR

• Explicit control flows

• Explicit data types
IR features

• Basic blocks and control Flow Graph (CFG)
  • The middle-end job: analyze, analyze, analyze, and transform
  • To help analyzing the IR: explicit control flow

• Data types
  • Multi dimension arrays

```
define int64 :myF (int64 %p1){
    :myLabel
    int64 %p2
    %p2 <- %p1 + 1
    return %p2
}
```
Multi-dimension arrays

- Implicit initialization to “1”
- Accessing array elements only in simple assignments
Indices and dimension# in length are not encoded

• Accessing length of a dimension
  $%l \leftarrow \text{length } %ar %\text{dimID}$

• Accessing array element
  $%ar[%e1][%e2] \leftarrow %v1$
  $%v2 \leftarrow %ar[%e1][%e2]$

• Allocating an array
  $%ar \leftarrow \text{new Array}(%dim1, %dim2)$
Multi-dimension arrays

• Implicit initialization to “1”

• Accessing array elements only in simple assignments

• The IR compiler must linearize all arrays
  • Data layout

• The IR compiler must store the dimension lengths
  • Data layout

```
int64[][] %m
int64 %e
int64 %l0
int64 %l1
%m <- new Array(7,7)
%m[0][0] <- 3
%m[2][1] <- 7
%e <- %m[0][0]
call print(%e)
%l0 <- length %m 0
%l1 <- length %m 1
```
Storing the lengths

int64[][] %m
%m <- new Array(7,9)
... <- length %m 0
... <- length %m 1

(3 * 4) + 1 + 2
Encoded “2” (#dimensions)
Translating length

... %l1 <- length %a 1 ...
...

Your work
... %v0 <- 1 * 8
%v1 <- %v0 + 16
%v2 <- %a + %v1
%l1 <- load %v2 ...
...

IR

L3
Translating new Array()

```plaintext
Int64[][] %a
%a <- new Array(%p1,%p2)

Computing the number of memory locations

Your work

arrayLength
#dimensions
%p1
%p2
...

Why 3?

Why 5?

Encoding

%p1D <- %p1 >> 1
%p2D <- %p2 >> 1
%v0 <- %p1D * %p2D
%v0 <- %v0 + 3
%v0 <- %v0 << 1
%v0 <- %v0 + 1
%a <- call allocate(%v0, 1)
%v1 <- %a + 8
store %v1 <- 5
%v2 <- %a + 16
store %v2 <- %p1
%v3 <- %a + 24
store %v3 <- %p2

...
Linearize an array

- \( m[0][0] \)
  - \( o1 \leftarrow 16 \)
  - \( o2 \leftarrow 2 \times 8 \)
  - \( o \leftarrow o1 + o2 \)
  - \( a \leftarrow m + o \)
  - store \( a \leftarrow \ldots \)

- \( m[0][1] \)?
  - By row, by column

```plaintext
int64[][] m
m \leftarrow \text{new Array}(7,9)
%m[0][0] \leftarrow \ldots
```
Data layout for this class

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,00</td>
<td>0,11</td>
<td>0,22</td>
<td>0,33</td>
</tr>
</tbody>
</table>

• Matrix M x N
  - Offset for all: \( B = 16 + (2 \times 8) \)
  - Offset \( A[0][1] = B + (1) \times 8 \)
  - Offset \( A[0][2] = B + (2) \times 8 \)
  - Offset \( A[0][i] = B + (i) \times 8 \)
  - Offset \( A[1][0] = B + (1 \times N + 0) \times 8 \)
  - Offset \( A[i][j] = B + (i \times N + j) \times 8 \)

• Tensor L x M x N: \( B = 16 + (3 \times 8) \)
  - Offset \( A[k][i][j] = B + ((k \times M \times N) + (i \times N) + j) \times 8 \)
Linearization example (2)

IR: L x M x N: %A[%k][%i][%j] <= 5

L3: Offset = 16 + (3 * 8) + (k * M * N) + (i * N) + j) * 8

- ADDR_M <- A + 24
- M_ <- load ADDR_M
- M <- M_ >> 1
- ADDR_N <- A + 32
- N_ <- load ADDR_N
- N <- N_ >> 1
- newVar1 <- i * N
- M_N <- M * N
- newVar2 <- k * M_N
- newVar3 <- newVar2 + newVar1
- index <- newVar3 + j
- offsetAfterB <- index * 8
- offset <- offsetAfterB + 40
- addr <- A + offset
- store addr <- 5

To fetch M

To fetch N

To fetch M

To fetch N

To fetch M

To fetch N

To fetch M

To fetch N
Multi-dimension arrays

• No limit to the number of dimensions

int64[][] %m
%m <- new Array(7,9)
Int64[][][][][][][][] %crazy
%crazy <- new Array(7,7,7,7,7,7,7,7)

• The data layout follows the scheme of the previous slides
IR features

• Basic blocks and control Flow Graph (CFG)
  • The middle-end job: analyze, analyze, analyze, and transform
  • To help analyzing the IR: explicit control flow

• Data types
  • Multi dimension arrays
  • Tuples
p ::= f+
f ::= define T label ( (type var)* ) { bb* }
bb ::= label i* te
t ::= br label | br t label label | return | return t
i ::= type var | var <- s | var <- t op t |
    var <- var([t]) | var([t]) <- s | var <- length var t |
    call callee ( args? ) | var <- call callee ( args? ) |
    var <- new Array(args) | var <- new Tuple(t)
T ::= type | void
type ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
vars ::= var | var (, var)*
args ::= t | t (, t)*
s ::= t | label
t ::= var | N
u ::= var | label
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
label ::= [a-zA-Z_][a-zA-Z_0-9]*
var ::= sequence of chars matching %[a-zA-Z_][a-zA-Z_0-9]*
Tuples

• Implicit initialization to “1”
• Argument of Tuple() is encoded
• Indices are not encoded (like for arrays) but values are (like for arrays)
• A tuple is an heterogeneous 1-dimension array
• Equivalent in L3: array

tuple %t
%t <- new Tuple(7)
%t[0] <- 5
int64[] %a
%a1 <- new Array(3)
%t[1] <- %a1
%a2 <- new Array(5,3)
%t[2] <- %a2
Translating tuples

... tuple %t
%t <- new Tuple(7)
%t[0] <- 5
%v <- %t[0]
...

... %t <- call allocate(7, 1)
%newVar0 <- %t + 8
store %newVar0 <- 5
%newVar1 <- %t + 8
%v <- load %newVar1
...

Your work
IR features

• Basic blocks and control Flow Graph (CFG)
  • The middle-end job: *analyze, analyze, analyze*, and transform
  • To help analyzing the IR: explicit control flow

• Data types
  • Multi dimension arrays
  • Tuples
  • Function pointers
IR

p ::= f+
f ::= define T label ( (type var)* ) { bb* }
bb ::= label i* te
t ::= br label | br t label label | return | return t
i ::= type var | var <- s | var <- t op t |
    var <- var( [t] )+ | var( [t] )+ <- s | var <- length var t |
    call callee ( args? ) | var <- call callee ( args? ) |
    var <- new Array( args ) | var <- new Tuple( t )
T ::= type | void
type ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
vars ::= var | var ( , var )*
args ::= t | t ( , t )* 
s ::= t | label
t ::= var | N 
u ::= var | label
op ::= + | - | * | & | << | >> | < | <= | = | >= | > 
label ::= /[a-zA-Z_][a-zA-Z_0-9]*
var ::= sequence of chars matching %/[a-zA-Z_][a-zA-Z_0-9]*
Function pointers

- Instances of type “code”
- Variables of type code can only store function names
- They can be used in call instructions
- They are normal variables
- They can be stored in tuples

```plaintext
define code :myF (tuple %t){
    code %fp
    %fp <- :myOtherF
    call %fp (%firstArg,2)
    %t[0] <- %fp
    return %fp
}
```
Translating function pointers

... code %fp
%fp <- :myOtherF
call %fp(2)
...

Your work

... %fp <- :myOtherF
call %fp(2) ...

...
IR features

• Basic blocks and control Flow Graph (CFG)
  • The middle-end job: *analyze, analyze, analyze*, and transform
  • To help analyzing the IR: explicit control flow

• Data types
  • Multi dimension arrays
  • Tuples
  • Function pointers

• Values (not length dimID and not indices of array/tuples) are still encoded
Homework #5: the IR compiler (IRc)

- To build IRc: translate an IR program to an equivalent L3
- We need to linearize the arrays
- We need to translate the other IR instructions