Simone Campanoni
simonec@eecs.northwestern.edu
A compiler

Source code \hspace{1cm} \downarrow \hspace{1cm} \text{Front-end} \hspace{1cm} \downarrow \hspace{1cm} \text{Middle-end} \hspace{1cm} \downarrow \hspace{1cm} \text{Back-end} \hspace{1cm} \downarrow \hspace{1cm} \text{Machine code}

High level (algorithm level) statements

The language needs to be easy to be analyzed and transformed

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

Explicit, simple, and architecture-independent instructions

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

The language needs to be easy to be executed efficiently

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

L3 language

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

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

```bash
define :main (){
   myRes <- call :myF (5)
   v1 <- myRes * 4
   myRes <- myRes + v1
   return myRes
}
define :myF (p1){
   p2 <- p1 + 1
   return p2
}
```
\[ p ::= (\text{label } f^+) \]
\[ f ::= (\text{label } N N i^+) \]
\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } M \mid \text{mem } M \leftarrow s \mid w \leftarrow \text{stack-arg } M \]
\[ w \text{ aop } t \mid w \text{ sop } sx \mid w \text{ sop } N \mid \text{mem } x M \leftarrow+ t \mid \text{mem } x M \leftarrow- t \mid w \leftarrow+ \text{mem } x M \mid w \leftarrow- \text{mem } x M \mid w \leftarrow t \text{ cmp } t \mid \text{cjump } t \text{ cmp } t \text{ label } \text{ label } \mid \text{label } \mid \text{goto } \text{label } \mid \text{return } \mid \text{call } u \text{ N } \mid \text{call } \text{print } 1 \mid \text{call } \text{allocate } 2 \mid \text{call } \text{array-error } 2 \mid \]
\[ w++ \mid w-- \mid w \leftarrow w \ w \ E \]
\[ w ::= a \mid rax \mid rbx \mid rbp \mid r10 \mid r11 \mid r12 \mid r13 \mid r14 \mid r15 \]
\[ a ::= rdi \mid rsi \mid rdx \mid sx \mid r8 \mid r9 \]
\[ sx ::= rcx \mid \text{var} \]
\[ s ::= t \mid \text{label} \]
\[ t ::= x \mid N \]
\[ u ::= w \mid \text{label} \]
\[ x ::= w \mid \text{rsp} \]
\[ \text{aop} ::= +\mid-\mid*=\mid&= \]
\[ \text{sop} ::= <<=\mid>=> \]
\[ \text{cmp} ::= <\mid<=\mid= \]
\[ E ::= 0 \mid 2 \mid 4 \mid 8 \]
\[ M ::= N \text{ times } 8 \]
\[ \text{label} ::= :\text{var} \]
\[ \text{var} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]
L3

p ::= f^+ 

f ::= define label ( vars ) { i^+ }

i ::= var <- s | var <- t op t | var <- t cmp t |
    var <- load var | store var <- s |
    br label | label | br var label label | return | return t |
    call callee ( args ) | var <- call callee ( args )

callee ::= u | print | allocate | array-error

vars ::= var | var (, var)*

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

s ::= t | label

t ::= var | N

u ::= var | label

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

cmp ::= < | <= | = | >= | >

label ::= :sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

var ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

The scope of labels is the function!
L3 program examples

```l3
define :main (){  
  myRes <- call :myF(5)  
  v1 <- myRes * 4  
  v2 <- myRes + v1  
  return v2  
}
define :myF (p1){ 
  p2 <- p1 + 1 
  return p2 
}
define :main (){  
  v1 <- 1  
  v2 <- 2  
  v3 <- v1 < v2  
  return v3  
}
define :myEqual (p1, p2){ 
  v3 <- p1 = p2  
  br :myLabelTrue  
  return 0  
  :myLabelTrue  
  return 1  
}
define :main (){ 
  return myEqual(3,5) 
}
```
Now that you know the L3 language

develop 5 L3 programs (per team)
with at least 100 L3 instructions
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
  API -> ABI
- We need to select which L2 instructions to use for the L3 ones
  Instruction selection
L3 parser

• Significantly simpler than the L2 parser
• Pay attention to the L3 grammar
  
  \[ i ::= \ldots \]
  
  \[ \text{call callee ( args ) } | \text{var } <\text{call callee ( args )} \]

  \[ \text{callee ::= u } | \text{print } | \text{allocate } | \text{array-error} \]

  \[ \text{u ::= var } | \text{label} \]

  \[ \text{args ::= t } | \text{t (, t)*} \]

  \[ \text{t ::= var } | \text{N} \]

• Same rule for all call instructions
Parsing an L3 program

```l3
define :main (){  
  myRes <- call :myF(5)  
  call :myF(5)  
  return 
}  

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

```l3
define :main (){  
  myA <- call allocate(3, 1)  
  call allocate(3, 1)  
  return 
}
```
Entry point

define :main()
{
    ...
}

Your work

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

define :main()

v1 <- call :myF(3)
...
}

(:main
 (:main
  0 0
  mem rsp -8 <- :myF_ret
  rdi <- 3
  call :myF 1
  :myF_ret
  v1 <- rax
  ...
 )
)

Your work
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

• Mapping 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
Label example

define :main ( ){
    :begin
    ...
    :end
    ...
}

• LL is "begin"
• LLG is "begin_global_"

:begin_global_begin

:begin_global_end
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
A compiler

Middle-end

IR

Back-end

... Instruction selection Register allocation

Machine code
Naive translation of an L3 function

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

Your work

(: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
    ...
}

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

Instruction selection depends on the context!
Instruction selection: it isn’t that easy

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

Your work

```
(:myF
  2 0
  p1 <- rdi
  p2 <- rsi
  v2 @ p2 p1 3
  ...
)
```

Instruction selection must satisfy all constraints of the target language!
Instruction selection: context

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

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

\[
v_1 \leftarrow p_1 \times 4
\]
\[
v_2 \leftarrow v_1 + p_2
\]
From L3 instructions to trees

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

\[
\begin{align*}
\text{VAR3} & \leftarrow \text{VAR1} \\
\text{VAR3} & \leftarrow \text{VAR2} \\
\text{VAR3} & \leftarrow \text{VAR1} \\
v1 & \leftarrow p1 \\
v3 & \leftarrow v2 \cdot v1
\end{align*}
\]
Example: tiles and tiling

\[ \begin{align*}
\text{VAR3} &\leftarrow \text{VAR1} \\
\text{VAR3} &\leftarrow \text{VAR3} \ast \text{VAR2}
\end{align*} \]

\[ \begin{align*}
\text{v1} &\leftarrow \text{v1} \ast \text{v2} \\
\text{v3} &\leftarrow \text{v1} \ast 5
\end{align*} \]
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 IR trees
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
• Usually want to pick large tiles: fewer instructions
• Instructions ≠ cycles: different instructions may take different cycles to execute
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
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

Total cost: 3

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

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