Assignment 3
Calling Conventions and Optimizations

15-411: Compiler Design
Frank Pfenning
Robbie Harwood, Sri Raghavan, Max Serrano

Due Thursday, October 10, 2013 (1:30pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Thursday, October 10. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1: Calling Conventions (15 points)

(a) The function on the next page is written in x86-64 assembly (with some syntactic liberties), and it is a bit *too* concerned about preserving the calling function's register values. Without modifying the function's "work" section, rewrite the function to remove as many of the "prologue" and "epilogue" instructions as you safely can. You should also give the reasons why those instructions are unnecessary, and why the remaining ones are necessary. You should also reduce the stack frame size to the amount that is actually necessary for your new function.

(b) In addition to saving and restoring registers it doesn't need to, the program is violating the x86-64 calling conventions. Explain how.

(c) There is nothing on an x86-64 processor actually enforcing the calling conventions, and there are a couple reasons why you might be tempted to forgo them. In some situations you could optimize away a few instructions by ignoring them, or you might simply want to use the easier to remember rule that `%r8` through `%r15` are the callee-save registers. However, conventions usually exist for a good reason. What would be the negative consequences of ignoring calling conventions in your compiler? Are there any circumstances (in your compiler, or in general) when it *would* be okay to ignore calling conventions?

```
// Prologue
subq $1337, %rsp
movq %rbx, 108(%rsp)
movq %rcx, 116(%rsp)
movq %rdx, 124(%rsp)
movq %rsi, 132(%rsp)
movq %rdi, 140(%rsp)
movq %rbp, 148(%rsp)
movq %r8, 156(%rsp)
movq %r9, 164(%rsp)
movq %r10, 172(%rsp)
movq %r11, 180(%rsp)
movq %r12, 188(%rsp)
movq %r13, 196(%rsp)
movq %r14, 204(%rsp)
movq %r15, 212(%rsp)

// Work
movq $8, %rax
movq %rsi, %rbp
addq %rdi, %rbp
movq %rbx, 0(%rsp)
movq %rbp, 8(%rsp)
addq 0(%rsp), %r8
subq 8(%rsp), %r8
addq %r8, %rax

// Epilogue
movq 108(%rsp), %rbx
movq 116(%rsp), %rcx
movq 124(%rsp), %rdx
movq 132(%rsp), %rsi
movq 140(%rsp), %rdi
movq 148(%rsp), %rbp
movq 156(%rsp), %r8
movq 164(%rsp), %r9
movq 172(%rsp), %r10
movq 180(%rsp), %r11
movq 188(%rsp), %r12
movq 196(%rsp), %r13
movq 204(%rsp), %r14
movq 212(%rsp), %r15
addq $1337, %rsp
ret
```

Problem 2: Tail Call Optimization (20 points)

Tail call optimization is an optimization which can be applied wherever a function f makes a call to a function g and then returns either nothing or directly the result of g. (An important special case of this is *tail recursion*, where f and g are the same function.) Because f does nothing with its local variables after the call to g, it is safe to have g overwrite the contents of f's stack frame instead of creating a new one, and additionally save a somewhat costly `ret` operation. In this problem you'll walk through tail-recursion optimization of an exponential function using an accumulator a.

```
int powacc(int b, int e, int a)
//@requires e >= 0;
{
  if (e == 0) return a;
  else return powacc(b, e-1, a*b);
}

int pow(int b, int e)
//@requires e >= 0;
{
  return powacc(b, e, 1);
}
```

First, we translate to abstract assembly with parameters, not yet employing calling conventions.

```
powacc$(b, e, a)$ :
  if $(e == 0)$ goto done
  $t_0 \leftarrow e - 1$
  $t_1 \leftarrow a * b$
  $t_2 \leftarrow$ powacc$(b, t_0, t_1)$
  return $t_2$
done :
  return $a$

pow$(b, e)$ :
  $t_0 \leftarrow$ powacc$(b, e, 1)$
  return $t_0$
```

(a) Now insert standard code templates for calling sequences as described in Lecture 11, using abstract notations for the registers (for example, arg_1 is the first argument register, and res_0 the result register). Assume that we do not use any callee-save registers. Functions are no longer parameterized, function calls and returns no longer have explicit arguments; instead we are using argument and return registers. Be careful not to optimize (yet)!

(b) Next we want to apply some simple optimizations to obtain the code pattern

```
call f
ret
```

which we can then replaced by

```
goto f
```

The only optimizations you are allowed to perform are:

- *Copy and constant propagation*: given a move $t \leftarrow s$, you may replace an occurrence of t by s in code reachable from it. This is correct only under some conditions. You do not have to verify any explicit conditions, but you should make sure your particular application of copy or constant propagation is correct.
- *Self-move elimination*: delete an instruction $t \leftarrow t$.
- *Dead-code elimination*: an instruction $t \leftarrow s$ or $t \leftarrow s_1 \oplus s_2$ for a pure (effect-free) operator $\oplus$ can be deleted, if t is not live in its successor.

Indicate for each line that you optimize, which optimization(s) were necessary to achieve it.

(c) Perform register allocation for the remaining temps and show the resulting x86-64 program. Your register allocation may be as clever as you like (you can freely pick registers for temps, as long as they create no conflict in the interference graph). Briefly compare the resulting code to what you might obtain from an iterative program computing with a while loop.

Problem 3: Inlining (25 points)

In this problem we explore inlining, which is another important optimization related to function calls. In brief, we replace a function call with the body of the function, taking care to rename temps to avoid naming conflicts.

Consider the following functions, where catalan(n) computes the nth Catalan number.

```
int next(int n, int c)
//@requires n >= 0;
{
  return 2*(2*n+1)*c/(n+2);
}

int catalan(int n)
//@requires n >= 0;
{
  int i = 0;
  int c = 1;
  while (i < n) {
    c = next(i,c);
    i = i + 1;
  }
  return c;
}
```

(a) Here is the 3-address form of the catalan function

```
catalan(n):
  i <- 0
  c <- 1
L0:
  if (i >= n) goto L1
  c <- next(i,c)
  i <- i + 1
  goto L0
L1:
  return c
```

Transform this code by making calling conventions explicit. Stay in the 3-address form, but use argument and result registers appropriately.

(b) If you were to apply register allocation now, would you assign n, i, and c to caller-save or callee-save registers? Briefly explain.

(c) Compile the auxiliary function next to 3-address form, taking care that the evaluation order for subexpressions is *strictly from left to right*.

(d) Make parameter passing in next explicit, using appropriate argument and result registers.

(e) Inline the code for next inside catalan. Do not apply any optimizations (yet)!

(f) Apply applicable optimizations as in Problem 2. Which instructions, and how many of each, have been eliminated by inlining and subsequent optimization? What is the impact (if any) on subsequent register allocation? Compare to your answer in (b).