CMSC 631, Fall 2011
Written Exercises

Due Tuesday, December 13, 11:59:59pm

Each problem on this homework is worth a varying number of points. To get credit on this homework, you must answer at least 70 points' worth of questions; your final score will be out of the the total possible points of the problems you attempt.

This homework is *optional*. If you do not turn it in, your final homework score will be computed from the results of the first four homeworks. If you do it, your final homework score will be computed from all five homeworks assuming the end result is better than the first four homeworks alone. In short, if this homework, graded, does not improve your homework score then it will not factor into your grade.

Nevertheless, use this homework (all problems) as a study guide for the final exam.

1. Let A be a lattice, with order $\leq$. Define $A \to A$ to be the set of all functions from A to A, and define $f \leq' g$ iff $f(x) \leq g(x)$ for all $x \in A$.

   - (6 points) Show that $A \to A$ with order $\leq'$ is also a lattice. That is, show that for all $f, g \in A \to A$, $f \sqcup g$ and $f \sqcap g$ always exist.
   - (4 points) Suppose lattice A has height h and that A is finite with n elements. What is the height of the lattice $(A \to A, \leq')$? (When counting height, count "edges" rather than "nodes," e.g., if A were the lattice $\{a, b\}$ with $a < b$, then its height would be 1.)

2. In class we talked about how an analysis is *conservative* if it models the behavior of the program in a way that is safe. As it turns out, "safe" is in the eye of the beholder.

   When performing dataflow analysis to estimate the following properties, determine whether too-large or too-small estimates are conservative. Explain your answer in terms of the intended use of the information (e.g. in terms of the optimization for which it will be used). *Hint: This is a bit of a trick question.*

   (a) (3 points) Available expressions

   (b) (3 points) Variables changed by a procedure

   (c) (3 points) Variables not changed by a procedure

   (d) (3 points) Copy statements reaching a given program point

3. For the control flow graph in Figure 1,

   (a) (3 points) Draw the dominator tree

   (b) (3 points) List the dominance frontiers of each node (assume nodes 7 and 8 go to exit)

   (c) (3 points) Put the control-flow graph in SSA form (you can eyeball this instead of running the algorithm by hand)

4. Write down a sequence of reduction steps reducing each of the following terms to normal form. For this problem, reduction is allowed anywhere within a term, including under a λ.

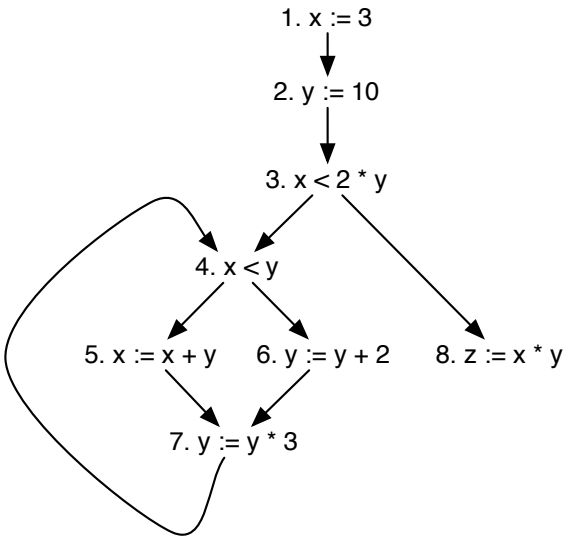

Figure 1: Control flow graph

(a) (3 points) $(\lambda x.x(xy))(\lambda u.u)$

(b) (4 points) $(\lambda xyz.zyx)aa(\lambda pq.q)$

(c) (4 points) $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$

Note: $\lambda xy.e$ is short for $\lambda x.\lambda y.e$. Remember also that the scope of λ extends as far to the right as possible, and that application associates to the left.

5. (a) (6 points) Give an encoding of lists in the lambda calculus. Your encoding should include combinators *nil*, *cons*, *head*, *tail*, and *isnil*, with the following requirements:

   - $head\ (cons\ e_1\ e_2) = e_1$
   - $tail\ (cons\ e_1\ e_2) = e_2$
   - $isnil\ nil\ e_1\ e_2 = e_1$
   - $isnil\ (cons\ e_1\ e_2)\ e_3\ e_4 = e_4$

   Here $=$ is beta-equality. Argue that your encoding is correct by showing that your combinators adhere to the above rules. *Hint:* This is exercise 5.2.8 in Pierce, chapter 5, which suggests representing the list $[x, y, z]$ as $\lambda cn.cx(cy(czn))$.

   (b) (4 points) Using Y and your combinators from part (a), write the *map* function, where (using OCaml notation) $map\ f\ [\,] = [\,]$ and $map\ f\ (x :: xs) = (f\ x) :: (map\ f\ xs)$. (Please use the combinators as primitives and do *not* expand them to their definitions.)

6. (5 points) (Barendregt exercise 6.8.14) Let

$$X = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$
$$Z = XXXXXXXXXXXXXXXXXXXXXXXXXX \quad (26\ X\text{'s})$$

   Show that Z is a fixed point combinator (that is, show that $ZX = X(ZX)$ under beta equivalence).

7. For each type, construct a simply-typed lambda calculus term (variables, functions, and function application only) whose *most general* type is that type, or argue that no term has that type. It should have the type you specify in the *empty* type environment; i.e., it has type τ such that $\vdash e : \tau$.

(Hint: You can double-check your answers in OCaml. *Extra credit*: for any type that has no simply-typed lambda calculus term, give an OCaml term that does have the type *without using the : operator to assign a type.*)

(a) (2 points) $\alpha \to \beta \to \beta$

(b) (2 points) $(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$

(c) (3 points) $\alpha \to \beta$

(d) (3 points) $\alpha \to \alpha \to \alpha$

8. (5 points) Does the simply-typed lambda calculus with integers have a *subject expansion* property, meaning if $\Gamma \vdash e : \tau$ and $e' \to e$, does $\Gamma \vdash e' : \tau$? Here $\to$ is reduction under call-by-value semantics. Either prove that subject expansion holds, or give a counterexample showing that it does not hold.

9. Suppose we were to add booleans to the simply-typed lambda calculus:

$$e ::= x \mid n \mid \mathit{true} \mid \mathit{false} \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e$$

(a) (4 points) Write down small-step call-by-value semantic rules for the new forms *true*, *false*, and *if*. (Here *if* should behave as it does in O'Caml, evaluating to the result of either the true or false branch depending on the guard.)

(b) (4 points) Extend the typing judgment $\Gamma \vdash e : \tau$ to the new forms *true*, *false*, and *if*.

(c) (8 points) Prove progress and preservation for the extended language. (You don't need to reprove the cases for the old forms; make your arguments as extensions to the proof given in the lecture slides.)

10. (5 points) Consider the following program, written in lambda calculus with tuples, integers, and strings:

let app2 $= \lambda f x y.(f\ x,\ f\ y)$ *in*
app2 $(\lambda x.x)$ *1 foo*

Write down the type for *app2* in simply-typed lambda calculus with Hindley-Milner style polymorphism. Does this program exhibit any run-time errors (i.e., will its evaluation ever be stuck)? Does the program type check using Hindley-Milner style polymorphism? Explain what goes wrong. Can you give a type for *app2* that is polymorphic but not Hindley-Milner such that this program would type check? (Note: You will not be able to construct a most-general type for *app2* without using intersection types, which we have not discussed, but your type should work for this particular use of *app2*.)

11. Consider the factorial function:

```
i = x;
r = 1;
while i > 0 do
  r = r * i;
  i = i - 1;
done
```

In particular, when this code terminates we will have, `r` = `x`! We wish to use abstract interpretation to prove that, given a non-negative `x` as input, this function always produces a positive `r` as output. Suppose we use as our abstract domain the rule of signs, so that integers are represented as one of $A = \{+, -, 0, \bot, \top\}$ and abstract states are maps from variables to values in A. Here, the concretization function γ (which maps $a \in A$ to sets of integers $S \subseteq \boldsymbol{Z}$ is defined as

x	$\gamma(x)$
$+$	$\{1, 2, 3, ...\}$
0	$\{0\}$
$-$	$\{-1, -2, -3, ...\}$
$\top$	*all integers*
$\bot$	$\emptyset$

The lattice ordering should also be obvious: $\bot \leq a$ and $a \leq \top$ for all $a \in A$.

(a) (4 points) Define the abstract semantics of the subtraction (`-`) and multiplication operations (`*`) on A. For example, $+ \texttt{ * } + = +$, while $\top \texttt{ * } a = \top$ for all $a \in \{+, -, 0, \top\}$. Write them out as tables. What would you have to do to prove that these operations are sound?

(b) (4 points) We would like to show that starting with an abstract state that maps `x` to either + or 0 will always end up with `r` as being mapped to abstract value +. As with the constant propagation example on slide 22 of Schmidt's slides, we will need to "accelerate" termination by joining (using the $\sqcup$ operator) the abstract states resulting from each iteration of the while loop with the state prior to entering the loop. Compute successive approximations to the state entering the while loop, using an abstract execution tree as in the Schmidt slides. Is it the case that you will be able to prove the desired result? What goes wrong?

(c) (4 points) Construct a new abstract domain A' that slightly refines A and is sufficient to prove the desired result. Draw a picture showing the lattice structure for A', and describe (briefly) operations in your new domain (if they are obvious, you can just say that you use the obvious abstract operations). Finally, show how an abstract interpretation of the program in A' will produce the desired result.