Compilers Optimization Capstone Abstract

Constant Propagation:

The goal of constant propagation is to replace expressions involving only constants with their computed values. This can eliminate unnecessary computations and improve the efficiency of the code. First, we analyze the expressions in the bodies of definitions and the program and match whether the expression is included in the list of expressions we are required to optimize (ie. Add1, Plus, Let, etc). The fold function used to do this is also recursive and it "folds" again using the subexpression of the expression until we conclude whether the expression can be evaluated to a constant or not. If this is possible, the constant is returned and replaces the original expression. In my code, I used symtab to keep track of the Let variables and whether they could be collapsed into a constant inside later subexpressions.

Inlining:

Before implementing inlining, I was required to first "uniquify" all variables such that all variables had globally unique names. This process ensures that variables are not to be confused with each other if the same variable is used within the scope of another variable. The helper function gensym was used to generate new names for each variable. The uniquify optimization itself was similar to the constant propagation implementation in that I looped through each args of definitions and bodies of both definitions and the program to recursively traverse through each expression, such as Plus and Ifs, then find and add each variable to symtab to keep track of their new names. I then replaced the original defns and body with a new defns and bodies at the end.

For the actual inlining, the compiler identifies functions that are suitable for inlining by looping through each and checking that it is both a leaf and fits the heuristics that we have set. For my implementation, the function size multiplied by the number of static calls had to be less than 28, which passed all tests as intended. Once we have identified such a function call, the compiler replaces the call with the entire body of the called function. This includes substituting the actual arguments at the call site. This is also done by recursively traversing through the expressions of both the definition bodies and the program body.

Peephole:

I implemented four different types of peephole optimizations in my code. Two of them implemented the suggestions given by the homework handout. I used pattern matching to identify when the assembly started with a Mov R, V1 and Mov R, V2. Then, I checked that V2 is not dependent on R (it is not a memory offset of R). If it is, I simply return the original directions. If not, I combine the two Mov directions into a single Mov R, V2. Similarly, I also had a pattern matching case for Mov R, S and Mov S, R, which would simply return Mov R, S instead. My third optimization is very similar to the first two, except that instead of Mov, I am now using Add and Sub directives. If a number is moved into a register,

which is then used for an Add or a Sub operation, I combine the two and directly use the number to perform the addition or subtraction instead of using the register. My last optimization also simplifies two Mov directions into one and is a combination of my other optimizations. In this optimization, if an immutable value is moved into a register and the register is moved to a memory offset (ie. QWORD [-16 + rsp]), the value is directly moved to the memory offset instead of the register. None of these changes modify the functionality of the code but only reduce the number of operations for efficiency.

The peephole function is also recursive to continue the peephole optimization down the assembly code after a peephole optimization has occurred (or not).

Below is an example test that I added whose assembly output changes depending on whether we use the peephole optimization or not.

Example: examples/peephole.lisp - (print (+ 1 (+ 1 (read-num))))

Before adding peephole:

```
lisp_entry:
    mov rax, 20
    mov QWORD [-8 + rsp], rax
    mov rax, 12
    mov QWORD [-16 + rsp], rax
    mov QWORD [-24 + rsp], rdi
    add rsp, -24
    call read_num
```

After adding peephole:

```
lisp_entry:
    mov QWORD [-8 + rsp], 20
    mov QWORD [-16 + rsp], 12
    mov QWORD [-24 + rsp], rdi
    add rsp, -24
    call read_num
```

The assembly output is produced with inlining and constant propagation, and the second output has all optimizations. Because one of our peephole optimizations combines consecutive Mov directions into a single one if a value is moved into a register, and that register is moved into a MemOffset, we have two fewer assembly directives than before, making it more efficient.

Optimization:

After implementing the above three optimizations, I was able to run all benchmark cases and observe the effect of adding each optimization on the time. For example, the test cases below exemplify a visible decrease in time taken as optimizations are added to the pass. The improvement is more drastic in some than others depending on the operations involved. For example, my implementation of peephole

may not be as effective on an expression like (+ 1 (+ 1 (+ 3 5))) due to there not being enough redundant movements between registers, though constant propagation would certainly help downsize the operation. However, adding many read-nums in between the expression would make peephole a much more efficient optimization than a constant propagation. This is one of the reasons why for some of the tests below, the decrease in time from just inlining to constant propagation and inlining was significantly larger than the decrease from const prop and inlining to all opimizations (including peephole).

```
constant-propagation2-ylu168.lisp,All optimizations,4210701.400006656
constant-propagation2-ylu168.lisp,Constant propagation and inlining,4405667.399987578
constant-propagation2-ylu168.lisp,Inlining,4689747.999873362
```

```
peephole-basic-tji6.out.lisp,All optimizations,666639.7999651963
peephole-basic-tji6.out.lisp,Constant propagation and inlining,696494.7002416011
peephole-basic-tji6.out.lisp,Inlining,926659.1001505731
```

Because there is not a single “correct” way of optimization for all cases, the time did slightly increase as optimizations were added for a few test cases, especially with inlining. For example, the time actually increased from no optimizations to inlining in the example below. This could be because our inlining increased the amount of computation more than it optimized by inlining. Changing the heuristics of our inlining optimizations could help mitigate this.

```
constant-folding.lisp,Inlining,722618.3006423526
constant-folding.lisp,No optimizations,720083.6000265554
```