Lecture Notes on Calling Conventions

15-411: Compiler Design
Frank Pfenning

Lecture 10

1 Introduction

In Lab 3 you will be adding functions to the arithmetic language with loops and conditionals. Compiling functions creates some new issues in the front end and the back end of the compiler. In the front end, we need to make sure functions are called with the right number of arguments, and arguments of the right type. In the back end, we need to create assembly code that respects the *calling conventions* of the machine architecture. Strict adherence to the calling conventions is crucial so that your code can interoperate with library routines, and the environment can call functions that you define.

Calling conventions are rather machine-specific and often quite arcane. You must carefully read the Section 3.2 of the AMD64 ABI [**?**], available at `http://www.x86-64.org/documentation/abi.pdf`. Examples and additional information is provided in Section 6 of a handout on x86-64 Machine-Level Programming by Bryant & O'Hallaron.

2 IR Trees

We have already seen in Lecture 9 that function calls should take pure arguments in order to easily guarantee the left-to-right evaluation order prescribed by our language semantics. Moreover, they should be lifted to the level of commands rather than remain embedded inside expressions because functions may have side-effects.

3 Low-Level Intermediate Language

In the low level intermediate language of quads that we have used so far in this course, it is convenient to add a new form of instruction

$$d \leftarrow f(s_1, \ldots, s_n)$$

where each s_i is a source operand and d is a destination operand.

The generic $\mathsf{def}(l, x)$, $\mathsf{use}(l, x)$ and $\mathsf{succ}(l, l')$ predicates are easily defined, assuming for simplicity that source and destinations are all temps.

$$\frac{l : d \leftarrow f(s_1, \ldots, s_n)}{\begin{array}{l}\mathsf{def}(l, d) \\ \mathsf{use}(l, s_i) \quad (1 \leq i \leq n) \\ \mathsf{succ}(l, l+1)\end{array}} \; J_8$$

Unfortunately, this is overly simplistic, because calling conventions prescribe the use of certain fixed registers for passing arguments and receiving results, so we will have to extend the above rule further.

4 x86-64 Calling Conventions

In x86-64, the first six arguments are passed in registers, the remaining arguments are passed on the stack. The result is returned in a specific return register `%rax`. These conventions do not count floating point arguments and results, which are passed in the dedicated floating point registers `%xmm0` to `%xmm7` and on the stack only if there are more than eight floating point parameters. Fortunately, our language has only integers at the moment, so you do not have to worry about the conventions for floating point numbers.

On the x86, stack frames were required to have a frame pointer `%ebp` (base pointer) which had to be saved and restored with each function call. It provided a reliable pointer to the beginning of a stack frame for easy calculation of frame offsets to handle references to arguments and local variables. It also allowed tools such as `gdb` to print backtraces of the stack. On the x86-64, this information is maintained elsewhere and a frame pointer is no longer required.

The general organization of stack frames at the time a procedure is called, will be as follows.

Position	Contents	Frame
...	...	Caller
16(%rsp)	argument 8	
8(%rsp)	argument 7	
(%rsp)	return address	

Note that all arguments take 8 bytes of space on the stack, even if the type of argument would indicate that only 4 bytes need to be passed.

The function that is called, the *callee*, should set up its stack frame, reserving space for local variables, spilled temps that could not be assigned to registers, and arguments passed to functions it calls in turn. We recommend calculating the total space needed and then decrementing the stack pointer `%rsp` by the appropriate amount. By changing the stack pointer only once, at the beginning, references to parameters and local variables remain constant throughout the function's execution. The stack then looks as follows, where the size of the callee's stack frame is n.

Position	Contents	Frame
...	...	Caller
n + 16(%rsp)	argument 8	
n + 8(%rsp)	argument 7	
n + 0(%rsp)	return address	
	local variables	Callee
	...	
	argument build area	
	for function calls	
	...	
(%rsp)	end of frame	
	128 bytes	red zone

Note that `%rsp` should be aligned $0 \mod 16$ before another function is called, and may be assumed to be aligned $8 \mod 16$ on function entry. This happens because the `call` instruction saves the 64-bit return address on the stack.

The area below the stack pointer is called the *red zone* and may be used by the callee as temporary storage for data that is not needed across function calls or even to build arguments to be used before a function call. The ABI states that the red zone "shall not be modified by signal or interrupt handlers." This can be tricky, however, because, for example, Linux kernel

code will not respect the red zone and overwrite this area. We therefore suggest not using the red zone.

5 Typical Calling Sequence

If we have 6 or fewer arguments, a typical calling sequence for 32-bit arguments with an instruction

$$d \leftarrow f(s_1, s_2, s_3)$$

will have the following form:

```
movl s3, %edx
movl s2, %esi
movl s1, %edi
call f
movl %eax, d
```

First we move the temps into the appropriate argument registers, then we call the function f (represented by a symbolic label), and then we move the result register into the desired destination.

This organization, perhaps just before register allocation, has the advantage that the live ranges of fixed registers (called *pre-colored nodes* in register allocation) is minimized. This is important to avoid potential conflict. We have already applied a similar technique in the implementation of div and mod operations, which expect their arguments in fixed registers.

We can now see a problem with our previous calculation of def and use information: the above sequence to actually implement the function call will overwrite the argument registers `%edx`, `%esi`, `%edi` as well as the result register `%eax` (the lower 32bits of the return register `%rax`)! In fact, any of the argument registers, the result register, as well as `%r10` (temporary register for passing static function chain pointers) and `%r11` (temporary register) may not be preserved across function call and therefore have to be considered to be *defined* by the call. If we represent this in the low-level intermediate language, we would *add* to the rule R_8 the following rule R'_8:

$$\frac{\begin{array}{l} l : d \leftarrow f(s_1, \ldots, s_n) \\ r \in \{\texttt{\%rax}, \texttt{\%rdi}, \texttt{\%rsi}, \texttt{\%rdx}, \texttt{\%rcx}, \texttt{\%r8}, \texttt{\%r9}, \texttt{\%r10}, \texttt{\%r11}\} \end{array}}{\mathsf{def}(l, r)} \; R'_8$$

Here we assume that register aliasing is handled correctly, that is, the register allocator understands that, for example, `%eax` constitutes the lower 32 bits of `%rax`.

Now if a temp t (except for d) is live after the function call, we have to add a edge connecting t with any of the fixed registers noted above, since the value of those registers are not preserved across the function call.

One more note: if it is possible that the function f is a function accepting a variable number of arguments, some additional considerations apply. For example, the low 8 bits of `%rax`, called `%al` hold the number of floating point arguments passed to the function. One therefore sometimes sees `xorl %eax, %eax` before a function call to define zero variable arguments.

6 Callee-Save Registers

The typical calling sequence above takes care of treating caller-save registers correctly. But what about callee-save registers, namely `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14` and `%r15`? In compiling a function we are required that the generated code preserves all the callee-save registers.

The standard approach is to save those that are needed onto the stack in the function prologue and restore them from the stack in the function epilogue, just before returning. Of course, saving and restoring them all is safe, but may be overkill for small functions that do not require many registers.

One way to let register allocation do the job for us, is to remember the rule that fixed registers should have short live ranges. Callee-save registers contradict that rule, because they are essentially live throughout the body of a function. In order to avoid this, we can move the contents of the callee-save registers into temps at the beginning of a function and then move them back at the end. If it turns out these temps are spilled, then they will be saved onto the stack. If not, they may be moved from one register to another and then back at the end. In order to avoid this move, we can apply register coalescing after register allocation. Register coalescing is briefly described in Section 8 of Lecture 3. Another optimization that can eliminate register-to-register moves is copy propagation, covered in a later lecture. Register coalescing is essentially copy propagation for registers.

The general shape of the code for a function f before register allocation

would be

f :
    $t_1 \leftarrow$ %rbx
    $t_2 \leftarrow$ %rbp
    ...
    function body
    ...
    %rbp $\leftarrow t_2$
    %rbx $\leftarrow t_1$
    ret

One complication with this approach is that we need to be sure to spill the full 64-bit registers, while registers holding 32-bit integer values might be saved and restored (or directly used as operands) using only 32 bits. Looking ahead, we see that we will need both 32 bit and 64 bit registers and spill slots in the next lab, so we might decide to introduce this complication now. Or we can still treat callee-save registers specially and switch over to a more uniform treatment in the next lab.

References