Simone Campanoni
simonec@eecs.northwestern.edu
A compiler

Source code → Front-end → IR 

Middle-end → IR 

Back-end → Machine code

The language needs to help humans to write (efficient and robust) code

The language needs to be easy to execute efficiently

Explicit, simple, and architecture-independent instructions

Only a few registers, explicit instructions with constraints (e.g., rdi @ rsi rax 4)

High level (algorithm level) statements
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
From L2 to IR going through L3

Explicit, simple, and architecture-independent instructions designed for code analysis and transformation

Explicit semantic (e.g., add)
add, br, load, store (no lea)
no registers, no calling convention
Small piece of computation
L2 language

- Explicit entry point
- Explicit calling convention
- Complex per-instruction semantic
- Registers and variables

```
(:go (:go 0 0
   rdi <- 5
   mem rsp -8 <- :myF_ret
   call :myF 1
   :myF_ret
   %myRes <- rax
   rax @ %myRes %myRes 4
   return )
(:myF 1 0
   rax <- rdi
   return
)
```

L3 language

- Pre-defined entry point
- Hidden calling convention
- Simple per-instruction semantic
- Variables only

```
define :main (){
   %myRes <- call :myF(5)
   %v1 <- %myRes * 4
   %myRes <- %myRes + v1
   return %myRes
}
define :myF (%p1){
   %p2 <- %p1 + 1
   return %p2
}
```
\( p ::= (\text{label } f^+) \)
\( f ::= (\text{label } N N i^+) \)
\( i ::= w \leftarrow s \mid w \leftarrow \text{mem } x M \mid \text{mem } x M \leftarrow s \mid w \leftarrow \text{stack-arg } M \)
\( w \oplus t \mid w \ominus \text{sop } \text{sx} \mid w \ominus \text{sop } \text{N} \mid \text{mem } x M \oplus t \mid \text{mem } x M \ominus t \mid w \oplus \text{mem } x M \mid w \ominus \text{mem } x M \mid w \leftarrow t \ominus \text{cmp } t \mid \text{cjump } t \ominus \text{cmp } t \text{label label} \mid \text{cjump } t \ominus \text{cmp } t \text{label label} \mid \text{label label} \mid \text{goto label} \mid \text{return return} \mid \text{call } u N \mid \text{call print } 1 \mid \text{call allocate } 2 \mid \text{call array-error } 2 \mid w++ \mid w-- \mid w @ w w E \)
\( w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid r10 \mid r11 \mid r12 \mid r13 \mid r14 \mid r15 \)
\( a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{sx} \mid r8 \mid r9 \)
\( \text{sx} ::= \text{rcx} \mid \text{var} \)
\( s ::= t \mid \text{label} \)
\( t ::= x \mid N \)
\( u ::= w \mid \text{label} \)
\( x ::= w \mid \text{rsp} \)
\( \oplus ::= + \mid - \mid * \mid \&= \)
\( \ominus ::= <<= \mid >>= \)
\( \ominus ::= < \mid <= \mid = \)
\( E ::= 1 \mid 2 \mid 4 \mid 8 \)
\( M ::= N \text{ times } 8 \)
\( \text{label} ::= \text{:name} \)
\( \text{var} ::= %\text{name} \)
\( \text{name} ::= \text{sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*} \)
Explicit signature

\[ p := f^+ \]
\[ f ::= \text{define label ( vars ) \{ i^+ \}} \]
\[ i ::= \text{var <- s} \mid \text{var <- t op t} \mid \text{var <- t cmp t} \mid \text{var <- load var} \mid \text{store var <- s} \mid \text{return} \mid \text{return t} \mid \text{label} \mid \text{br label} \mid \text{br var label} \mid \text{call callee ( args )} \mid \text{var <- call callee ( args )} \]
\[ \text{callee ::= u} \mid \text{print} \mid \text{allocate} \mid \text{array-error} \]
\[ \text{vars ::= | var} \mid \text{var (, var)*} \]
\[ \text{args ::= | t} \mid \text{t (, t)*} \]
\[ \text{s ::= t} \mid \text{label} \]
\[ \text{t ::= var} \mid \text{N} \]
\[ \text{u ::= var} \mid \text{label} \]
\[ \text{op ::= +} \mid \text{-} \mid \text{*} \mid \text{&} \mid \text{<=} \mid \text{>} \]
\[ \text{cmp ::= <} \mid \text{<=} \mid \text{=} \mid \text{>=} \mid \text{>} \]
\[ \text{label ::= :name} \]
\[ \text{var ::= %name} \]

name ::= sequence of chars matching \([a-zA-Z_][a-zA-Z_0-9]*\)
define :main (){
%myRes <- call :myF(5)
%v1 <- %myRes * 4
%v2 <- %myRes + %v1
return %v2
}

define :myF (%p1){
%l1 <- %p1 + 1
return %l1
}

define :main (){  
%v1 <- 1  
%v2 <- 2  
%v3 <- %v1 >= %v2  
return %v3  
}

define :myEqual (%p1, %p2){
%v3 <- %p1 = %p2
br %v3 :myLabelTrue
return 0 :myLabelTrue
return 1
}
define :main (){  
%ret <- :myEqual(3,5)  
return %ret  
}
Now that you know the L3 language

Rewrite your L2 programs in L3
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
The L3 compiler (L3c)

- To build L3c: translate an L3 program to an equivalent L2
- We need to encode the calling convention
  \( \text{API} \rightarrow \text{ABI} \)
- We need to select which L2 instructions to use for the L3 ones
  \textit{Instruction selection}
L3 parser

• Significantly simpler than the L2 parser

• Pay attention to the L3 grammar

\[
i ::= \ldots
\]
\[
\text{call callee ( args ) | var } \leftarrow \text{call callee ( args )}
\]
\[
callee ::= u | \text{print} | \text{allocate} | \text{array-error}
\]
\[
\text{args} ::= t | t (, t)^*
\]
\[
t ::= \text{var} | N
\]

• Same rule for all call instructions
Parsing an L3 program

```
define :main (){
  %myRes <- call :myF(5)
call :myF(5)
return }
```

```
define :main (){
  %myA <- call allocate(3, 1)
call allocate(3, 1)
return }
```
Entry point

```clojure
define :main()
  ...
}
```

Your work

```
(:main
  (:main 0 0
    ...
  )
  ...
)
```
Making the calling convention explicit: caller

```
define :main()
  %v1 <- call :myF(3)
  ...
}
```

Your work

```
(:main
  (:main
    0 0
    mem rsp -8 <- :myF_ret
    rdi <- 3
    call :myF 1
    :myF_ret
    %v1 <- rax
    ...
  )
)
```
Making the calling convention explicit: callee

```
define :myF (%p1){
    return %p1
}
```

Your work

```
(:myF
  1 0
  %p1 <- rdi
  rax <- %p1
  return
)
```
Stack arguments, registers, and variables

• L3c is responsible to allocate space on the stack for >6 arguments

• L3c can generate L2 code with registers and variables

• L2c already performs a good register allocation

• Good engineering: don’t replicate functionality
  • L3c should not perform register allocation
  • L3c should use variables always with the only exceptions of implementing the calling convention
Labels

- The L3 compiler needs to translate L3 labels to L2 labels
  - L3 labels: the scope is the function
    - 2 labels with the exact name in 2 different function are possible
  - L2 labels: the scope is the program
    - 2 labels with the exact name are not possible

- A possible mapping from L3 labels to L2 ones:
  1. Find the longest label for the whole L3 program: LL
  2. Append “_global_” to it: LLG
  3. For every L3 label “:LABELNAME”, generate an L2 label by appending “LABELNAME” to LLG
define :main ( ){
  :begin
  ...
  :end
  ...
}

- LL is “:begin”
- LLG is “:begin_global_”
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
A compiler

Middle-end

IR

Back-end

Machine code

... Instruction selection

Register allocation
define :myF (%p1, %p2) {
    %v1 <- %p1 * 4
    ...
}

(:myF 2 0
  %p1 <- rdi
  %p2 <- rsi
  %v1 <- %p1
  %v1 *= 4
  ...
)
Naive translation of an L3 function: problem

define :myF (%p1, %p2){
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
...
}

Instruction selection depends on the context!

Your work

(:myF
  2 0
  %p1 <- rdi
  %p2 <- rsi
  %v1 <- %p1
  %v1 *= 4
  %v2 <- %v1
  %v2 += %p2
  ...
)

(:myF
  2 0
  %p1 <- rdi
  %p2 <- rsi
  %v2 @ %p2 %p1 4
  ...
)
Instruction selection: it isn’t that easy

define :myF (%p1, %p2){
  %v1 <- %p1 * 3
  %v2 <- %v1 + %p2
  ...
}

Instruction selection must satisfy all constraints of the target language!

( :myF
  2 0
  %p1 <- rdi
  %p2 <- rsi
  %v2 @ %p2 %p1 3
  ...
)
Instruction selection: context

• Instruction selection depends on the context
• Context for CC: sequence of instructions that does not include labels and branches

%V3 <- %v2 + %v1
%V4 <- %v3 * 4
:a_label
%V5 <- %V4 * 2
br :another_label

In CC: a context is almost a basic block (we will learn the concept of basic block next week)
Instruction selection step 1: identify contexts

Inst = F.entryPoint()
C = new Context()
While (Inst){
    if Inst is Label && C $\not\in$ "
        C = new Context()
        C.add(Inst)
    if Inst is a branch
        C = new Context()
        Inst = F.nextInst(Inst)
}
Delete empty contexts

:myLabel
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :otherLabel
Instruction selection step 2: tree generation

We need to generate a list of trees per context

1. Generate a tree per instruction
2. Merge trees
   I. Let $T_1, T_2$ be
      A. Two trees that belong to the same context
      B. $T_1$ uses a variable $%V$ defined by $T_2$
   II. Merge $T_2$ into $T_1$ only when it is safe to do so
      1. $%V$ is dead after the instruction attached to $T_1$
      2. ...

$$%v1 \gets %p1 \times 4$$
$$%v2 \gets %v1 + %p2$$
Instruction selection as tree matching

• In order to take context into account, instruction selectors use pattern-matching on trees
  • Use a tree-based code representation
  • Each target instruction defines a tile (pattern) that can be used to cover the tree
  • Used tiles (patterns) = selected target instructions to generate

{%v1 <- %p1 * 4
%v2 <- %v1 + %p2

var *= N

\begin{align*}
\%v1 & \leftarrow \%p1 \times 4 \\
\%v2 & \leftarrow \%v1 + \%p2
\end{align*}
From L3 instructions to L2 instructions

1. Translate L3 instructions of a context into a list of trees
   • Order needs to be preserved
2. Merge as many trees as possible
3. For each tree (in order):
   A. **Tiling**: cover the tree with L2 tiles
   B. **Code generation**: from the bottom to the top of the tree:
      i. Get the next tile
      ii. Append L2 instructions generated by the current tile
Example: tiles and tiling

\[ ? \times ? \]

\[ \text{VAR3} \times \text{VAR2} \]

\[ \text{VAR3} \leftarrow \text{VAR1} \]

\[ \text{VAR3} \times= \text{VAR2} \]

\[ ? \leftarrow ? \]

\[ \%v1 \leftarrow \%p1 \]

\[ \%v3 \leftarrow \%v2 \times \%v1 \]
Specialized tiles

VAR3 \* VAR1
VAR3 \*= VAR2

VAR1
VAR2

\%v1 \*= \%v2
\%v3 \*= \%v1 * 5

\%v1 \* VAR2
VAR1
VAR3

\%v1
\%v2

\%v1
VAR1

\%v1 \*= \%v2
\%v3 \*= \%v1
\%v3 \*= 5
Large tiles

```
? *= ?

VAR1 + VAR2

VAR3 <- VAR1
VAR3 += VAR2

VAR1 + VAR2

VAR2
VAR1 + VAR1
VAR1

VAR1 += VAR2

VAR1

VAR1 += var2

var1 <<= 2
```
Tiles and tiling

- Tiles capture compiler’s understanding of the target instruction set
- In general, for any given tree, many tilings are possible
  - Each resulting in a different instruction sequence
- We can ensure pattern coverage by covering, at a minimum, all atomic L3 trees
The instruction selection problem

• Many solutions to cover a tree are possible

• How to pick tiles that cover our tree with minimum execution time?

• Need a good selection of tiles
  • Small tiles to make sure we can tile every tree
  • Large tiles for efficiency
Quality of a tile in CC

• Instruction selection should prefer high-quality tiles
• The quality of a tile $t$ is related to the latency of the instructions generated by $t$
• In this class, we use the number of instructions as proxy to the latency
  • Each tile reports the number of instructions generated by it
Tiles in CC

• Tiles need to be designed such that a large tile $t$ has $\leq$ instructions than a possible set of small tiles that cover the same sub-tree

• Hence, we prefer larger tiles: fewer instructions
Quality of a solution of the tiling problem

• Tiling problem: choose a set of tiles to cover a tree

• Quality of a tiling solution: the cumulative execution time of all instructions generated to cover a tree

• In instruction selection, we estimate the total execution time as the sum of costs of all tiles
  • In CC: the cost of a tile is the number of instructions of it
  • Hence, in CC: the quality of a tiling solution is the total number of instructions generated
Example of tiling cost for L3

? * = ?

VAR3
| *
| VAR1
| VAR2

Cost: 2

VAR3 <- VAR1
VAR3 *= VAR2

%v3 <- %v2 * %v1

%v2

%v1 <- %p1
%v3 <= %v2 * %v1

? <= ?

VAR2 <- VAR1

%v1 <- %p1
%v3 <- %v2
%v3 *= %v1

Total cost: 3
Quality of a tile in state-of-the-art compilers

• Instruction selection should prefer high-quality tiles

• The quality of a tile $t$ is related to the latency of the instructions generated by $t$

• Instructions $\neq$ cycles: different instructions may take different cycles to execute
  • Each tile $t$ reports the number of clock cycles that are required to execute the generated instructions of $t$
Timing model

• Idea: associate cost with each tile (proportional to # cycles to execute)
  • Caveat: cost is fictional on modern architectures
• Estimate of total execution time is sum of costs of all tiles

Total cost: 5
Example of L2 tiles

```
VAR + CONST

VAR + VAR
VAR *
CONST 1, 2, 4
```

Cost:1

Cost:1
Global vs. local optimal solution

• We want the “lowest cost” tiling
  • Take into account cost/delay of each instruction (i.e., timing model)

• **Optimum** tiling: lowest-cost tiling

• **Locally Optimal** tiling: no two adjacent tiles can be combined into one tile of lower cost
Locally optimal tilings

• A simple greedy algorithm works extremely well in practice: Maximal munch

• Choose the largest pattern with lowest cost, i.e., the “maximal munch”

• Algorithm:
  • Start at root
  • Use “biggest” match (in # of nodes)
    • This is the munch
  • Use cost to break ties
  • Recursively apply maximal much at each subtree of this munch
Maximal munch example

```
%v2 <- %v1 + 8
%v3 <- %v2
%v3 <- 5 + %v3
%v4 <- load %v2
```

Total cost: 4

```
v4 <- mem v1 8
v3 <- v1
v3 += 8
v3 += 5
```
Maximal munch

- Maximal munch does not necessarily produce the optimum selection of instructions

But:
- it is easy to implement
- it tends to work “well” for current instruction-set architectures
... but if we want the optimum?
Finding optimum tiling

• **Goal**: find minimum total cost tiling of tree

• **Algorithm**:  
  • For every node, find minimum total cost tiling of that node and sub-tree

• **Lemma**:  
  • Once minimum cost tiling of all children of a node is known,  
  • We can find minimum cost tiling of the node by trying out all possible tiles matching the node

• **Therefore**: start from leaves, work upward to top node
Optimum selection

• To achieve optimum instruction selection: Dynamic programming

• In contrast to maximal munch, the trees are matched bottom-up

• But
  • Significantly more complex to implement
  • More time and memory consuming than maximal munch
Dynamic programming

• First pass: tiling
  • Working bottom up
  • Given the optimum tilings of all subtrees, generate optimum tiling of the current tree
    • Consider all tiles for the root of the current tree
    • Sum cost of best subtree tiles and each tile
    • Choose tile with minimum total cost

• Second pass: code generation
  • Generates the code using the obtained tiles
Value of instruction selection

• The simpler the target ISA is, the less important obtaining the optimum is
  • Reduced Instruction Set Computing (RISC)

• The more complex the target ISA is, the bigger is the gap between the solution found by a simple (e.g., maximal munch) instruction selection and the optimum one (e.g., dynamic programming)
  • Complex Instruction Set Computing (CISC)
Instruction selection complexity

• Finding the optimum for tree: P

• Finding the optimum for DAG: NP
  • Countless number of heuristics proposed

• Most (all) of programs we run are DAGs