

631 exam practice mid-term

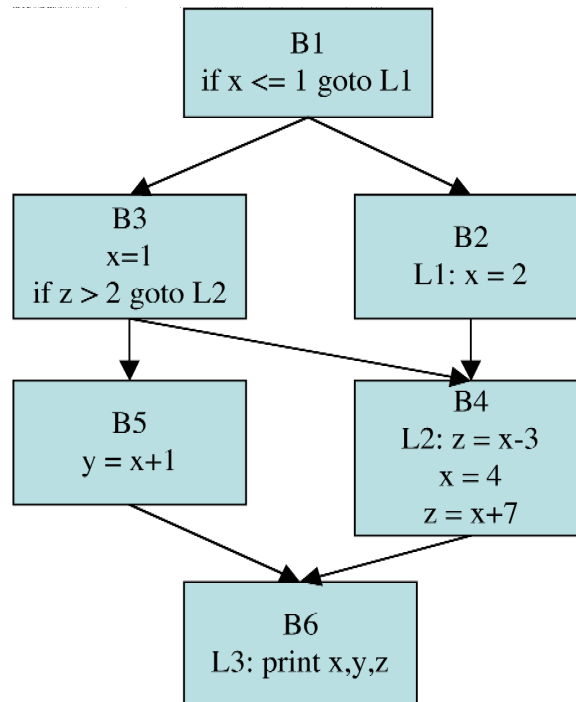
This program is to be used in questions 1-3.

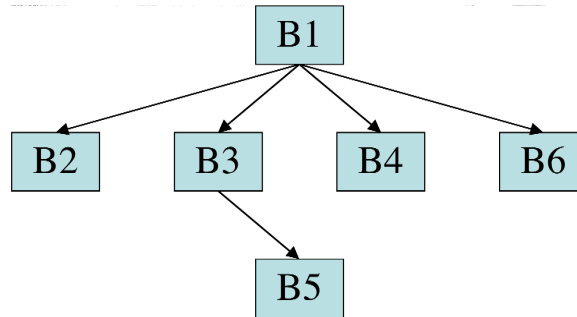
```
S0      if (z <= 1) goto L1
S1      x = 1
S2      if (z > 2) goto L2
S3      y = x+1
S4      goto L3
S5 L1   x = 2
S6 L2   z = x - 3
S7      x = 4
S8      z = x+7
S9 L3   print x,y,z
```

1. Control flow analysis

- (a) What are the basic blocks?
- (b) Show the control flow graph
- (c) Number the nodes of the CFG in reverse postorder.
- (d) Give the dominator tree for this program

**Answer:**





2. Control dependence - which block(s) are control-dependent on which block(s)?

Block is control dependent on edge

B2 B1 → B2

B3 B1 → B3

B4 B3 → B4 and B1 → B2

B5 B3 → B5

3. Dominance frontiers and SSA form

(a) For each block, give the dominance frontier and the iterated dominance frontier.

Block	DF	DF <sup>+</sup>
B2	B4	B4, B6
B3	B4, B6	B4, B6
B4	B6	B6
B5	B6	B6

(b) Give the (minimal) SSA form of the program.

```

if (z0 <= 1) goto L1
x1 = 1
if (z0 > 2) goto L2
y1 = x1+1
goto L3
L1 x2 = 2
L2 x4 = φ(x1, x2)
z1 = x4 - 3
x3 = 4
z2 = x3+7
L3 x5 = φ(x1, x3)
y2 = φ(y1, y0)
z3 = φ(z0, z2)
print x5, y2, z3
  
```

4. Data flow lattices

(a) Remember that we define our  $\sqsubseteq$  operator as  $u \sqsubseteq v \equiv u = u \sqcap v$ . A framework is monotone if and only if  $u \sqsubseteq v \Rightarrow f(u) \sqsubseteq f(v)$ .

Prove or disprove that even if a framework is not distributive (i.e.,  $f(u \sqcap v) = f(u) \sqcap f(v)$ ), monotonicity guarantees that  $f(u \sqcap v) \sqsubseteq f(u) \sqcap f(v)$ .

**Answer:** First, we prove that  $u \sqcap v \sqsubseteq v$ . To prove this, we need to prove  $u \sqcap v = u \sqcap v \sqcap v$ , which is true since  $\sqcap$  is associative  $v \sqcap v = v$ .

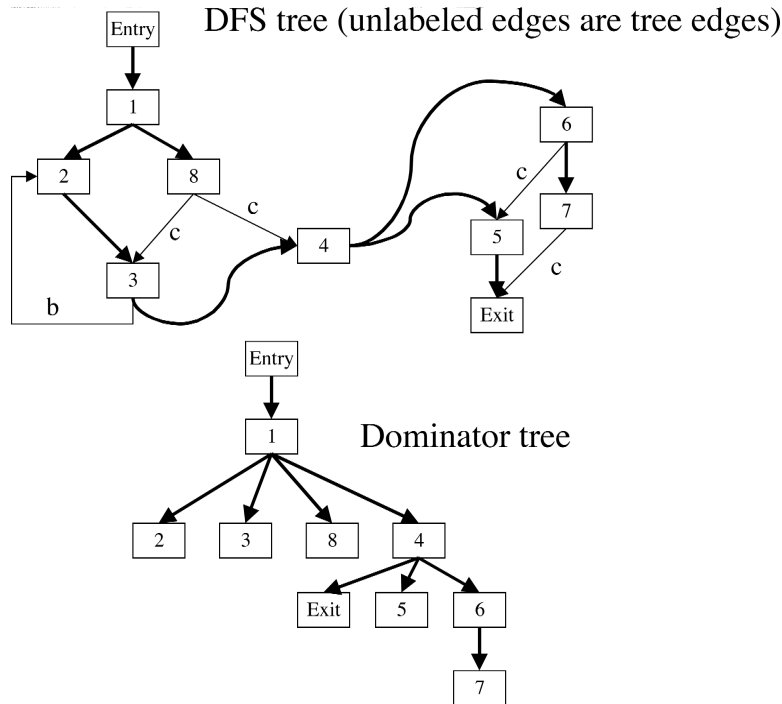
Similarly,  $u \sqcap v \sqsubseteq u$ .

To prove  $f(u \sqcap v) \sqsubseteq f(u) \sqcap f(v)$ , we need to prove that  $f(u \sqcap v) = f(u \sqcap v) \sqcap f(u) \sqcap f(v)$ . Since  $u \sqcap v \sqsubseteq u$  and  $f$  is monotone, we have  $f(u \sqcap v) \sqsubseteq f(u)$  and therefore  $f(u \sqcap v) = f(u \sqcap v) \sqcap f(u)$ . Therefore, our proof obligation is reduced to  $f(u \sqcap v) = f(u \sqcap v) \sqcap f(v)$ . Similarly, we know  $f(u \sqcap v) \sqsubseteq f(v)$  and use that to reduce our proof obligation to  $f(u \sqcap v) = f(u \sqcap v)$ , which is trivially true.

- (b) Informally/intuitively, describe what would be true of a system that is not monotone.

**Answer:** Learning more would result in you knowing less. The most likely situation would be if the analysis decides that a certain path is unfeasible, it is tempting to reset the lattice value to  $\top$ ; don't do this, it makes the framework non-monotone.

5. Depth First Search tree and dominator tree:



6. We want to be able to determine if a pointer might be null. We want to warn the user and refuse to compile when we find a statement that is a definite error, and omit run-time null pointer checks whenever possible.

Assume that doing a reference through a null pointer throws an exception that is not caught within the code we are analyzing.

For our analysis, try to compute accurate information for copy statements (e.g., after  $p = q$ , whatever we had previously known about  $q$  we now know about  $p$ ) and pointer uses (e.g., after  $p = q.x$  we know that  $q$  is non-null and don't know anything about  $p$ ).

Formulate this problem as one or more data flow problems. There are several different ways to do this; the more accurate your solution the higher your grade.

**Answer:** This is like a constant propagation problem, where the possible values are null or non-null. There are other ways to do it, but this is the way I will be doing it.

(a) What is your lattice (e.g., the type of the values you compute at each point)? For each variable, the value can be  $\top$  (no definitions seen), `null`, `non-null`, or  $\perp$  (either `null` or `non-null`).

(b) Discuss the ways in which your analysis is conservative, how it generate incorrect (but conservative) answers, and why the decisions we might make on it are safe.

We might compute an answer of  $\perp$  for a variable when the variable could only be `null` or could only be `non-null`. We could be approximate because of infeasible paths or because of data constraints we don't understand (e.g., the next pointer might never be `null`).

(c) What is your meet function? For each variable, the appropriate element-wise meet is shown below:

- $\perp \wedge X \equiv \perp$
- $X \wedge X \equiv X$
- $\top \wedge X \equiv X$
- `null`  $\wedge$  `non-null`  $\equiv \perp$

(d) I'll use a forward solution.

(e) Use  $\top$  to initialize In values.

(f) Describe how to compute the transfer functions.

**p = q** Copy information for q to information for p.

**p = null** Set p to null.

**p = &x** Set p to non-null.

**p.f** Set p to non-null. We could do something where if p were known to be null, we would set all values to top (it would be an infeasible path). But then the data flow problem would not be monotone and there would many questions about what answer we were computing.

**p = q.f** Set p to be  $\perp$  and q to be nonnull.

(g) If your framework monotone? Is it distributive? Justify your answers.

Given the limitation on how we handle a read of p.x, it is monotone. If  $v \leq u$ , then  $f(v) \leq f(u)$ . In fact, it is distributive as well  $f(u) \wedge f(v) = f(u \wedge v)$ . The transfer function simply sets specific values for some variables, and copies values for other variables. Doing the meet before or after makes no difference.

(h) Briefly describe the changes that would need to be made to take into account conditional tests such as `if p == null goto L5`.

We would have different transfer functions, one for each exit from a block. If an exit would be taken only under certain conditions, that would be reflected in the transfer function (in the above example, the transfer function for the exit that led to L5 would set p to null and the other exit would set p to non-null).



Block	Dominance Frontier	Iterated Dominance Frontier
B1	B3	B3, B2
B2	B3	B3, B2
B3	B2	B2, B3
B6	B2	B2, B3
B7	B5	B5

8. Put the following program in SSA form (you may draw a control flow graph to illustrate your solution):

```

x := 0;
do {
  x := x + 1;
  z := x;
  y := 0;
  if (...) {
    y := 1;
  }
  w := y + z;
} while (...);
print(x, y, z, w);

x1 := 0;
do {
  x3 := φ(x1, x2)
  x2 := x3 + 1;
  z := x2;
  y1 := 0;
  if (...) {
    y2 := 1;
  }
  y3 := φ(y1, y2)
  w := y3 + z;
} while (...);
print(x3, y3, z, w);

```

9. Type inference through data flow analysis. We have a dynamically typed language with a simple type system:

Type = INT — FLOAT — &Type — unknown

The third possibility indicates a pointer type (e.g., &int (pointer to an int) or &&float (pointer to a pointer to a float)).

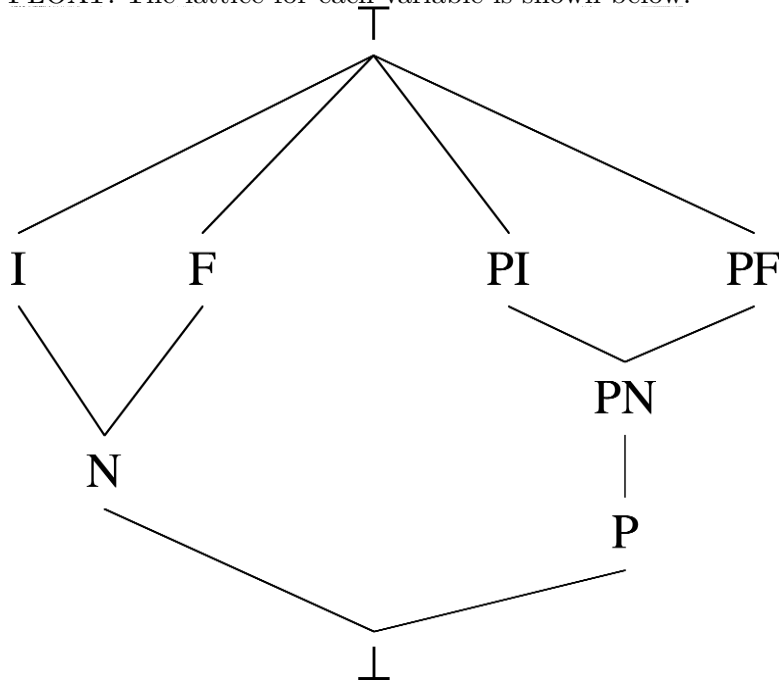
Although this system does not require static types for variables, we want to deduce static types when possible so that we will not have to generate run-time type-checks when possible.

Assume that the language allows simple arithmetic and comparison operators on ints and floats (it suffices to discuss just + in your answer), dereferencing a pointer and taking the address of a variable. For arithmetic operations, ints are converted to floats when combined with floats. Dereferencing an int or pointer arithmetic generates a run-time exception.

Define and describe an appropriate data flow analysis framework to compute static types when appropriate. Discuss how your framework can be inaccurate. Do you compute a meets-over-all-paths answer?

I haven't spelled out all the details. Treat this as though you were being asked to come up with an example for a textbook. Fill in details that illuminate the problem, and omit those that do not.

**Answer** OK, I'm going to use a simple lattice and not keep track of pointers to pointers, other than to just recognize that they are a pointer. The type NUMBER will indicate either an INT or a FLOAT. The lattice for each variable is shown below:



The meet function is simply taking the greatest lower bound for each variable. The data flow graph is initialized to a value of  $\top$  for variable. Assume that uninitialized variables are assumed to have the integer value 0. Then the value at start is for all variables to have type I.

OK, now the difficult part: computing the transfer function. For expressions, the following table describes the type of each expression. The left column gives the minimal types that will produce the result type in the right column. The type of an expression is determined by the first row that matches. For example,  $I + F$  has type F. I assume that syntactic constraints prevent things such as taking the address of an R-value.

+			*			&	
expression	type		expression	type		expression	type
$I + I$	I		$*PI$	I		$\&I$	PI
$N + F$	F		$*PF$	F		$\&F$	PF
$F + N$	F		$*PN$	N		$\&N$	PN
$\perp + \perp$	N		$*\perp$	$\perp$		$\&\perp$	P

OK, now we deal with assignment. Assignment to a simple variable sets the type of the variable on the left-hand side to be the type on the right-hand side. Further, the types of any pointers that might be pointing to the variable on the left-hand side are updated. Since we don't have points-to analysis, we will just use type information. For example, if x was previously an Int, and a float value is assigned to x, then the type of x becomes F, and the types of any variables that might have been pointing to x (i.e., those with types PI, PN, P or  $\perp$  get merged with PF.

When we assign through a pointer (e.g.,  $*p = 42$ ), we have to make similar updates. If  $p$  had previously been of type  $PF$ , then  $*p = 42$  causes all variables of type  $F$ ,  $N$  or  $\perp$  to be merged with  $I$ .

The framework is monotone, but not distributive. For example, in the code fragment below, the system would only determine that  $z$  was of type numeric, while a meets-over-all-paths solution would determine that  $z$  was of type float.

```

if (...) {
  x = 1;
  y = 2.0;
} else {
  x = 1.0;
  y = 2;
}
z = x+y

```

The framework is approximate because:

- Because the framework is not distributive, we only compute a maximal fixed point solution, rather than a meets-over-all-paths solution.
- The transfer functions are approximate, particularly with regards to pointers
- The presense of infeasible paths

However, in all cases we compute an answer that is  $\leq$  "truth", which allows us to safely use the answer.

**Alternative solutions** You might consider a solution which kept track of types such as  $PPI$ ,  $PPPF$ . If the lattice was set up so that the meet of  $PPI$  and  $PPPI$  was bottom, then you don't have any infinite ascending or decending chains, and everything works just fine. If instead the meet of  $PPI$  and  $PPPI$  is a type  $PP$  that indicates a pointer to a pointer to something, then you have to worry about infinite chains and non-termination.

10. Exercise 1.3 from the handout

**Answer:** We know that for all  $X$  and  $Y$

$$X \subseteq \gamma(Y) \Leftrightarrow \alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma'(Y)$$

which directly reduces to

$$X \subseteq \gamma(Y) \Leftrightarrow X \subseteq \gamma'(Y)$$

which tells us that  $\gamma = \gamma'$ .

11. Exercise 1.4 from the handout

**Answer:** This turned out to be too much work for an exam question. You need to do lots of steps, such as the one for  $G_{\text{exit}}(4)$  and  $F_{\text{exit}}(4) = RD_{\text{exit}}(4)$  given in the middle of page 16.

For reaching definitions you have  $F = \alpha \circ G \circ \gamma$ . This would not be the case for other problems, in particular problems that are not distributive.

The inductive proof is straightforward for the case where  $F = \alpha \circ G \circ \gamma$ .

12. Consider an abstract interpretation that abstracts integers as either even or odd. In particular, the environment of a program, rather than mapping variables to specific integer values, maps each variable to either odd or even.

- (a) Give the abstraction ( $\alpha$ ) and concretization ( $\gamma$ ) function. Show that  $(\alpha, \gamma)$  is a Galois connection (e.g., that  $\alpha(X) \subseteq Y \Rightarrow X \subseteq \gamma(Y)$ ).

**Answer:** Our concrete semantics will record the possible sets of values variables might have at a particular program point. For example, if a point a, either  $x=1$  and  $y=2$ , or  $x=2$  and  $y=3$ , then our concrete semantics would be  $\{(x=1, y=2), (x=2, y=3)\}$ . Let  $V$  be our set of variables.

Our abstraction function takes an environment and for each variable, says whether the values taken on by that variable are odd, even, or either.

For example,  $\alpha(\{(x=1, y=2), (x=2, y=4)\}) = \{(x, \text{either}), (y, \text{even})\}$ .

On an element by element basis, we have  $\text{odd} \sqsubseteq \text{either}$  and  $\text{even} \sqsubseteq \text{either}$ . When comparing two abstract values  $a_1$  and  $a_2$ , we have

$$a_1 \subseteq a_2 \Leftrightarrow \forall v \in V, a_1(v) \sqsubseteq a_2(v)$$

For example,  $\{(x, \text{odd}), (y, \text{even})\} \sqsubseteq \{(x, \text{either}), (y, \text{even})\}$ .

A full proof that this is a Galois connection is too long for a mid-term question. Sorry about that.

- (b) Give the abstract meaning of  $+$ ,  $*$  and  $/$ .

$+$	odd	even	either	$*$	odd	even	either	$/$	odd	even	either
odd	even	odd	either	odd	odd	even	either	odd	either	either	either
even	odd	even	either	even	even	even	either	even	either	either	either
either	either	either	either	either	either	either	either	either	either	either	either

- (c) Give the abstract value of the environment at the end of the following code:

`x := 5; y := 4 z := 21; while (...) x++; y--; z := x+y;`

**Answer:**  $\{(x, \text{either}), (y, \text{either}), (z, \text{either})\}$

- (d) Specify a more precise abstraction that allows derivation of the fact that  $z$  is odd. In particular, you should allow abstract states such as "either  $x$  is odd and  $y$  is even, or  $x$  is even and  $y$  is odd". Give the revised abstraction ( $\alpha$ ) and concretization ( $\gamma$ ) function. Show that  $(\alpha, \gamma)$  is a Galois connection (e.g., that  $\alpha(X) \subseteq Y \Rightarrow X \subseteq \gamma(Y)$ ).

**Answer:** The revised abstraction would be sets of a vector of even/odd bits. For example,  $\alpha(\{(x=1, y=2), (x=2, y=4)\}) = \{(\text{odd}, \text{even}), (\text{even}, \text{even})\}$ . The maximum number of vectors in an abstraction would be  $2^{|V|}$ .

This would allow us to compute the following abstract value of the program fragment:  $\{(\text{odd}, \text{even}, \text{odd}), (\text{even}, \text{odd}, \text{odd})\}$ .

Once again, a full proof that this is a Galois connection is too long for a mid-term question.