Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• L2

• The L2 compiler
Introduction to L2

• Like L1, but we have variables in L2!

Example of L1 programs

(:go
 (:go 0 1
   mem rsp 0 <- 5
   //code
   rdi <- mem rsp 0
   rdi <<= 1
   mem rsp 0 <- rdi
   return
 )
)

L2 programs

(:go
 (:go 0
   %myVar1 <- 5
   //code
   %myVar1 <<= 1
   return
 )
)

(:go
 (:go 0 0
   %myVar1 <- 5
   rdi <- %myVar1
call print 1
   return
 )
)

Variable is a powerful abstraction
Compilers have to deal with the complexity of implementing that abstraction
Variables in L2

- Variables (on top of registers)
- L2 variables are function local

```assembly
(:myF
  1
  %myVar <- 5
  rdi++
  mem rsp -8 <- :RET
  call :myF2 1
  :RET
  rdi <- %myVar
  call print 1
  return
)

(:myF2
  1
  %myVar <- rdi
  %myVar++
  return
)
```
\[ p ::= (\text{label } f^+) \]
\[ f ::= (\text{label } N N i^+) \]
\[ i ::= w \leftarrow s | w \leftarrow \text{mem } x \text{M} | \text{mem } x \text{M} \leftarrow s | \]
\[ w \text{ aop } t | w \text{ sop } s x | w \text{ sop } N | \text{mem } x \text{M} \leftarrow t | \text{mem } x \text{M} \leftarrow t | w \leftarrow \text{mem } x \text{M} | w \leftarrow \text{mem } x \text{M} | \]
\[ w \leftarrow t \text{cmp} t | \text{cjump} t \text{cmp} t \text{ label} | \text{label} | \text{goto} \text{ label} | \]
\[ \text{return} | \text{call } u \text{ N} | \text{call print } 1 | \text{call allocate } 2 | \text{call array-error } 2 | \]
\[ w \leftarrow t | w \leftarrow t | w \leftarrow @ w w E \]
\[ w ::= a | \text{rax} | \text{rbx} | \text{rbp} | \text{r10} | \text{r11} | \text{r12} | \text{r13} | \text{r14} | \text{r15} \]
\[ a ::= \text{rdi} | \text{rsi} | \text{rdx} | \text{s x} | \text{r8} | \text{r9} \]
\[ s ::= \text{rcx} \]
\[ \text{s} ::= t | \text{label} \]
\[ t ::= x | N \]
\[ u ::= w | \text{label} \]
\[ x ::= w | \text{rsp} \]
\[ \text{aop} ::= \leftarrow t | \leftarrow t | *= t | &*= t \]
\[ \text{sop} ::= <<= t | >>= t \]
\[ \text{cmp} ::= < t | \leq t | = \]
\[ E ::= 1 | 2 | 4 | 8 \]
\[ M ::= N \text{ times } 8 \]
\[ N ::= (+|-)? [1-9][0-9]* \]
\[ \text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]* \]
L2

\[ p ::= (label f^+) \]
\[ f ::= (label N f^+) \]
\[ i ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
   w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
   w <- t cmp t | cjump t cmp t label | label | goto label |
   return | call u N | call print 1 | call allocate 2 | call array-error 2 |
   w ++ | w -- | w @ w w E \]
\[ w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15 \]
\[ a ::= rdi | rsi | rdx | sx | r8 | r9 \]
\[ sx ::= rcx | var \]
\[ s ::= t | label \]
\[ t ::= x | N \]
\[ u ::= w | label \]
\[ x ::= w | rsp \]
\[ aop ::= += | -= | *= | &= \]
\[ sop ::= <<= | >>= \]
\[ cmp ::= < | <= | = \]
\[ E ::= 1 | 2 | 4 | 8 \]
\[ M ::= N times 8 \]
\[ N ::= (+|-)? [1-9][0-9]* \]
\[ var ::= %name \]
\[ label ::= sequence of chars matching :[a-zA-Z][a-zA-Z_0-9]* \]
A problem for L1 developers

• We want to print the last argument of :myF plus 1
• Is there any bug in our L1 program?
• We need to save r10
• Is there any bug in our L1 program?
A problem for L1 developers

- We want to print the last argument of :myF plus 1
- Is there any bug in our L1 program?
- We need to save r10
- Is there any bug in our L1 program?
The new L2 instruction

• It accesses stack-based arguments
  \( w \leftarrow \text{stack-arg } M \)

• (stack-arg 0) is always the last stack argument
• (stack-arg 8) is always the second to last argument
Final notes on L2

As for L1:

• The scope of labels is the program
• `cjump t cmp t label`  
  *Fall-through semantic*
• Values are encoded following the same rules of L1
• Same calling conventions of L1
  • No need to store the return address on the stack for calls to runtime functions
  • Return label must be just after the call
  • Same partition of caller/callee saved registers
• Same rules for memory heap allocation
Tests for homework 1

• Rewrite your L1 programs using the L2 language
  • To compile an L2 program:
    • Use the original framework, which is still available on Canvas
      • Not the one you have modified to implement your L1 compiler
    • cd L2
    • ./L2c L2_PROGRAM.L2
    • ./a.out

• Deadline: check Canvas
Outline

• L2

• The L2 compiler
A compiler

Character stream --- Source code (e.g., C)

Front-end

Middle-end

Back-end

Machine code (e.g., x86_64)
The L2 compiler (L2c)

- To build L2c:
  - translate an L2 program to an equivalent L1

- We need to map L2 variables to registers
  - Register allocation

- We need to translate the L2 instruction
  \( w \leftarrow \text{stack-arg} M \)
Optional: testing your L1 compiler with my L2 compiler

To compile an L2 program:
  • Compile your own L1 compiler: L1/bin/L1
  • Make sure you have my L2 compiler: L2/bin/L2
  • cd L2
  • ./L2c L2_PROGRAM.L2
  • ./a.out
Register allocation task

• Intra-procedural approach (most used)
  For each function \( f \)
  Map each variable of \( f \) to either a register or to a stack location
  (within locals in the L1 stack)

• Inter-procedural approach
  Map variables of functions in registers
  exploiting their caller-callee links
Task: From Variables to Registers

(:MyVeryImportantFunction

  %MyVar1 <- 2
  %MyVar2 <- 40
  %MyVar3 <- %MyVar1
  %MyVar3 += %MyVar2
  print %MyVar3

)

No overlapping
To register allocators: what are you doing?

(:MyVeryImportantFunction

| %MyVar1  <-  2
| %MyVar2  <-  40
| %MyVar3  <-  0
| %MyVar3  += %MyVar1
| %MyVar3  += %MyVar2
| print %MyVar3 |

Two naïve solutions for register allocation:
1. Spill all variables
2. Increase the #registers
A register allocator structure

Register allocator

Map heuristic

f without varX

Spill

Current f, varX

f without variables

f
Register Allocation

A. Spill all variables
B. Puzzle solving
C. Linear scan
D. Graph coloring
E. Integer linear programming