Intermediate Representation I: Control Structures

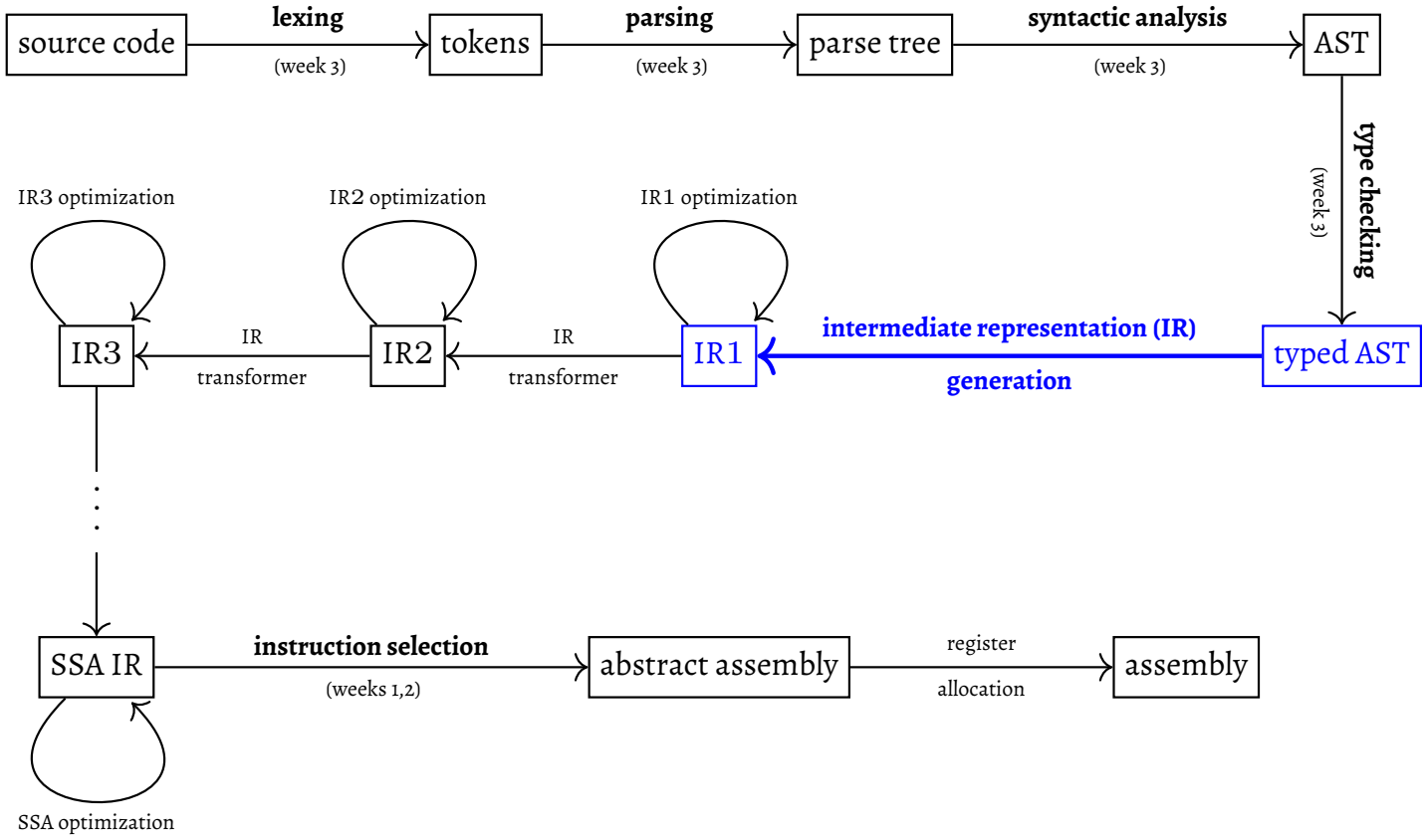
CSE 302 – Compilers – Week 4

Kaustuv Chaudhuri

Inria & École polytechnique

2019-09-30

Where in the compiler are we?



Summary of our Compiler

Things you have seen

- Define the grammar of the language
- Parse source code into an abstract syntax tree (AST)
- Convert an AST for sequential code (BX0) directly into machine instructions (instruction selection)

Summary of our Compiler

Things you have seen

- Define the grammar of the language
- Parse source code into an abstract syntax tree (AST)
- Convert an AST for sequential code (BX0) directly into machine instructions (instruction selection) by spilling all temporaries onto the stack

Summary of our Compiler

Things you have seen

- Define the grammar of the language
- Parse source code into an abstract syntax tree (AST)
- Convert an AST for sequential code (BX0) directly into machine instructions (instruction selection) by spilling all temporaries onto the stack

What more is necessary to compile BX1?

- Conditionals and loops
- Booleans and boolean expressions
- Comparisons

Summary of our Compiler

Things you have seen

- Define the grammar of the language
- Parse source code into an abstract syntax tree (AST)
- Convert an AST for sequential code (BX0) directly into machine instructions (instruction selection) by spilling all temporaries onto the stack

What more is necessary to compile BX1?

- Conditionals and loops
- Booleans and boolean expressions
- Comparisons

Common feature: jumps

Today's Agenda

1. Jumps and comparisons in AMD64
2. Register Transfer Language (RTL)
3. Compiling control structures to RTL
4. Code-generation for RTL

Jumps and Comparisons in AMD64

Labels and Instruction Pointer

- Pointers: (virtual) memory addresses as values
  Can be dereferenced to read/write memory
- Label: addresses of an instruction in compiled code
- Instruction pointer (`%rip`): contains the address of the *next* instruction, so setting `%rip` changes the code path
- Jump: specialized instructions to set `%rip`.
  - `jmp`: absolute jump
  - `je`, `jne`, `jle`, etc: conditional jumps

Example of Jumps and Conditionals

```
var x = 0, y = 1 : int64;
var z : int64;

while (x < 200) {
    print x;
    z = x;
    x = y;
    y = y + z;
}
```

```
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $24, %rsp
    movq $0, -8(%rbp)          # x = 0
    movq $1, -16(%rbp)         # y = 1
.L0:                           # while begin
    cmpq $199, -8(%rbp)
    jge .L1                    # 199 >= x ?
    movq -8(%rbp), %rdi
    callq bx1_print_int
    movq -8(%rbp), %r11
    movq %r11, -24(%rbp)       # z = x
    movq -16(%rbp), %r11
    movq %r11, -8(%rbp)        # x = y
    movq -24(%rbp), %r11
    addq %r11, -16(%rbp)       # y = y + z
    jmp .L0
.L1:                           # while end
    movq %rbp, %rsp
    popq %rbp
    xorq %rax, %rax
    retq
```

Comparison Instruction

```
cmpq      $42,   %rax
          src1   src2
          (reg/imm)  (reg/mem)
```

- Sets the status flags register (`%eflags`) based on src1 − src2
- Note: does not affect any of the other registers
- Overlays:
  - `cmpl  $42, %eax`
  - `cmpw  $42, %ax`
  - `cmpb  $42, %al`

Jump Instructions

```
jmp     .L42
jcnd    .L42
```

label

- Jumps unconditionally (`jmp`) or when a particular condition flag is set/unset based on an earlier `cmpq`
- Some conditional jumps: (note synonyms)

Instruction	Condition	wrt: [`cmpq src1, src2`]
`je`, `jz`	ZF=1	`src1 - src2 == 0`
`jne`, `jnz`	ZF=0	`src1 - src2 != 0`
`jle`, `jng`	ZF=1 or SF≠OF	`src1 - src2 <= 0`
`jl`, `jnge`	ZF=0 and SF≠OF	`src1 - src2 < 0`
`jge`, `jnl`	ZF=1 or SF≠OF	`src1 - src2 >= 0`
`jg`, `jnle`	ZF=0 and SF=OF	`src1 - src2 > 0`

Register Transfer Language

Intermediate Languages

- The AST retains much of the structure of the original program
  - Control structures
  - Variables
  - Aggregate types, abstract types, polymorphism, etc. (to come)
- An intermediate language eliminates most of these structures and abstractions
  - Only unstructured control (jumps, implicit or explicit)
  - Pseudo-registers of only primitive types + pointers
  - Simple family of instructions, close to assembly
  - Structure access is explicit (to come)
- Control Flow Graph (CFG): the purpose of all intermediate languages is to yield a graph representation of the flow of control through a program, which can be used for analysis and optimization in later phases

Register Transfer Language (RTL)

Our first intermediate language

- Register Transfer Language
  - Can use an infinite number of pseudo-registers (aka "*pseudos*")
    - Will be written `#1`, `#2`, etc.
    - Overlay suffixes: `q` (64-bit), `d` (32-bit), `w` (16-bit), `b` (8-bit)
    - Special pseudo `##` for "*don't care*"
    - Line comments will now start with `;`
  - Pervasive use of labels, written `L0`, `L1`, etc. (globally unique)
  - Each RTL instruction has:
    - A unique source label (aka "*in label*")
    - A list of successor labels (aka "*out label*")
  - No global relative ordering of instructions
  - For procedures: (to come)
    - Two unique labels: *enter*, *exit*
    - Certain pseudo-registers marked as *input* / *output*
    - Instructions know which procedure they are for

- Assembly:
  - Can only use a finite number of hardware registers
  - Has a fixed ordering of instructions

RTL Instructions by Example

(Inspired by, but not the same as, AMD64 assembly)

```
L42: binop  add,  #7q,  #10q  ⟶  L57
```

in label | instruction | opcode | arg1 | arg2 | out label

- All arguments must be registers (except for `move`)
- Some instructions have an additional opcode, which unites many instructions into the same category
- Must mention both in label and out label(s)
- Multiple out labels separated by `,`

RTL Instructions: Summary

RTL instruction	Description
`Li: move n, r1 ⟶ Lo`	move imm., `r1 = n`
`Li: copy r1, r2 ⟶ Lo`	copy regs, `r2 = r1`
`Li: unop op, r1 ⟶ Lo`	`r1 = op r1`
`Li: binop op, r1, r2 ⟶ Lo`	`r2 = r2 op r1`
`Li: ubranch op, r1 ⟶ Lo1, Lo2`	unary branch: `op r1`
`Li: bbranch, r1, r2 ⟶ Lo1, Lo2`	binary branch: `r1 op r2`
`Li: goto ⟶ Lo`	unconditional jump
`Li: call f(r1, ..., rn), r ⟶ Lo`	function call, result in `r`
`Li: return r`	return (note: no out label)

`n`	immediate (no $)	`r1, r2, ...`	pseudo-registers
`op`	opcode	`Li, Lo, ...`	labels

RTL: Example

```
var x = 0, y = 1 : int64;
var z : int64;

while (x < 200) {
    print x;
    z = x;
    x = y;
    y = y + z;
}
```

```
enter: L0
exit: L9
----
L0: move 0, #0q                    ⟶ L1
L1: move 1, #1q                    ⟶ L2
L2: move 200, #2q                  ⟶ L3
L3: bbranch jl, #0q, #2q           ⟶ L4, L8
L4: call bx1_print(#0q), ##        ⟶ L5
L5: copy #0q, #3q                  ⟶ L6
L6: copy #1q, #0q                  ⟶ L7
L7: binop add, #3q, #1q            ⟶ L3
L8: move 0, #0q                    ⟶ L9
L9: return #0q
```

RTL Generation

Generating RTL

(From the typed AST – for now)

- Basic idea: adapt maximal munch to deal with labels, control structures, new expression forms, etc.
  - Next few slides present the *top down* variant
- The BX1 AST is still simple enough that we can directly generate RTL
- Later in the course, the AST will first need to be *simplified* or elaborated before the RTL generation

RTL Generation: Integer Expressions

Simple cases

$L_i = \mathrm{RTL_i}\,(e, \mathrm{r}_d, L_o)$

- Given: e (an `int64` expression), r_d (a destination pseudo-register), L_o (an out-label)
- Returns: L_i (an in-label)
- Side-effect: emits RTL instructions

e	$\mathtt{Li} = \mathrm{RTL_i}\,(e, \mathtt{rd}, \mathtt{Lo})$	fresh
`42`	`Li: move 42, rd` $\longrightarrow$ `Lo`	`Li`
`ri`	`Li: copy ri, rd` $\longrightarrow$ `Lo`	`Li`
$e_1 + e_2$	$\mathtt{Li} = \mathrm{RTL_i}\,(e_1, \mathtt{rt}, \mathtt{L1})$ $\mathtt{L1} = \mathrm{RTL_i}\,(e_2, \mathtt{rd}, \mathtt{L2})$ `L2: binop addq, rt, rd` $\longrightarrow$ `Lo`	`L2, rt`

Read bottom to top

RTL Generation: Boolean Expressions

$L_i = \text{RTL}_\text{b}\ (e, L_t, L_f)$

- Given: e (a `bool` expression), L_t (an out-label for the `true` case), L_f (out-label for the `false` case)
- Returns: L_i (an in-label)
- Side-effect: emits RTL instructions

e	$\texttt{Li} = \text{RTL}_\text{b}\ (e, \texttt{Lt}, \texttt{Lf})$	fresh
`true`	Li is just Lt	–
`false`	Li is just Lf	–
! e_1	$\texttt{Li} = \text{RTL}_\text{b}\ (e_1, \texttt{Lf}, \texttt{Lt})$	–
e_1 && e_2	$\texttt{Li} = \text{RTL}_\text{b}\ (e_1, \texttt{L1}, \texttt{Lf})$ $\texttt{L1} = \text{RTL}_\text{b}\ (e_2, \texttt{Lt}, \texttt{Lf})$	–
e_1 || e_2	$\texttt{Li} = \text{RTL}_\text{b}\ (e_1, \texttt{Lt}, \texttt{L1})$ $\texttt{L1} = \text{RTL}_\text{b}\ (e_2, \texttt{Lt}, \texttt{Lf})$	–
$e_1 < e_2$	$\texttt{Li} = \text{RTL}_\text{i}\ (e_1, \texttt{r1}, \texttt{L1})$ $\texttt{L1} = \text{RTL}_\text{i}\ (e_2, \texttt{r2}, \texttt{L2})$ L2: bbranch jl, r1, r2 $\longrightarrow$ Lt, Lf	L2, r1, r2

RTL Generation: Boolean Comparison

When e_1, e_2 : `bool`, you can use the following equivalences:

$$(e_1 == e_2) \equiv (e_1 \;\&\&\; e_2) \;||\; !\;(e_1 \;||\; e_2)$$
$$(e_1 \;!= e_2) \equiv !\;(e_1 \;\&\&\; e_2) \;\&\&\; (e_1 \;||\; e_2)$$

Note: each subexpression e_1 and e_2 must only be computed once!

RTL Generation: Statements

$L_i = \mathrm{RTL}_s\,(s, L_o)$

- Given: s a statement, L_o an out-label
- Returns: L_i, an in-label
- Side-effect: emits RTL instructions

s	Li $= \mathrm{RTL}_s\,(s,$ Lo$)$	proviso
$x = e;$ (int64)	Li $= \mathrm{RTL}_i\,(e,$ r, Lo$)$	r $=$ lookup(x)
$x = e;$ (bool)	Li $= \mathrm{RTL}_b\,(e,$ Lt, Lf$)$ Lt: move 1, r $\longrightarrow$ Lo Lf: move 0, r $\longrightarrow$ Lo	r $=$ lookup(x) Lt, Lf fresh

Handle print similarly

RTL Generation: Blocks, Conditionals, Loops

s	$\texttt{Li} = \text{RTL}_\text{s}\,(s, \texttt{Lo})$	fresh
$\{\,\}$	Li is just Lo	–
$\{s_1\ s_2\ \cdots\ s_n\}$	$\texttt{Li} = \text{RTL}_\text{s}\,(s_1, \texttt{L2})$ $\texttt{L2} = \text{RTL}_\text{s}\,(s_2, \texttt{L3})$ $\vdots$ $\texttt{Ln} = \text{RTL}_\text{s}\,(s_n, \texttt{Lo})$	–
$\texttt{if}(e_c)\ s_t\ \texttt{else}\ s_f$	$\texttt{Li} = \text{RTL}_\text{b}\,(e_c, \texttt{Lt}, \texttt{Lf})$ $\texttt{Lt} = \text{RTL}_\text{s}\,(s_t, \texttt{Lo})$ $\texttt{Lf} = \text{RTL}_\text{s}\,(s_f, \texttt{Lo})$	–
$\texttt{while}(e_c)\ s_b$	$\texttt{Li} = \text{RTL}_\text{b}\,(e_c, \texttt{Lt}, \texttt{Lo})$ $\texttt{Lt} = \text{RTL}_\text{s}\,(s_b, \texttt{Lend})$ Lend: goto $\longrightarrow$ Li	Lend

RTL to Assembly

Stupid Algorithm (That Works)

Step 1/3

- For all non-jumping instructions, jump to out-labels.

```
L10: binop addq, #3, #4 ⟶ L20
```

```
.L10:
    ### TODO: load #4 into %rax
    ### TODO: load #3 into %rdx
    addq %rdx, %rax
    ### TODO: save %rax into #4
    jmp .L20
```

Stupid Algorithm (That Works)

Step 2/3

- For jumping instructions, jump to the false out-label

```
L10: ubranch jz, #3 ⟶ L20, L30
```

```
.L10:
    ### TODO: load #3 into %rax
    cmpq $0, %rax
    je .L20          # true case
    jmp .L30         # fall-through false case
```

Stupid Algorithm (That Works)

Step 3/3

- Line up all the generated instructions in some order (say top-to-bottom)
- Remove redundant `jmp`s.

```
    ### ...
    addq %esi, %eax
    movq %eax, -8(%rbp)
    jmp .L10
.L10:
    movq -16(%rbp), %eax
    cmpq $200, %eax
    ### ...
```

$\Rightarrow$

```
    ### ...
    addq %esi, %eax
    movq %eax, -8(%rbp)
# --- jmp removed ---
.L10:
    movq -16(%rbp), %eax
    cmpq $200, %eax
    ### ...
```

Stupid Algorithm (That Works)

Step 3/3

- Line up all the generated instructions in some order (say top-to-bottom)
- Remove redundant `jmp`s.

```
    ### ...
    addq %esi, %eax
    movq %eax, -8(%rbp)
    jmp .L10
.L10:
    movq -16(%rbp), %eax
    cmpq $200, %eax
    ### ...
```

$\Rightarrow$

```
    ### ...
    addq %esi, %eax
    movq %eax, -8(%rbp)
# --- jmp removed ---
.L10:
    movq -16(%rbp), %eax
    cmpq $200, %eax
    ### ...
```

- Warning! Don't remove labels (unless you can prove that nothing else jumps to that label)

BX and RTL: The Road Ahead

- Pointless to build a more complex algorithm *at this point* for linearizing RTL to AMD64
- Next step: RTL with explicits (ERTL):
  - Instructions with both pseudo-registers and machine registers
  - Break down complex instructions (e.g., `idivq`) into simpler instruction sequences
  - Coalesce linear sequences into basic blocks
- BX2 will add *functions* and *procedures*, so we will add to ERTL:
  - Input and output pseudo-registers
  - Calling conventions and stack frames
- We will then move on to Static Single Assignment (SSA) form

BX1 and Lab 3/Week 2

- At the end of Week 1 (this Thursday) you should:
  - Parse the BX1 syntax
  - Have written and submitted 5 example BX1 programs
    These programs will be given to everyone as challenges
  - Test your programs with the provided BX1 interpreter
- Week 2 tasks:
  - Implement the RTL instruction format
  - Transform BX1 AST to RTL
  - Transform RTL to AMD64
  - We will give you an RTL interpreter, which you can use to test your BX1 to RTL phase