Functions and stack frames

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

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

October 25, 2015

Contents

Structure of the module

Parsing ✓

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

C and call stack

- 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

Aims and overview

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

Clang and LLVM, the bestest and mostest

Clang C/C++ compiler
`http://clang.llvm.org`
Compiler Infrastructure
`http://llvm.org`
`http://www.aosabook.org/en/llvm.html`
Apple `https://developer.apple.com/xcode/`
Many projects, for example:
Emscripten: An LLVM to JavaScript Compiler
Rust: a safe, concurrent, practical language

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.

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

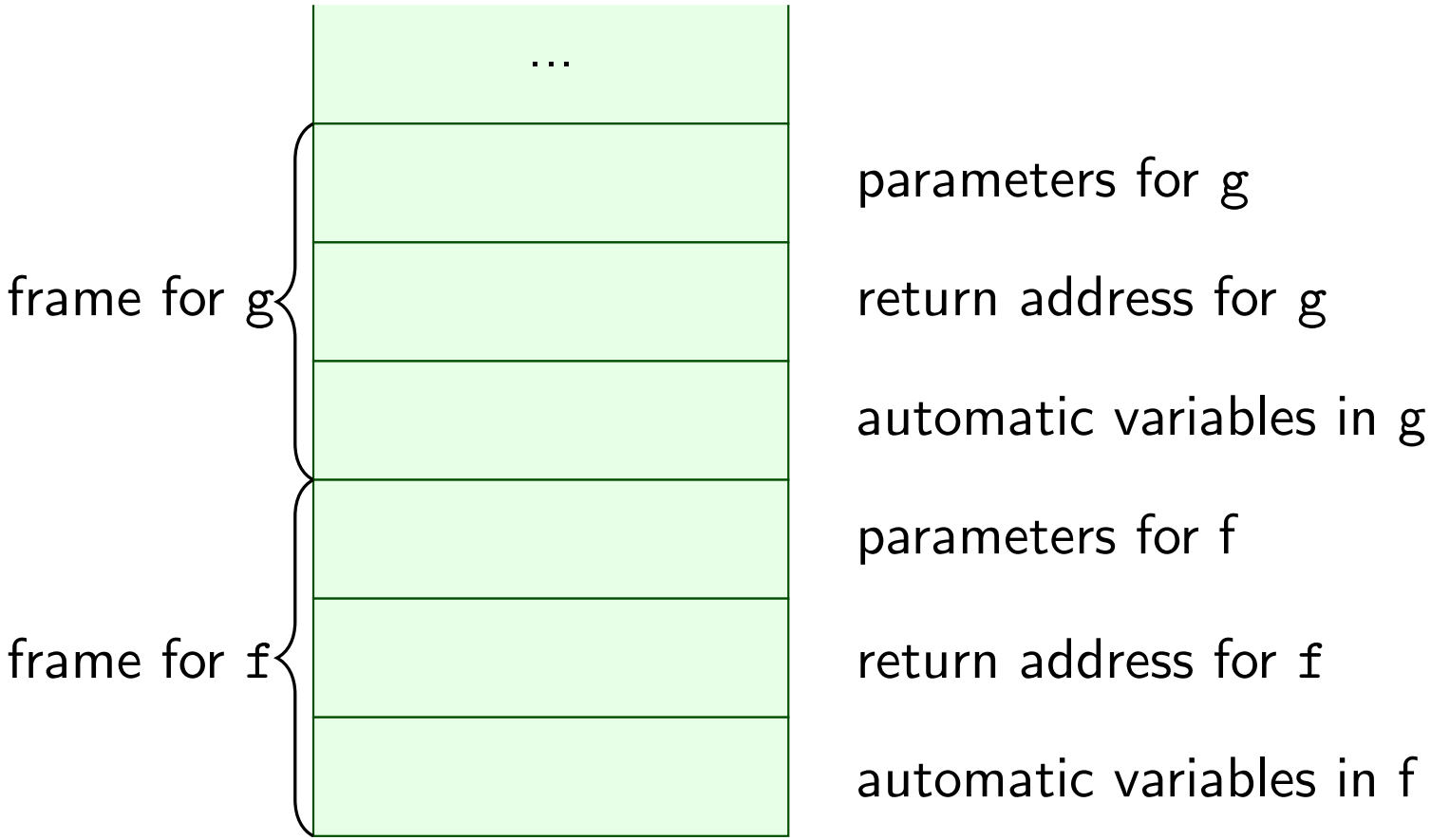
Names $\rightarrow$ indices

Names can be compiled into indices, discovered many times

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

Call stack: used by C at run time for function calls

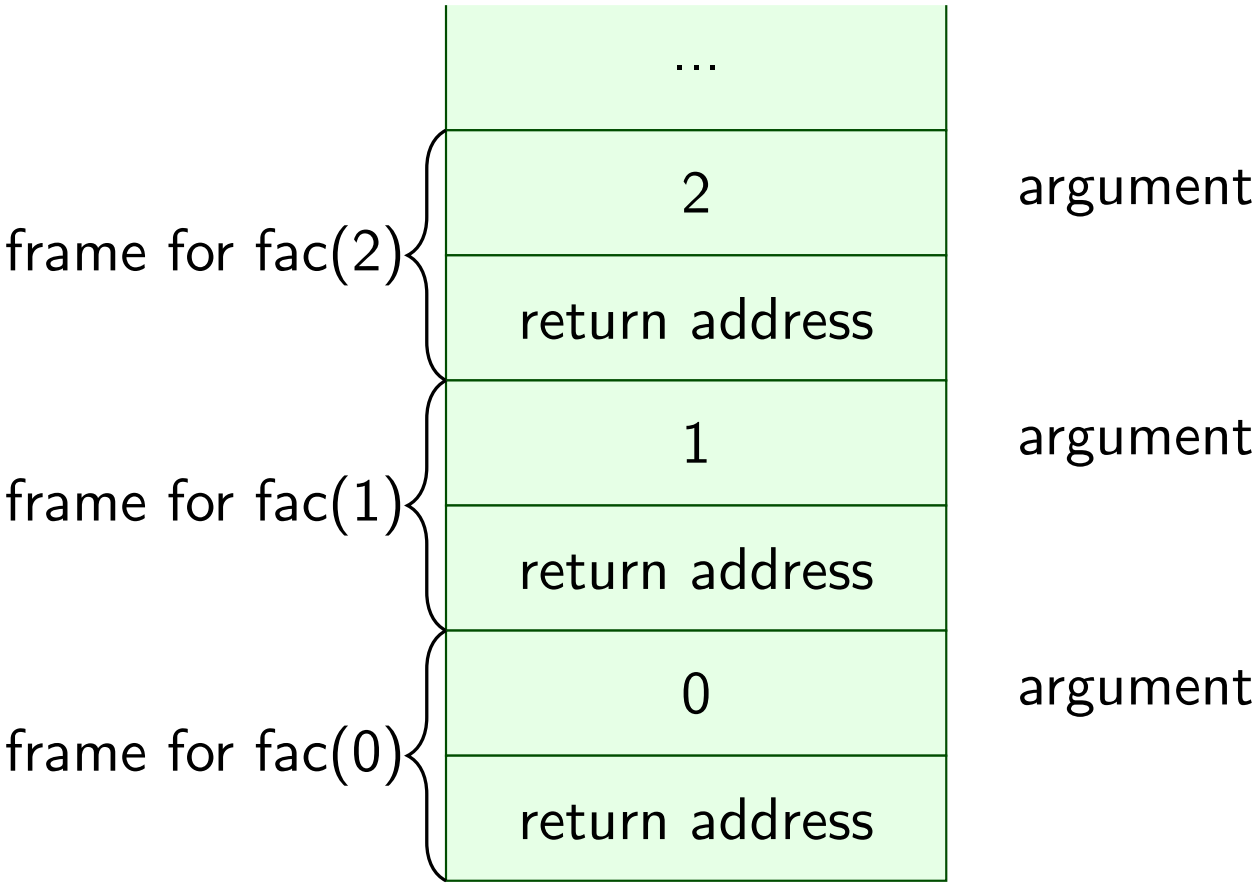
Convention: we draw the stack growing downwards on the page.
Suppose function g calls function `f`.



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

Call stack: one frame per function call

Recursion example: fac(n) calls fac(n - 1)



Target architecture

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
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

x86 in AT&T syntax

mov syntax is target last
`mov` x y is like y = x ;
Assembly generated by clang version 3.3
r prefix on registers means 64 bit register
movq etc: q suffix means quadword = 64 bits
% register
$ constant
indexed addressing `-24(%rbp)`

Clang function idiom

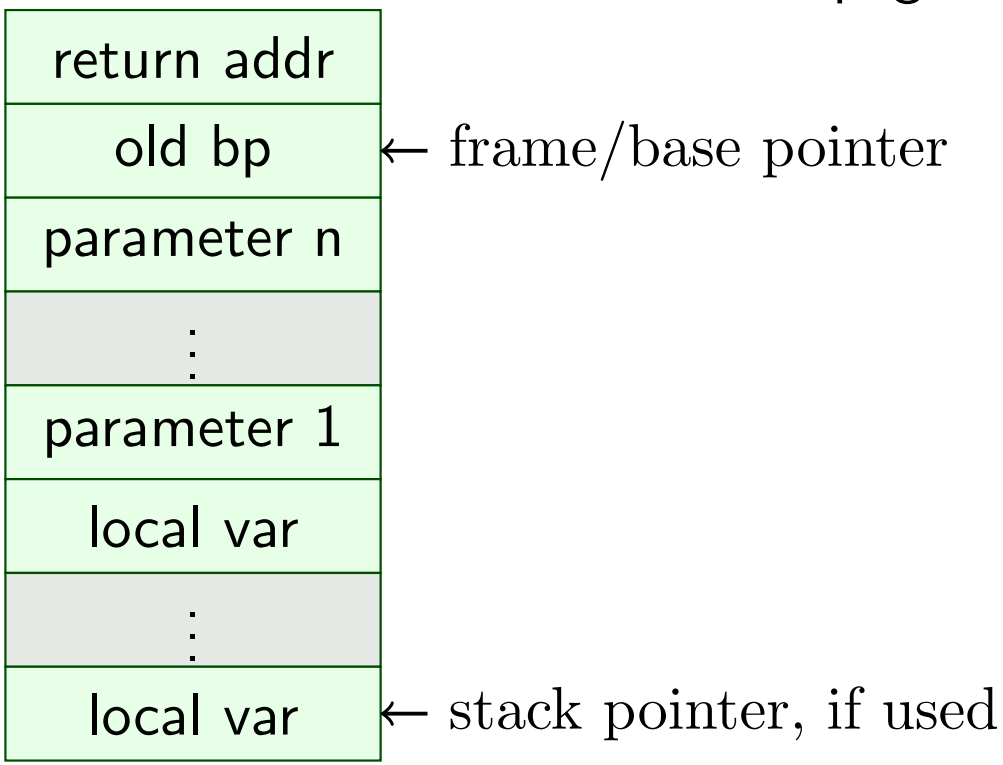
```
http://llvm.org/docs/LangRef.html#calling-conventions

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

Stack frame in clang C calling convention on x86

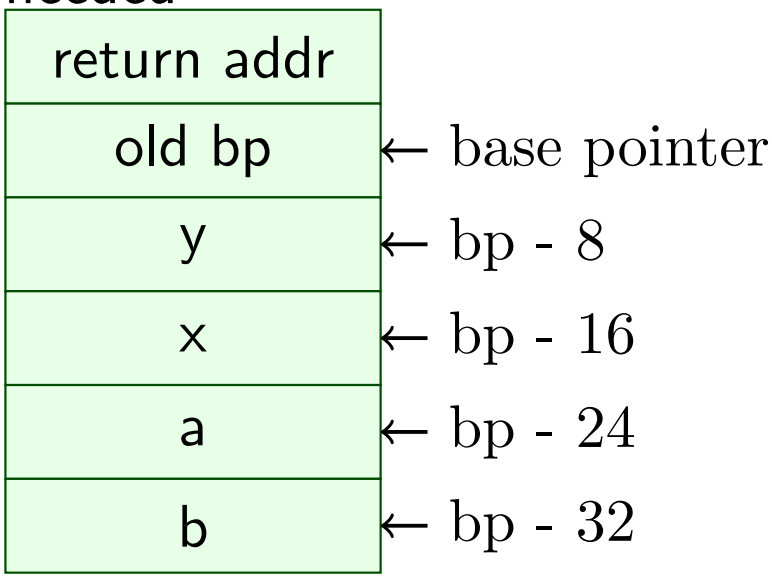
The stack grows down on the page

lower addresses are lower on the page



Clang stack frame example

The stack grows down on the page
lower addresses are lower on the page
parameters are passed in registers, may be saved into frame if needed



Compiled with clang -S

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

y$\mapsto$ rbp-8
x$\mapsto$ rbp-16
a$\mapsto$ rbp-24
b$\mapsto$ rbp-32

```
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
```

Compiled with clang -S -O3

```
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
```