Compiling C with Clang by examples

$$C \xrightarrow{\text{Clang}} x86$$

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

October 27, 2015

Contents

Structure of the module

Parsing ✓

- Progression from: Language + Logic, Models of Computation
- abstract machines, formal, "mathy"

Compiling C with Clang

- Progression from: Computer Systems + Architecture, C/C++
- not so formal, by example, x86 machine code

Implementing functional languages

- Progression from: functional programming
- builds on abstract machines and C stack

Example

C code

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

x86 generated by Clang

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```

The assembly code does not look much like the source code.
What happened to variables?
What happened to types?

These are open source lectures and notes

The LaTeX source is in
`https://github.com/hayo-thielecke/clang-lectures`
`c-clang.tex` is for my slides
`c-clang-notes.tex` is for collaborative note taking.

Aims and overview

- We will see some typical C code compiled to x86 assembly by LLVM/Clang
- Emphasise general principles used in almost all compilers
- Use Clang on C and x86 for example and concreteness
- What Clang does, not details of how it does it internally
- Enough to compile some C code by hand line by line
- C language features $\mapsto$ sequence of assembly instructions + addresses
- Various language features on top of vanilla functions
- Optimizations

Clang and LLVM, the bestest and mostest compiler

Clang is the bestest C/C++ compiler
`http://clang.llvm.org`
LLVM is the mostest compiler infrastructure
`http://llvm.org`
Apple uses it
`https://developer.apple.com/xcode/`
Many projects, for example:
Emscripten: An LLVM to JavaScript Compiler
Rust: "a safe, concurrent, practical language" (as per blurb)
A not too technical intro to LLVM:
`http://www.aosabook.org/en/llvm.html`

Using Clang

Please do experiments yourself for seeing how LLVM/Clang compiles C.
Clang comes with XCode on OS X.
If you do not have LLVM on your computer:
ssh into a lab machine and type
module load llvm/3.3
To compile, type
clang -S test.c
Then the assembly code will be in test.s
Function frodo will be labelled frodo: in test.s
For optimization, use
clang -S -O3 test.c

Target architecture for Clang output

We will only need a tiny subset of assembly.
Quite readable.
Instruction we will need:

```
mov push pop call ret jmp add mul test be lea
```

The call instruction pushes the current instruction pointer onto the stack as the return address
ret pops the return address from the stack and makes it the new instruction pointer
A nice target architecture should have lots of general-purpose registers with indexed addressing.
Like RISC, but x86 is getting there in the 64-bit architecture

Assembly generated by clang is x86 in AT&T syntax

mov syntax is target-last:

`mov` x y is like y = x ;

r prefix on registers means 64 bit register

movq etc: q suffix means quadword = 64 bits

% register

$ constant

`%rbp` = base pointer = frame pointer in general terminology

`%rsp` = stack pointer, push and pop use it

indexed addressing `-24(%rbp)`

Typical C code to compile

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

Parameters/arguments:
x and y
Local/automatic variables
a and b

More precisely, x and y are *formal* parameters.
In a call f(1,2), 1 and 2 are the *actual* parameters.
We will use the words "parameter" and "argument" interchangeably.

Two big ideas in compiling functions

stack $\leftrightarrow$ recursion

compare: parsing stack

many abstract and not so abstract machines use stacks

including JVM

In C: one stack frame per function call

Names $\rightarrow$ indices

Names can be compiled into indices, discovered many times

In C: variables become small integers to be added to the base pointer

Stack frame details

The details differ between architectures (e.g., x86, ARM, SPARC)

Ingredients of stack frames, in various order, some may be missing:

return address

parameters

local vars

saved frame pointer

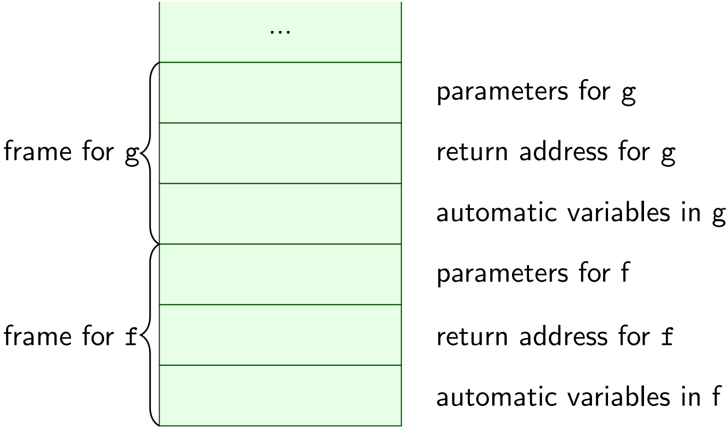
caller or callee saved registers

static link (in Pascal and Algol, but not in C)

this pointer for member functions (in C++)

A traditional stack layout (but not Clang)

Convention: we draw the stack growing downwards on the page.
Suppose function `g` calls function `f`.



There may be more in the frame, e.g. saved registers

What about recursive functions?

Consider the standard example of recursion:

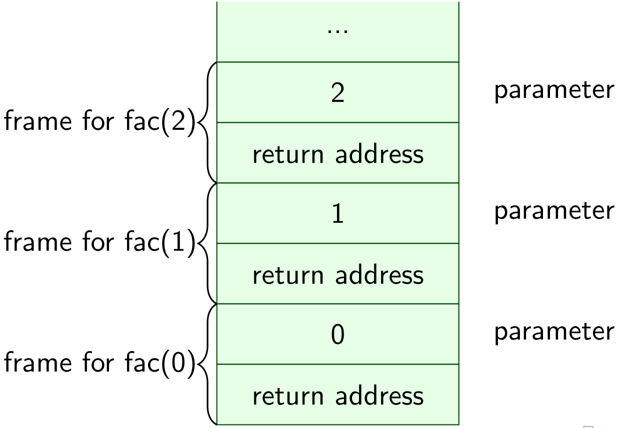
```
long factorial(long n)
{
  if(n == 0)
    return 1;
  else
    return factorial(n - 1) * n;
}
```

Call stack: one frame per function call

Recursion example: fac(n) calls fac(n - 1). Each recursive call gets a smaller parameter.

The return address points into the code segments, not the stack or heap.

What are the return addresses?



Return address example

```
long factorial(long n)
{
  if(n == 0)
    return 1;
  else
    return factorial(n - 1) * n;
}
```

The return address is a pointer to the compiled code. The returned value is returned into the hole ◯ position in the last statement,

```
return ◯ * n;
```

Thus when the function returns, 1 is plugged into the hole, then 2, then 6, . . .
The return address represents a continuation.

Calling conventions and stack frame layout

The calling convention differs between compilers and architectures
Old school:
push arguments onto stack, then do a call instruction (which pushes return address)
Modern architectures have many registers
$\Rightarrow$ pass arguments in registers when possible; Clang does this
Some RISC architectures put return address into a link register
more exotic: SPARC has register windows for parameter passing

Stack frame in clang C calling convention on x86

Clang passes parameters in registers `rdi`, `rds`, ...
The parameters also have a slot in the frame

return addr	
old bp	← frame/base pointer
parameter n	
⋮	
parameter 1	
local var	
⋮	
local var	← stack pointer, if used

Clang function idiom

```
http://llvm.org/docs/LangRef.html#calling-conventions

f:
pushq %rbp
movq %rsp, %rbp
    ... body of function f
popq %rbp
ret
```

parameters are passed in registers rdi, rsi
return value is passed in register rax

Computing the index in the frame

Simple in principle:
walk over the syntax tree and keep track of declarations
The declarations tell us the size: long x means x needs 8 bytes
That is why C has type declarations in the first place

```
long f(int x, int y) // put y at -8 and x at -16
{
    int a;    // put a at -24
    int b;    // put b at -32
    a = x;    // now we know where a and x are
              // relative to rbp
}
```

Exercise: what happens if we also have char and float declarations?

Clang stack frame example

```
long f(int x, int y) // put y at -8 and x at -16
{
    int a;    // put a at -24
    int b;    // put b at -32
    ...
}
```

return addr	
old bp	← base pointer
y	← bp - 8
x	← bp - 16
a	← bp - 24
b	← bp - 32

Compiled with clang -S

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

$$
\begin{array}{lcl}
y & \mapsto & \mathtt{rdi} \\
x & \mapsto & \mathtt{rsi} \\
y & \mapsto & \mathtt{rbp} - 8 \\
x & \mapsto & \mathtt{rbp} - 16 \\
a & \mapsto & \mathtt{rbp} - 24 \\
b & \mapsto & \mathtt{rbp} - 32
\end{array}
$$

```
f:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq -8(%rbp), %rsi
addq $42, %rsi
movq %rsi, -24(%rbp)
movq -16(%rbp), %rsi
addq $23, %rsi
movq %rsi, -32(%rbp)
movq -24(%rbp), %rsi
imulq -32(%rbp), %rsi
movq %rsi, %rax
popq %rbp
ret
```

Optimization: compiled with clang -S -O3

```c
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```

Many arguments

Some passed on the stack, not in registers. These have positive indices. Why?

```
long a(long x1, long x2,
long x3, long x4, long x5,
long x6, long x7, long x8)
{
  return x1 + x7 + x8;
}
```

```
a:
addq
8(%rsp), %rdi
addq
16(%rsp), %rdi
movq %rdi, %rax
ret
```