CS 243 Final Examination Solution

Winter 2023

March 21, 2023

This is an open-book, open-notes, open-laptop, *closed*-network exam. You have 3 hours to complete the exam. The examination has 8 problems worth 180 points. Please budget your time accordingly. Write your answers in the space provided on the exam. If you use additional scratch paper, please turn that in as well.

Your Name: _____ SUNet ID: _____

The following is a statement of the Stanford University Honor Code:

1. The Honor Code is an undertaking of the students, individually and collectively:
   (a) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
   (b) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.
2. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.
3. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

Signature: _____

Problem	#1	#2	#3	#4	#5	#6	#7	#8	**Total**
Max	21	14	15	20	25	25	30	30	180

Problem 1. Short questions. You must **justify your answer** to each question in 1–2 sentences. [21 points, 3 points each]

1. True or False. A semi-lattice of infinite domain size must have an infinite descending chain.

   False. Consider constant propagation.

2. The Cartesian product of two sets is defined as $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. $\mathbb{Z}^+$ is the set of all positive integers. Consider the set of $\mathbb{Z}^+ \times \mathbb{Z}^+$ as the domain. We define a meet operator $\min((x_1, x_2), (y_1, y_2)) = (\min(x_1, y_1), \min(x_2, y_2))$. Does this form a semilattice? If yes, write down the top element and bottom element if they exist. If no, briefly explain why.

   Yes. The three axioms are satisfied. The top element does not exist, the bottom element is $(0, 0)$.

3. Let a be the number of live ranges in a program, b be the number of *merged* live ranges (as defined for register allocation), and c be the number of variables in the same program translated to static single-assignment (SSA) form. Express the relationship between a, b, and c using one or more inequalities. Explain your answer.

   $b \leq a \leq c$. $b \leq a$ since there cannot be more *merged* live ranges than live ranges. $a \leq c$ since every live range corresponds to one variable definition, which corresponds to one SSA variable – but SSA can introduce more variables for the ϕ-nodes.

4. After successfully software pipelining a loop with an initiation interval T, registers `R1` and `R2` in the original program have live ranges of $3 \times T$ and $4 \times T$ cycles, respectively. Suppose you have 10 registers to devote to `R1` and `R2`, how many times would you unroll the steady state of the loop?

   4 unrolls. This follows from the formula discussed in class $\max_r \left(\frac{\text{lifetime}_r}{T} \right)$. In homework, we discussed an alternative scheme that uses fewer registers but unrolls more, but it's unneeded (and harmful, as it increases code size) in this case since we have enough registers.

5. True or False. A single-issue machine will not benefit from global instruction scheduling because it cannot issue more than one instruction per cycle.

   False. As an example, the program can benefit from moving operations with a latency of more than one clock cycle earlier.

6. True or False. The Fourier–Motzkin elimination (FME) algorithm returns real solutions rather than integer solutions. But since loop indices are always integers, FME is unsuitable for conducting data dependence analysis for automatic loop parallelization.

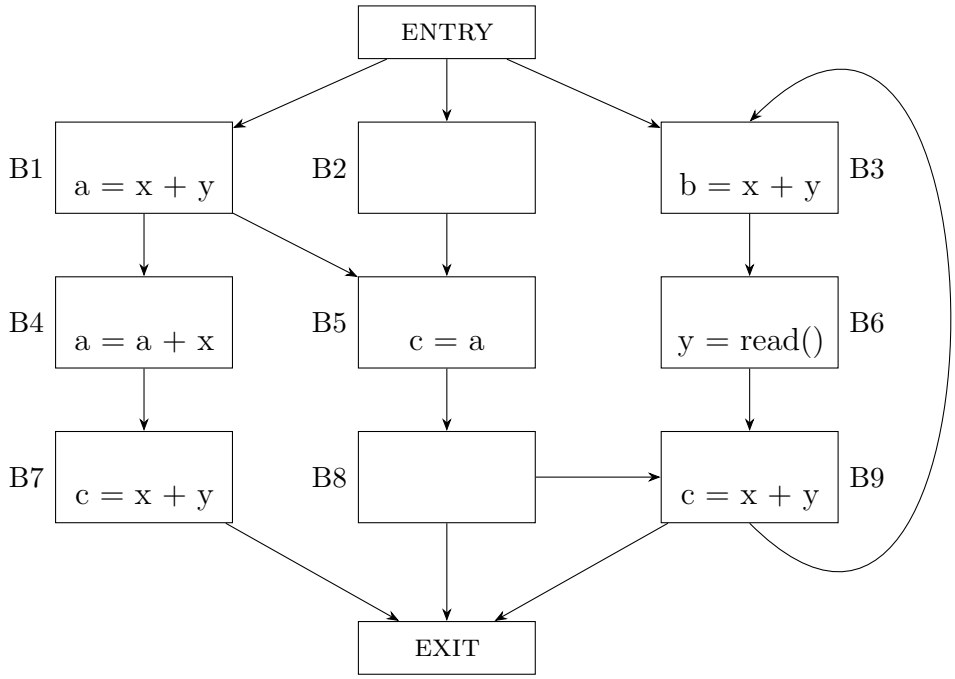
   False. If FME returns no real solution, then there must be no integer solution, so the loop can be parallelized. Even if it returns non-integer solutions, there are workarounds (such as by forming subproblems that constrain away the real solutions).

7. True or False. If the two outermost loops of a 3-deep loop can be parallelized without using any synchronization, then the loop must be fully permutable.

   True. This implies that the outer two loops are do-all loops, and the only data dependencies that can exist are relative to the last level. Thus, iterating in a different order is always acceptable.

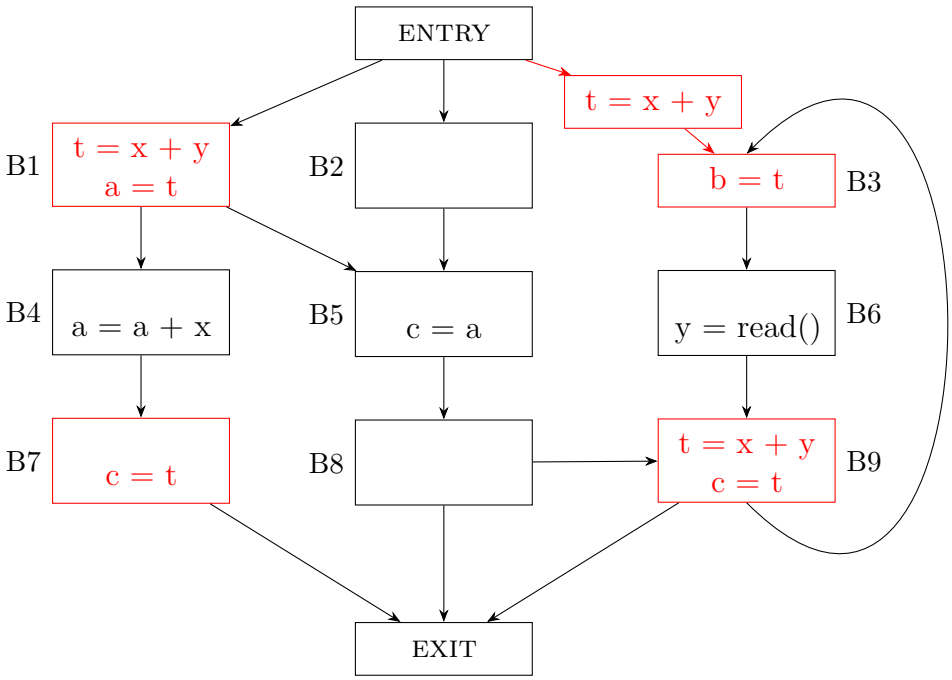
Problem 2. Partial Redundancy Elimination [14 points]

Answer the following question based on the program below.



Show the result of running partial redundancy elimination. **Don't apply any optimization other than PRE.** What's the final optimized flow graph? You don't need to show the intermediate steps.

Solution:


ENTRY
t = x + y
B1
t = x + y
a = t
B2
b = t
B3
B4
a = a + x
B5
c = a
y = read()
B6
B7
c = t
B8
t = x + y
c = t
B9
EXIT

Problem 3. Garbage Collection [15 points, 5 points each]

You have the following automatic memory management systems at your disposal:

(a) Reference counting

(b) Generational garbage collector with incremental marking

(c) Non-generational tracing garbage collector

Which garbage collector would you choose for each of the following scenarios? Explain your choice based on performance and usability.

1. A numerical program running on a multiprocessor that only allocates memory at the beginning of the program, and makes ten passes through all allocated memory.

   (c) Non-generational tracing garbage collector.

   Reference counting adds a lot of overhead checking whether to free the memory, even though we know memory will never be dead. Parallelism is also bad for reference counting since the code now has to communicate between processors to discover the latest reference count. Additionally, the generational hypothesis does not hold true in this scenario, making (b) a poor choice. With the stated memory allocation patterns where we never allocate memory after the initial setup, the non-generational GC will probably never get triggered at all, making concerns about pause time moot.

2. A programming language designed for resource-constrained embedded devices, where small executable code size is more important than speed.

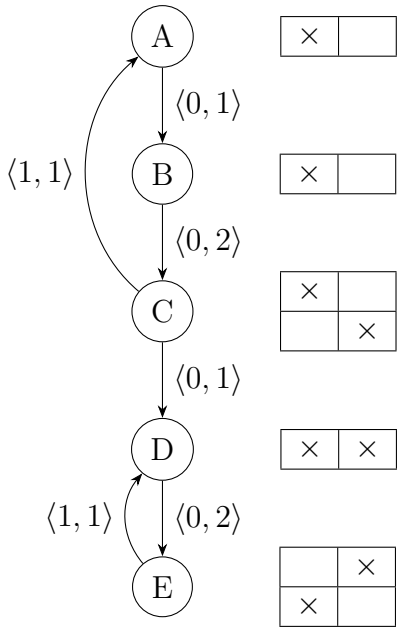
   (a) Reference counting. Reference counting is super simple to implement. There will be some overhead in code size when entering/leaving a scope, but that is minor compared to an entire tracing garbage collection system which needs to maintain its own data structures and perform graph traversals. Discussion on runtime memory usage, while important in real world, is less important than code size for this question.

3. A single-threaded dynamic language that allocates a lot of short-living intermediate objects on the heap.

   (b) Generational garbage collector with incremental marking. The generational hypothesis is stated to be true, which makes this a good choice. Reference counting can work, but will cause more memory fragmentation than generational GC.

Problem 4. Software Pipelining [20 points]

Consider the following dependence graph for a single iteration of a loop, with resource constraints:



1. What are the bounds on the initiation interval T according to the precedence and resource constraints for this program? [4 points]

   The A–B–C–A cycle imposes a bound of 4 due to data dependence, as computed by dividing cycle length by iteration difference. (The cycle D–E–D imposes a bound of 3 cycles.)

   Using resource constraints, the left resource imposes a bound of 5 and the right resource 3. So overall, $T \geq 5$.

2. Is it possible to software-pipeline this loop with the minimum bound found in the previous part? If not, what is the minimum possible initiation interval?

   Show the modulo reservation table for the optimal software-pipelined schedule. Also show the code schedule for an iteration in the source loop. [8 points]

   The minimum initiation interval is 5. Modulo reservation table:

A	C
B	E
E	
D	D
C	

   Single iteration: A, B, nop, nop, C, nop, nop, nop, D, nop, nop, E.

   Note that the number of instructions between A and C (exclusive) must not exceed $T-1=4$ due to the "loop-back" edge from C to A. Similarly, the number of instructions between D and E (exclusive) must not exceed 4.

3. Can the scheduling algorithm described in class produce the optimal schedule for this loop? If not, show the modulo reservation table and code schedule generated by the algorithm. [6 points]

   The scheduling algorithm described in class will backtrack only within strongly connected components. In this case, it will not produce an optimal schedule. With $T=5$, the algorithm will first schedule the $\{A, B, C\}$ SCC shown on the left:

A	
B	
C	
	C

A	
B	
D	D
C	
	C

   Now we are stuck. There is only one place we can schedule D (shown on the right), and that place does not allow us to schedule E without breaking some constraints.

   However, the algorithm will be able to schedule instructions with an initiation interval of 6. Modulo reservation table:

A	
B	E
E	
C	
	C
D	D

   Single iteration: A, B, nop, C, nop, D, nop, E

Problem 5. Pointer Analysis [25 points]

Consider the following simple implementation of a singly-linked list. We use static methods to avoid `this` pointers which may be ambiguous.

```
1  public class Node {
2      public Node n = null;
3
4      // This function is callable as: Node.extend(a, b)
5      public static void extend(Node a, Node b) {
6          if (b == null)
7              b = new Node();    // h3
8          Node c = Node.getTail(a);
9          c.n = b;
10     }
11
12     // This function is callable as: Node.getTail(d)
13     public static Node getTail(Node d) {
14         if (d.n == null)
15             return d;
16         Node e = d.n;
17         return Node.getTail(e);
18     }
19
20     public static void main(String[] args) {
21         Node x = new Node();    // h1
22         Node y = new Node();    // h2
23         Node.extend(x, y);
24     }
25 }
```

Apply flow-insensitive context-insensitive pointer analysis to the program and write down all `vP` and `hP` tuples derived from the analysis. `h1`, `h2`, `h3` are the allocation sites, and `a`, `b`, `c`, `d`, `e`, `x`, `y` are all the variables.

The complete answer is as follows:

vP(x, h1)
vP(y, h2)
vP(a, h1)
vP(b, h2), vP(b, h3)
vP(c, h1), vP(c, h2), vP(c, h3)
vP(d, h1), vP(d, h2), vP(d, h3)
vP(e, h2), vP(e, h3)

hP(h1, n, h2), hP(h1, n, h3)
hP(h2, n, h2), hP(h2, n, h3)
hP(h3, n, h2), hP(h3, n, h3)

Steps:

Start with:
vP(x, h1) – line 21
vP(y, h2) – line 22
vP(b, h3) – line 7

And rules:
assign(a, x) – line 23
assign(b, y) – line 23
assign(c, d) – lines 8 & 15
assign(d, a) – line 8
assign(d, e) – line 17
load(d, n, e) – line 16
store(c, n, b) – line 9

We first get the current set of vP tuples:
vP(x, h1)
vP(y, h2)
vP(a, h1)
vP(b, h2), vP(b, h3)
vP(d, h1)
vP(c, h1)

Current set of hP tuples:
hP(h1, n, h2), hP(h1, n, h3)

Then use load to generate new vP tuples:
vP(e, h2), vP(e, h3)

Propagate to d then c:
vP(d, h2), vP(d, h3)
vP(c, h2), vP(c, h3)

Finally all hP tuples:

hP(h1, n, h2), hP(h1, n, h3)
hP(h2, n, h2), hP(h2, n, h3)
hP(h3, n, h2), hP(h3, n, h3)

Problem 6. Satisfiabiliy Modulo Theories [25 points]

In this problem, we use SMT to analyze the following function. We assume that `data` is a zero-indexed array of length `N`.

```
1  int foo(int data[], int N) { {
2      int v = 0;
3      int i = 0;
4      while (i <= N) {
5          if (data[i] > v) {
6              v = data[i];
7          }
8          i = i + 1;
9      }
10     return v;
11  }
```

1. Notice the while loop on lines 4–9. One method for handling a loop in SMT analysis is to unroll it several times. Show the SSA form of the code when you unroll the loop twice (convert the first two iterations of the `while` loop into `if` statements). [12 points]

```
1  v0 = 0;
2  i0 = 0;
3  if (φ0 = (i0 <= N)) {
4      if (φ1 = (data[i0] > v0)) {
5          v1 = data[i0];
6      }
7      v2 = φ1 ? v1 : v0;
8      i1 = i0 + 1;
9
10     if (φ2 = (i1 <= N)) {
11         if (φ3 = (data[i1] > v1)) {
12             v3 = data[i1];
13         }
14         v4 = φ3 ? v3 : v2;
15         i2 = i1 + 1;
16     }
17     v5 = φ2 ? v4 : v2;
18     i3 = φ2 ? i2 : i1;  // optional
19 }
20 v6 = φ0 ? v5 : v0;
21 i4 = φ0 ? i3 : i0;  // optional
22 return v6;
```

2. Let P be the SMT constraints based on the SSA form of the code with the loop unrolled twice. Write a SMT formula that is satisfied if there is an out-of-bound array access. Make sure to take into account path conditions for each array access.

   Will an SMT solver return `SAT` or `UNSAT`? If `SAT`, give a satisfying model. If `UNSAT`, briefly explain why. [5 points]

   $$P \wedge \phi_0 \wedge ((i_0 < 0 \vee i_0 \geq N) \vee (\phi_2 \wedge (i_1 < 0 \vee i_1 \geq N)))$$

   `SAT`. One model is where $N = 0$.

3. Suppose we can safely assume that the `data` array has at least 10 elements. Still unrolling the loop twice, write down the updated SMT formula that is satisfied if there is an out-of-bound array access. Will an SMT solver return `SAT` or `UNSAT`? If `SAT`, give a satisfying model. If `UNSAT`, briefly explain why. [4 points]

   `UNSAT`. Conjunct the previous formula with $N \geq 10$.

4. Again assume that the `data` array has at least 10 elements. What if we unroll the loop 20 times? Will an SMT solver return `SAT` or `UNSAT` on the updated constraint? If `SAT`, give a satisfying model. If `UNSAT`, briefly explain why. [4 points]

   `SAT`. One model is where $N = 10$, `data` is arbitrary.

Problem 7. Parallelization with Locality Optimization [30 points]

Consider a multiprocessor with 8 processors, where each processor has two levels of caching, and there is a **significant overhead** in moving data between processors. It is therefore very important to maximize locality. The following function is a simplification of the ADI Integration Method; it is executed repeatedly in the full program. Your task is to parallelize the function, while maximizing locality. Assume that the array `A` is much larger than 8 times the size of the first-level cache in each processor.

```
1  for i = 1 to N {
2      for j = 1 to N {
3          A[i,j] = 0.5 * (A[i,j] + A[i,j-1]);
4      }
5  }
6  for i = 1 to N {
7      for j = 1 to N {
8          A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
9      }
10 }
```

1. Without performing any code transformations, indicate which of the loops in this function are do-all loops? (Refer to the loops by their line numbers.) Suppose all such do-all loops are parallelized by inserting barriers at the end of each loop. Discuss the cache performance for the individual processors, as well as the synchronization and communication cost between processors. [6 points]

   Loops at lines 1 and 7 are parallelizable. The first loop nest has good performance since individual processors have memory locality. The second loop nest has bad performance since every processor needs to work on the same block of memory (e.g., processor 1 writes to $A[1,1]$, processor 2 writes to $A[1,2]$), increasing communication costs dramatically.

   Additionally, there must be a barrier between the two loops, incurring synchronization cost. More importantly, the first loop partitions by rows, the second loop partitions by columns, requiring the whole array to be transposed, which is extremely costly.

2. Does this function have any communication-free parallelism (i.e., can you transform the program into a **single** nested loop nest whose one or more outer loops are do-all loops)? If so, show the SPMD code. If not, briefly explain why. [6 points]

   Notation: use i, j to describe the first loop nest, and i', j' the second loop nest.

   There is no communication-free parallelism. Suppose for contradiction that there is, and we have processor mappings $p(i,j) = xi + yj + z$ for the first loop nest and $p'(i',j') = x'i' + y'j' + z'$ for the second loop nest. We know that iteration (i',j') depends on iteration (i,j) in the first loop nest and also $(i'-1,j')$, and iteration (i,j) depends on $(i,j-1)$. Communication-free parallelism implies that dependent iterations must

be run on the same processor, so we have

$$p(i, j-1) = p(i, j) = p'(i', j') = p'(i'-1, j')$$
$$xi + y(j-1) + z = xi + yj + z = x'i' + y'j' + z' = x'(i'-1) + y'j' + z'.$$

The first equality implies $y = 0$. The second implies $x = x'$, $y = y'$, $z = z'$. The third implies $x' = 0$. In other words, the processor mappings $p(i, j) = p(i', j') = z$ map all iterations to a single processor, so there is no parallelism.

3. Does this function have pipelined parallelism? If so, show the transformed code with as many outermost fully permutable loops as possible. (You only need to show the transformed sequential code, there is no need to parallelize it with synchronization.) [9 points]

   Yes.

   Note that the following loop does not give the right answer because there is an anti-dependence from the read of A[i,j-1] and the store of A[i,j] in the second loop.

```
for i = 1 to N {
    for j = 1 to N {
       A[i,j] = 0.5 * (A[i,j] + A[i,j-1]);
       A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
   }
}
```

   One solution is to remove the anti-dependence by saving the value that would otherwise be overwritten.

```
for i = 1 to N {
    tmp = A[i, 0];
    for j = 1 to N {
       A[i,j] = 0.5 * (A[i,j] + tmp);
       tmp = A[i,j];
       A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
   }
}
```

   Another solution that only uses affine partitioning is to shift the second loop nest by 1 to satisfy the anti-dependence.

```
for i = 1 to N+1 {
    for j = 1 to N {
       if (i <= N)
          A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
       if (i > 1)
          A[i-1,j] = 0.5 * (A[i-1,j] + A[i-2,j]);
   }
}
```

We give full credits to students who recognize that it is a full permutable loop nest even if they do not handle the anti-dependence (because of the limited time they have to solve the problem in the final).

4. Describe how you would parallelize this function. Discuss the cache performance for the individual processors, as well as the synchronization and communication cost between processors. [9 points]

   Parallelize with blocking. This hinders some parallelism, but in exchange it:

   (a) reduces communication significantly since now processors only need to communicate the "edges" of the blocks to each other;

   (b) provides good cache locality, as long as we make sure the same processor works on adjacent blocks;

   (c) minimizes synchronization since two processors only sync once every B elements processed, where B is the block size.

Problem 8. Optimizing Dynamically Typed Languages II [30 points]

Consider a dynamically-typed language similar to JavaScript and Python. The language has two kinds of instructions of interest to us:

- $x = T(\ldots)$, where T is one of `int`, `float`, and `string`. This sets the variable x to a value of type T.
- $x = y + z$, where x, y, z are all variables.

Unlike statically-typed languages like C and Java, the type of a program variable is allowed to change in this language. As an example, the program `x = int(1); x = float(3.14)` will result in variable `x` having an `int` value at first, but afterwards a `float` value.

The addition operator `+` accepts operands of any types with the following rules:

1. The sum of two values of the same type has that same type.
2. The sum of an `int` and a `float` is a `float`.
3. The sum of a `string` and a value of any type is a `string`.

Suppose we are writing an optimizing compiler for this language. Because there are special instructions for adding two integers (`iAdd`), adding two floats (`fAdd`), and adding two strings (`sAdd`), your task is to identify those additions that can be replaced with such special instructions at compile time.

To get full points, your dataflow analysis should identify as many opportunities for optimization as possible.

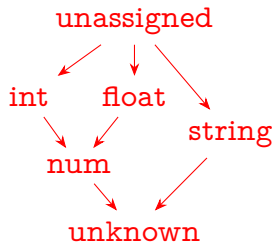
This problem has four parts: (1) Define your dataflow analysis by filling in the table below. (2) State clearly how you perform the optimization. (3) Is your dataflow analysis monotone? Explain. (4) Is your dataflow analysis distributive? Explain.

(Questions begin on the next page.)

1. Dataflow analysis specification. You may assume that each instruction is in its own basic block. [22 points]

Direction of your analysis (forward/backward)	Forward
Lattice elements and meaning	Map from each variable to a subset of {`int`, `float`, `string`}
Meet operator	For each variable: set union
Is there a top element? If yes, what is it?	Mapping each variable to ∅
Is there a bottom element? If yes, what is it?	Mapping each variable to {`int`, `float`, `string`}
Transfer function	For b: $x = T(\ldots)$: $\textsc{out}[b, v] = \textsc{in}[b, v]$ for all $v \neq x$, and $\textsc{out}[b, x] = \{T\}$. For b: $x = y + z$: $\textsc{out}[b, v] = \textsc{in}[b, v]$ for all $v \neq x$; for $\textsc{out}[b, x]$, let $\oplus$ represent the type rules of `+`: $\textsc{out}[b, x] = \{\tilde{y} \oplus \tilde{z} \mid \tilde{y} \in \textsc{in}[b, y], \tilde{z} \in \textsc{in}[b, z]\}$.
Boundary condition	Mapping each variable to ∅
Interior points	Mapping each variable to ∅

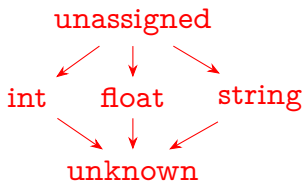
Alternative correct solution:



This works as well as the first solution, but the transfer function is more onerous to specify.

Common mistakes:

1. Using a constant propagation–style lattice:



   The problem is that it's not precise enough. Consider the program:

```
1 if ...: a = int(0)
2 else:   a = float(3.14)
3 b = float(3.14)
4 c = a + b
```

```
5 d = c + b
```

After the `if`–`else` branch, we know that variable `a` can be either an `int` or a `float`. So variable `c` must be a `float`, meaning that the last operation can be converted to `fAdd`. But with this smaller lattice, `a` would have value `unknown`. So `c` would also be `unknown`, obviating the optimization.

This non-optimal solution is monotonic but not distributive. For non-distributivity, consider the two mappings

$$V = \{a \mapsto \texttt{int}, b \mapsto \texttt{float}\}, W = \{a \mapsto \texttt{float}, b \mapsto \texttt{float}\}$$

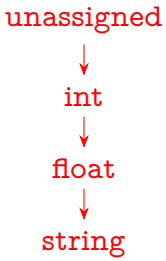
and the program `a = a + b`. We find that

$$f(V) = f(W) = \{a \mapsto \texttt{float}, b \mapsto \texttt{float}\} = f(V) \wedge f(W),$$

but

$$V \wedge W = \{a \mapsto \texttt{unknown}, b \mapsto \texttt{float}\} = f(V \wedge W).$$

2. Using a lattice based on type rules for addition:



The problem here is that this is unsafe. Consider the same program as above:

```
1 if ...: a = int(0)
2 else:   a = float(3.14)
3 b = float(3.14)
4 c = a + b
5 d = c + b
```

We know that after the `if`–`else` branch, `a` can be either an `int` or a `float`. But the analysis would conclude that it is a `float`, and try to optimize the next line `c = a + b` using `fAdd`. This is incorrect and unsafe.

This solution is monotonic and distributive.

2. How do you optimize the program? [2 points]

   For each instruction of form b: $x = y + z$ where $\text{IN}[b, y] = \text{IN}[b, z] = \{T\}$ for the same type T, replace it with the specialized instruction for adding two variables of type T.

3. Is your dataflow analysis monotone? Briefly sketch a proof for it if so, or provide a counterexample if not. [3 points]

   Yes. What follows is a full proof, and not expected on an exam.

   The $x = T(\ldots)$ case is easy: there is no cross-variable dependency, and the transfer function is either the identity function (for $v \neq x$) or the constant function (for $v = x$).

   Now consider the $x = y + z$ case. Let V, V' be variable mappings where $V \leq V'$, and suppose for now that they differ at only one variable y. If $V'[y] \subseteq V[y]$, meaning that $V[y]$ has more possibilities than V', then clearly $f_b(V')[x] \subseteq f_b(V)[x]$. Variables other than x and y are equal between $f_b(V)$ and $f_b(V')$, so $f_b(V) \leq f_b(V')$. The same argument works if they only differ at z. If they differ at only one variable, the destination x (which is not one of the operands), then we see that $f_b(V)$ and $f_b(V')$ would both recompute x, so $f_b(V) = f_b(V')$. Finally, if they differ somewhere else, then that variable would not change, so $f_b(V) \leq f_b(V')$.

   For any two mappings V and V' such that $V \leq V'$, there is always a chain of mappings $V_1, V_2, \ldots$ such that $V \leq V_1 \leq V_2 \leq \cdots \leq V'$, where every consecutive pair of values differs in at most one variable. We know from above that $V_i \leq V_{i+1}$, so by induction, monotonicity holds for any two mappings $V \leq V'$.

4. Is your dataflow analysis distributive? Briefly sketch a proof for it if so, or provide a counterexample if not. [3 points]

   No. Let $V = \{a \mapsto \{\texttt{float}\}, b \mapsto \{\texttt{string}\}\}$, $W = \{a \mapsto \{\texttt{string}\}, b \mapsto \{\texttt{float}\}\}$. Suppose we have the instruction $a = a + b$. Then the transfer function f has the following behavior:

$$\begin{aligned} f(V) &= \{a \mapsto \{\texttt{string}\}, b \mapsto \{\texttt{string}\}\}, \\ f(W) &= \{a \mapsto \{\texttt{string}\}, b \mapsto \{\texttt{float}\}\}, \\ f(V) \wedge f(W) &= \{a \mapsto \{\texttt{string}\}, b \mapsto \{\texttt{float}, \texttt{string}\}\}. \end{aligned}$$

   But $V \wedge W = \{a \mapsto \{\texttt{float}, \texttt{string}\}, b \mapsto \{\texttt{float}, \texttt{string}\}\}$, so

$$f(V \wedge W) = \{a \mapsto \{\texttt{float}, \texttt{string}\}, b \mapsto \{\texttt{float}, \texttt{string}\}\}.$$

   This differs from $f(V) \wedge f(W)$.