Recitation 3: Dataflow Analysis Solutions 24 September

Introduction to Dataflow Analysis

Dataflow Analysis is used to derive information about the dynamic behavior of the program by only examining the static code. We can use a dataflow framework to answer the question "Is it **legal** to perform an optimization?" Generally, a dataflow operation is defined by two parameters:

1. The direction of flow (forward/backward)
2. The function used to combine In/Out sets (May/Must)

	Union (may)	Intersection (must)
Forward	Reaching Defs	Available Expression
Backward	Liveness Analysis	Very Busy

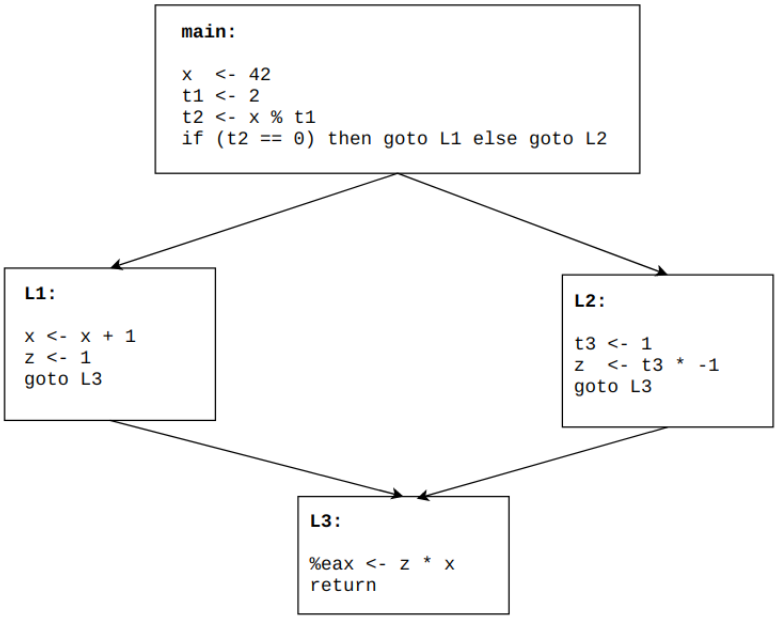
Liveness Analysis (Backward May)

Liveness analysis is one instance of a dataflow problem; specifically, it is a "backward may" dataflow problem. What this means is that the code will be iterated over in reverse order, propagating information from successor to predecessor. The "may" part refers to the fact that we union all the information from the successor to pass to the predecessor. A "must" operation refers to taking the intersection of all the information.

```
1  main:
2    x <- 42
3    t1 <- 2
4    t2 <- x % t1
5    if (t2 == 0) then goto L1 else goto L2
6  L1:
7    x <- x + 1
8    z <- 1
9    goto L3
10 L2:
11   t3 <- 1
12   z <- t3 * -1
13   goto L3
14 L3:
15   %eax <- z * x
16   return
```

Checkpoint 0

Analyze the above program (in the context of backward dataflow analysis) to determine the gen and kill sets for each of the basic blocks. For simplicity, the control flow graph is given below.



Solution:

```
main: gen: {},     kill: {x, t1, t2}
L1:   gen: {x},    kill: {x, z}
L2:   gen: {},     kill: {z}
L3:   gen: {x, z}, kill: {eax}
```

Worklist Algorithm

The worklist algorithm is the standard way to perform dataflow analysis on a program's CFG. The template remains the same for forward and backward analysis, as well as may versus must. The following algorithm is for **backwards may** dataflow:

```
In(s) = Empty set for all s
Worklist = all statements (ideally in postorder traversal order)
while Worklist is not empty {
   take some s from Worklist
   Out(s) = union In(s') for all s', successor of s
   temp = Gen(s) union (Out(s) – Kill(s))
   if (temp != In(s)) {
      In(s) = temp
      Worklist = Worklist union (all predecessors of s)
   }
}
```

Key:

Red - Things to toggle for backward/forward flow (In -> Out, successor -> predecessor, postorder -> reverse postorder, etc)

Blue - Things to toggle for may/must functions (union -> intersect, empty set -> all facts)

Checkpoint 1

Perform backward may analysis on these blocks to determine the In and Out sets for each basic block.

Solution:

```
main: in: {},     out: {x}
L1:   in: {x},    out: {x, z}
L2:   in: {x},    out: {x, z}
L3:   in: {x, z}, out: {}
```

Checkpoint 2

Finally, deconstruct the In and Out sets for each block to calculate the Live In and Live Out sets at each line

Solution:

```
main:
Line 2: in: {},       out: {x}
Line 3: in: {x},      out: {x, t1}
Line 4: in: {x, t1},  out: {x, t2}
Line 5: in: {x, t2},  out: {x}

L1:
Line 7: in: {x},      out: {x}
Line 8: in: {x},      out: {x, z}
Line 9: in: {x, z},   out: {x, z}
```

```
L2:
Line 11: in: {x},     out: {x, t3}
Line 12: in: {x, t3}, out: {x, z}
Line 13: in: {x, z},  out: {x, z}

L3:
Line 15: in: {x, z},  out: {eax}
Line 16: in: {eax},   out: {}
```

Lab 2 Hints

- We recommend implementing a dataflow analysis frameworks. This will be crucial for numerous optimizations including partial redundancy elimination, deadcode elimination, and copy propagation. Using a control flow graph with basic blocks will speedup your dataflow analysis and is necessary for SSA.
- Construct the CFG by constructing basic blocks from instructions - while it is possible to perform dataflow over each instruction as a node, this is significantly slower.
- Optimal worklist ordering for backwards flow is postorder and for forwards flow is reverse postorder (converges faster)

- Consider decoupling your instruction representation from your dataflow, liveness, and register allocation code to some degree. Obviously you can add a lot of complexity by going overboard with this, but you will add many new instructions to your compiler over its lifespan and you will still want dataflow, liveness, and register allocation to work.