15-411: Compiler Design

Spring 2026

Recitation 5: Calling Conventions, Typechecking, and Dataflow Analysis Solutions

20 February

Announcements

- Lab 3 Tests due on February 21st
- Written 3 due on Februrary 25th
- Lab 3 due before Spring Break, on February 28th
- Code reviews will be happening one week after Lab 3 is due. Take extra care to refine the style of your compiler and add documentation to each function, file, and a global README explaining overall architecture and design decisions.

The L3 language adds support for function calls, type definitions, and header files with C interoperability. In this recitation, we'll discuss some of the implications of adding these features and how your compiler should deal with them.

Caller- and Callee-Saved Registers

In Lab 3, your compiler's code-generation and register allocation phases will need to distinguish between *callee-saved* and *caller-saved* registers:

- The values stored in **callee-saved registers** must be preserved across function calls. This means that your function must save and restore any callee-saved registers that it modifies.
- The values stored in **caller-saved registers** may be modified by any function call, so your compiler cannot assume that they will retain their values after calling a function. If you need those values to be preserved, you must save and restore them before and after the function call.

To avoid having callee-saved registers occupy a very long live range during register allocation, we can handle them separately. Prioritize allocating caller-saved registers; if they are insufficient, we assign callee-saved registers before we resort to spilling, but we make sure to save them to the stack at the beginning of a function and restore them at the end. This is more efficient than always saving and restoring all callee-saved registers.

Function	64-bit	32-bit	16-bit	8-bit
Return Value	**%rax**	**%eax**	%ax	%al
Callee saved	**%rbx**	**%ebx**	%bx	%bl
4th Argument	**%rcx**	**%ecx**	%cx	%cl
3rd Argument	**%rdx**	**%edx**	%dx	%dl
2nd Argument	**%rsi**	**%esi**	%si	%sil
1st Argument	**%rdi**	**%edi**	%di	%dil
Callee saved	**%rbp**	**%ebp**	%bp	%bpl
Stack Pointer	**%rsp**	**%esp**	%sp	%spl
5th Argument	**%r8**	**%r8d**	%r8w	%r8b
6th Argument	**%r9**	**%r9d**	%r9w	%r9b
Caller saved	**%r10**	**%r10d**	%r10w	%r10b
Caller saved	**%r11**	**%r11d**	%r11w	%r11b
Callee saved	**%r12**	**%r12d**	%r12w	%r12b
Callee saved	**%r13**	**%r13d**	%r13w	%r13b
Callee saved	**%r14**	**%r14d**	%r14w	%r14b
Callee saved	**%r15**	**%r15d**	%r15w	%r15b

Tracing Function Calls in x86-64

In Lab 3, your compiler must conform to the standard C calling conventions for x86-64. As a reminder, this means that:

- The first six arguments to a function should be stored in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` (respectively).
- The remaining arguments should be placed on the stack. The seventh argument should be stored at the address `%rsp`, the eighth at `%rsp + 8`, etc.
- The return value of a function should be stored in `%rax`.
- The use of `%rbp` as a base pointer is not required (but you may find that using it simplifies your compiler's logic significantly). LLVM uses the base pointer, but GCC does not.

Another interesting observation: unlike in C, every function in C0 (and thus in L3) has a fixed stack size that can be computed at compile time. This observation allows you to make your compiler's stack-handling much simpler than if you were unable to determine the stack size beforehand.

Checkpoint 0

Draw a stack diagram for the following L3 program at the point when execution reaches line 4. Assume that %rbp is being used as a base pointer.

```
1 int f(int we, int dont, int care, int about, int these, int args, int a, int b) {
2    // assume that x is spilled on the stack
3    int x = a + b;
4    return 2 * x;
5 }
6
7 int main() {
8    return f(0,0,0,0,0,0,3,5);
9 }
```

Solution:

Value	Pointers
Return address of _main()	
Previous %rbp	
b; Arg. 8 of f()	
a; Arg. 7 of f()	
Return address of f()	
main's %rbp	← %rbp
x	← %rsp

Checkpoint 1

Using your stack diagram, convert the program to x86-64 assembly following the standard calling conventions. Remember to use the 64-bit and 32-bit versions of the registers appropriately and that stack grows downward!

Solution:

```
_c0_f:
  push %rbp
  movq %rsp, %rbp

  # reserve space for x, can be 16 to keep rsp 16 byte aligned
  subq $8, %rsp

  # int x = a + b; and spill x to the stack
  movl 16(%rbp), %eax
  addl 24(%rbp), %eax
  movl %eax, (%rsp)

  # return 2 * x
  movl (%rsp), %eax
  imull $2, %eax
  addq $8, %rsp
  pop %rbp
  ret

  _c0_main:
```

```
push %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $0, %edi
movl $0, %esi
movl $0, %edx
movl $0, %ecx
movl $0, %r8d
movl $0, %r9d
movl $3, (%rsp)
movl $5, 8(%rsp)
call _c0_f
addq $16, %rsp
pop %rbp
ret
```

Checkpoint 2

At the instruction `call _c0_f`, according to x86-64 calling convention, which registers are used, and which registers are defined/clobbered?

Solution: Recall from lecture that a call uses the argument registers (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`) and the arguments passed via memory at the stack pointer (`k(%rsp)`).

A call defines `%rax` and clobbers all caller-saved registers (`%r10`, `%r11` and the argument registers as well).

In practice, wee only need to save caller-saved registers that hold values that are live across the call, which is exactly what liveness analysis tells us.

Typechecking with Functions

Typechecking becomes more interesting when we add functions.

Checkpoint 3

Consider the following code:

```
int add(int x, int y) { return x + y; }

int main() {
  bool b = true;
  return add(b, 3);
}
```

What happens here?

Solution: Since `add` has type `add : (int, int) -> int` but the function call `add(b, 3)` has parameter types `(bool, int)`, so `b:bool` needs `int`. Hence the program is ill-typed and should be rejected by the typechecker.

Checkpoint 4

We also add header functions in Lab 3, and extend the behavior analogously so if `header.h` has:

```
  int mystery(int);
```

and `main.l3` has:

```
  int mystery(char x) {
    return 411;
  }

  int main() {
    return mystery(true);
  }
```

What happens?

Solution: There are two errors:

The header introduces `mystery : (char) -> int` into the global typing environment.

The typechecker will fail when it typechecks `int mystery(char x) { ... }`, since `mystery` is of type `mystery : (int) -> int`.

We can, however, change the function definition to `int mystery (int x) { return 411; }`. This is fine because we can redeclare functions as long as the declaration signature matches the existing signature for the function.

If we do so, then the function call `mystery(true)` passes in a bool, so the program would still be ill-typed.

Dataflow Analysis

Dataflow analysis is a framework that allows us to prove facts about a program based on all possible execution paths to an instruction by propagating along the control-flow graph. It helps us evaluate whether or not an optimization is *legal*, even if it doesn't always guarantee that it's *beneficial*.

Recall from lecture the following definitions:

- A definition d of a variable v **reaches** point u if there exists a path of control flow edges from d to u that does not contain another definition of v. We store these in use-def chains.
- A variable x is **live-out** of a stmt s if x can be used along some path starting at s, otherwise x is dead. From liveness, we can construct **def-use chains**, a list of all uses of a definition for each definition of a variable x.
- An expression $x + y$ is **available** at statement S if $x + y$ is computed along every path from the start to S and neither x nor y is modified after the last evaluation of $x + y$.
- An expression is **very busy at point p** if on every path from p, e is evaluated before the value of e is changed.

We will now do a short example of each of these and the respective optimizations they enable.

Checkpoint 5 - Reaching Definitions

Recall that for a statement s with ID d that defines variable v:

$$Gen[s] = \{d\}, \qquad Kill[s] = defs(v) - \{d\}.$$

We also have the following dataflow equations:

$$IN[b] = \bigcup_{p \in pred(b)} OUT[p], \qquad OUT[b] = Gen[b] \cup (IN[b] - Kill[b]).$$

This should remind you of the liveness equations. Now, consider the following program:

```
void f(int c) {
  1: x <- 5
  2: if (c == 0) goto A else goto B
A:
  3: x <- 5
  4: goto C
B:
  5: y <- 2
  6: goto C
C:
  7: z <- x + 1
}
```

(a) Compute $Gen[s]$ and $Kill[s]$ for statements 1 and 3.

(b) Which definitions of x reach statement 7?

(c) Is it legal to replace x with constant 5 at statement 7?

Solution:

(a) Definitions of x occur at 1 and 3.

- For stmt 1: $gen[1] = \{1\}$, $kill[1] = \{3\}$.
- For stmt 3: $gen[3] = \{3\}$, $kill[3] = \{1\}$.

(b) At statement 7, control may arrive via A or B.

- Along A: def from line 3 reaches 7.
- Along B: def from line 1 reaches 7.

Thus reaching defs of x at 7 are $\{1, 3\}$.

(c) Since both reaching definitions assign the constant 5, it is legal to replace x with 5 at statement 7. This lets us do simple constant and copy propagation in our programs.

Checkpoint 6 - Def-Use Chains

We can construct def-use chains from liveness analysis. A variable x is live-out of a stmt s if x can be used along some path starting at s, otherwise x is dead. Consider the following program:

```
1: a <- 1
2: b <- a + 2
3: d <- 4
4: c <- d + b
5: e <- b + c
6: f <- c / 0
7: return c
```

(a) For each definition, list its uses (construct the def-use chains).

(b) Which definitions are dead?

Solution:

(a)

$$a \mapsto \{2\} \quad b \mapsto \{4, 5\} \quad c \mapsto \{5, 6, 7\} \quad d \mapsto \{4\} \quad e \mapsto \{\} \quad f \mapsto \{\}$$

(b) Only e is dead. f is not dead because there are side effects due to dividing by zero. This lets us remove instruction 5 and do a pass of dead code analysis (DCA).

Checkpoint 7 - Available Expressions

An expression e is available at S if:

- e is computed on every path to S,
- and none of its operands are modified afterward.

Recall to compute available expressions, we let

$$Gen[b] = \text{if } b \text{ evals expr } e \text{ and doesn't define variables used in } e$$

$$Kill[b] = \text{if } b \text{ assigns to } x, \text{ then all exprs using } x \text{ are killed}$$

$$Out[b] = (In[b] - Kill[b]) \cup Gen[b]$$

$$In[b] = \bigcap_{p \in pred(b)} out[p]$$

Consider the following program:

```
void f(int c) {
  1: a <- 2
  2: b <- 3
  3: t <- a * b
  4: s <- t + b
  5: if (c == 0) goto A else goto B
A:
  6: a <- 7
  7: goto C
B:
  8: noop
  9: goto C
C:
  10: u <- a * b
  11: v <- t + b
}
```

Is $a * b$ available at statement 10? Is $t + b$ available at statement 11?

Can we replace either of these two instructions with `u <- t` or `v <- s`?

Solution: If we go through A: a is modified at 5 before reaching 9.

If we go through B: nothing is modified.

Note that $a * b$ was computed at 3 and $t + b$ was computed at 4.

Since availability requires the expression to hold on *every* path, $a * b$ is not available at 10. However, $t + b$ is available at 11.

Thus, replacing line 10 with `u <- t` is not legal, but it is legal to replace line 11 with `v <- s`. This lets us perform common subexpression elimination (CSE).

Checkpoint 8 - Very Busy Expressions

For the sake of example, we omit the equation, but it is similar to the previous three equations. See lecture slides for more detail. Consider the following program:

```
void f(int c, int x, int y) {
  1: if (c == 0) goto A else goto B
A:
  2: t1 <- x + y
  3: goto C
B:
  4: t2 <- x + y
  5: goto C
C:
  6: z <- x + y
  7: return z
}
```

Is $x + y$ very busy at the start of the program? What optimization could this enable?

Solution:

If we go through A: $x + y$ is evaluated at 2 before any modification.

If we go through B: $x + y$ is evaluated at 4 before any modification.

Thus $x + y$ is very busy at the start of f.

Hence, we can safely hoist the computation of of $x + y$ to the beginning of the function and reuse it. This allows us to perform the code motion optimization.

Tips and Hints for Lab3

- **Header Files in L3:** Unlike in C, header files in L3 (and above) are only used to declare types and external functions. If a function is declared in a header file, then it may not be defined in the program – it is declared as *external*. External functions are defined in C source files, which are linked together with the assembly produced by your compiler.
- **RBP:** You are not required to use %rbp as a base pointer, so you are allowed to treat it like a normal callee-saved register in your compiler.
- **Code Review:** Code Review happens one week after Lab3 is due. So if you haven't polished the style of your compiler and added a README describing the design of various passes of your compiler, now would be a good time to start. We are looking for good coding style and comments, modular design, and that both of you are familiar with all components of the implementation.