CS164 Programming Languages and Compilers **Spring 2024**

Written Assignment 6

Assigned: April 14 **Due:** April 25 at 11:59pm

Instructions: This assignment asks you to prepare written answers to questions on local optimization, global optimization, register allocation and dataflow analysis. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

Please write your name, email address, and discussion section on your homework. ***Please start each question on a new page. All written assignments must be submitted as a PDF via Gradescope: https://gradescope.com.*** Instructions for how to submit assignments to Gradescope can be found at the following links: `https://gradescope.com/get_started#student-submission`

1. Consider the following code:

```
p := a + b
r := 1 * b
s := a * b
x := 0 * r
y := a + b
u := x + 2
w := y * y
v := u + w
z := p ** 2
```

   Assume that only v and z are live at the exit of this block. Apply the following optimization in order, one by one, to this basic block. Show the result of each transformation and the final optimized code.

   (a) algebraic simplification

   (b) common sub-expression elimination

   (c) copy propagation

   (d) constant folding/propagation

   (e) dead code elimination

   (f) When you've completed part (e), the resulting program will still not be optimal. What optimization can you apply to optimize the result of (e) further? What does this tell you about how one should apply local optimizations?

2. Optimize the following code, using global constant propagation (do NOT perform constant folding as that is not part of the global constant propagation algorithm). Draw the CFG and the associated constant propogation dataflow, then write out the optimized code.

```
L0:
  a := 1
  b := 2
  if y < 0 goto L2

L1:
  c := a + b
  a := a + 1
  b := b + 1
  jump L3

L2:
  c := 3
  a := a + c

L3:
  b := a * b
  a := c
  if y > 0 goto L2
```

3. Consider the following source-level optimizations proposed for the ChocoPy language:

   (a) Replace '`False and e`' with '`False`', where `e` is any ChocoPy expression of type `bool`.

   (b) Replace '`e or True`' with '`True`', where `e` is any ChocoPy expression of type `bool`.

   (c) Replace '`x - 0`' with '`x`', where `x` is any ChocoPy variable of type `int`.

   (d) Replace '`0 % x`' with '`0`', where `x` is any ChocoPy variable of type `int`.

   (e) Replace '`x + 1 >= x`' with '`True`', where `x` is any ChocoPy variable of type `int`.

   It is not important how the optimizations are implemented; you can think of them as search-and-replace over ASTs of semantically valid and well-typed programs.

   For each optimization, answer the following sub-questions:

   (i) Is there any ChocoPy program, whose output will be different if you execute it with and without this optimization? You can use the compiler on chocopy.org to observe a program's output. If your answer is 'yes', then provide a short program (in unoptimized form) that demonstrates this difference, and list its output before and after optimization.

   (ii) Is this optimization sound? Answer 'yes' or 'no'. A compiler optimization is sound if its application does not change the behavior of the program under the language's operational semantics.

4. Consider the following program.

```
L0:
  d := e + b
  a := 4
  b := 2
  e := b + d
  if e > 5 goto L2

L1:
  if e > 0 goto L0
  jump L3

L2:
  c := e + d
  e := 2 + d
  e := d + a

L3:
  e := e - a
  a := e
  return e
```

(a) Construct a control-flow graph for the program

(b) Perform Liveness Analysis on the program (use the method described in lecture on 11/19). Do not do any copy propagation or constant folding but eliminate dead code based on your liveness analysis.

(c) Construct a RIG (Register Interference Graph) for the original program above, without the optimization done in part b. (Hint: a, b, c, d, e should be the nodes in your graph.)

(d) Find a coloring scheme with 3 colors for the RIG using the heuristic introduced in class. Break possible ties in alphabetical order. You will find that you get stuck at a certain point and will need to spill a certain register into memory. Decide this node alphabetically.

(e) Construct a CFG for the spilled program.

5. In this problem, we will construct a dataflow analysis for an extended form of the intermediate language described in lecture. Recall that we defined an intermediate language whose instructions were:

```
instr:
| id := id op id         (binary operation)
| id := op id            (unary operation)
| id := id               (copy)
| push id                (push)
| id := pop              (pop)
| if id relop id goto L  (conditional jump)
| L:                     (label)
| jump L                 (unconditional jump)
```

Now let us add the following new kinds of instructions:

```
instr:
...
| id := new T
| id := call id, m
```

The instruction `id := new T` constructs a new instance of class `T`, and assigns it to `id`; the equivalent in ChocoPy code would be `id = T()`. The instruction `id1 := call id2, m` calls the method `m` on the class object stored in `id2`, and assigns it to `id1`; it is the equivalent of `id1 = id2.m(...)`. (Note that the arguments for `m` are located on the stack, so the intermediate-language instruction itself does not need to refer to the arguments.)

As mentioned in lecture, it is sometimes possible to statically determine which exact instance of `m` will be called by any `id := call id, m` instruction. For example, in the following ChocoPy code, it is easy to see that the `f` method called is necessarily the version defined in `B`, instead of the version defined in `A`:

```
class A(object):
    def f(self: "A"):
        print("A:f")
class B(A):
    def f(self: "B"):
        print("B:F")
obj : A = None
result : object = None
obj = B()
result = obj.f()
```

The resulting IL (intermediate language) representation of the top-level code could be:

```
obj := new B
push obj
result := call obj, f
```

From this IL we may immediately verify the observation that `B::f` is called instead of `A::f`. Since `obj` is assigned a `B` value, and there are no intervening assignments to `obj`, then the last line `result := call obj, f` necessarily calls the version of `f` defined in class `B`. In other words, we know that the last line will always call the `B` version of `f` since we know that just before the last IL instruction, the dynamic type of `obj` is in the set {`B`}, and since this set is a singleton, this just means that we know exactly what the dynamic type of `obj` is.

Of course we may not be able to statically determine in all cases which instance of a method will be called at a particular program point. Consider the following IL, which uses the same class definitions as before; also assume that the register `i` has some unspecified integer value:

```
    if i == 0 goto i_is_zero
    jump i_is_not_zero
i_is_not_zero:
    obj := new A
    jump make_the_call
i_is_zero:
    obj := new B
    jump make_the_call
make_the_call:
    result := call obj, f
```

In this IL example, the instance of `f` called (i.e. `A::f` or `B::f`) depends on the value if `i`, which may not be statically determinable; if `i == 0` then we call the `B` version of `f`, and if `i != 0` then we call the `A` version of `f`. We may view this as related to the following data: we know that just before the last IL instruction, the dynamic type of `obj` is in the set $\{\texttt{A},\texttt{B}\}$; since this set is *not* a singleton, then we don't know exactly which version of `f` will be called.

We now want to define a dataflow analysis which will provide (conservative) data on what the possible dynamic types are for any variable at any program point. For this analysis, our *value domain*, i.e. the collection of abstract values we can assign to any variable at any program point, will be the collection of *any set of classes.* For example, $\{\texttt{A},\texttt{B}\}$ is an element of our value domain, as is $\{\texttt{B}\}$ and also $\emptyset$, the empty set (which may represent the possible dynamic types of an unassigned variable).

For any IL instruction p and register x, let $T_{in}(x,p)$ be a conservative approximation for the dynamic types of x just before p, and let $T_{out}(x,p)$ be a conservative approximation for the dynamic types of x just after p. This means that if x can have dynamic type T just before the instruction p, then we must have $T \in T_{in}(x,p)$; and similarly, if x can have dynamic type T just after the instruction p, then we must have $T \in T_{out}(x,p)$.

(a) Should this analysis be a forward or backward dataflow analysis?

(b) For an IL program which has (only) the classes `A`, `B`, and `C` defined, where `B` and `C` are subclasses of `A`, draw the resulting value-domain lattice. Make sure that $\emptyset$ is at the bottom and that $\{\texttt{A},\texttt{B},\texttt{C}\}$ is at the top.

(c) Now we must define the transfer functions of this analysis.

   i. Suppose that $p_1,\ldots,p_n$ are the direct predecessors of the IL instruction s. What should $T_{in}(x,s)$ be in terms of $T_{out}(x,p_1),\ldots,T_{out}(x,p_n)$? If you use the `lub` operator of our value domain, then you must define it. (It has a relatively simple form.)

   ii. Suppose that p is the IL instruction `x := y`. What should $T_{out}(r,p)$ be in terms of $T_{in}(r,p)$, for any register r? (Hint: split the analysis into two cases: $r = \texttt{x}$ or $r \neq \texttt{x}$.)

   iii. Suppose that p is the IL instruction `x := new T`. What should $T_{out}(r,p)$ be in terms of $T_{in}(r,p)$, for any register r? (Hint: again, split the analysis into cases.)

   iv. Suppose that p is the IL instruction `x := call y, m`, and suppose that the dynamic type of the return value of m is known to be U. What should $T_{out}(r,p)$ be in terms of $T_{in}(r,p)$, for any register r?

(d) Here are the transfer functions for the other IL instructions:

- $T_{out}(\texttt{x},\texttt{x := y op z}) = \{t_{\texttt{op}}(T_{\texttt{y}},T_{\texttt{z}})\}$, where $T_{\texttt{y}}$ is the statically-known type of `y` (and $T_{\texttt{z}}$ the statically-known type of `z`), and $t_{\texttt{op}}(T,U)$ for two primitive types T and U is defined to be the resulting type of evaluating `op` on a value of type T and a value of type U. For example, $t_+(\texttt{int}, \texttt{int}) = \texttt{int}$, while $t_+(\texttt{str}, \texttt{str}) = \texttt{str}$ and $t_{==}(\texttt{object}, \texttt{[object]}) = \texttt{bool}$ since the `==` operator on any two values always returns a `bool`. We also have $T_{out}(r, \texttt{x := y op z}) = T_{in}(r, \texttt{x := y op z})$ for $r \neq \texttt{x}$.

- Similarly define $T_{out}(\texttt{x}, \texttt{x := op y}) = \{t_{\texttt{op}}(T_{\texttt{y}})\}$, where now $t_{\texttt{op}}(T)$ is defined to be the resulting type of evaluating `op` on a value of type T. As before, $T_{out}(r, \texttt{x := op y}) = T_{in}(r, \texttt{x := op y})$ for $r \neq \texttt{x}$.
- For p an instruction `push id` or `if id relop id goto L`, we simply have $T_{out}(x, p) = T_{in}(x, p)$ for any register x.

Now consider the following full analysis algorithm:

```
Initialize Tin(x,p) and Tout(x,p) to {} for every register x and instruction p.
while there is any Tin(x,p) or Tout(x,p) not satisfying our transfer functions:
    update Tin(x,p) and Tout(x,p) using our transfer functions
```

Explain why this algorithm is guaranteed to terminate. (Hint: make the claim that any time we update $T_{in}(x, p)$ or $T_{out}(x, p)$, the new value is at least as high in our value domain lattice as is the previous value, that is, if v was the original value and v' the new value, then $v \subseteq v'$. Then argue that our value domain lattice has finite height, and explain why these two statements imply that the algorithm terminates.)

(e) Now let's see how we can actually use the information produced by our dataflow analysis for the purpose of optimization. Suppose that after running our analysis, we know that for an IL instruction `x := call y, m`, we have $T_{in}(\texttt{y}, \texttt{x := call y, m}) = C$ for some collection of classes C. Under what conditions on C can we replace the dynamic dispatch of this call by a dispatch to a statically determined target function? (Hint: you may assume that you statically know whether or not m is overwritten, or merely inherited, by each class $T \in C$.)