15-745 Homework 2

Haojia Sun (haojias), Yikang Cai (dcai)

2.0 Iterative Dataflow Analysis Framework

1. Overview of the Template Framework `dataflow.h`

The framework is a templated class `DataflowAnalysis<Element, bool Forward>`, where `Element` is the analysis unit (e.g., expressions for available expressions, variables for liveness) and `Forward` specifies the direction (default forward; liveness is backward). This design makes the framework reusable: clients only need to define `Element`, the meet operator, and the transfer function, while the framework handles iterative propagation.

2. Handling Aliases with PHI Nodes

The first step in my implementation was to address the issue of aliasing that arises from PHI nodes in SSA form. To do this, I constructed a `DenseMap<Value*, Value*> aliasMap` that records the canonical representative of each value. I then iterated over all instructions in the function and whenever I encountered a `PHINode`, I mapped each incoming value on the right-hand side of the PHI to the result of the PHI instruction itself. This unified different SSA names that represent the same logical value. I also defined a helper function `findRepresentative`, which recursively follows these mappings until it finds the canonical representative. This function is used throughout the rest of the analysis to prevent duplicates in the set of expressions.

3. Expression Universe and Offset Mapping

Here, `Element`s are the target of the analysis (expression in available expression pass and variables in liveness pass). At each program point, the result is represented as a `BitVector`. The static function `createBitVectorOffsetMap` scans all instructions (after mem2reg) and uses a callback `getElementsFromInstruction` to extract relevant elements and assign a monotonically increasing integer as their offset in `BitVector`. Each unique element is assigned a unique bit offset, producing a compact `DenseMap<Element,int>` plus a reverse map for printing.

4. Meet Operator and Transfer Function Abstraction

To keep the framework general, both the meet operator and the transfer function are passed in as function objects when constructing the `DataflowAnalysis`. The meet operator is defined as `std::function<BitVector(const BitVector&, const BitVector&)>`, which allows clients to specify, for example, intersection for available expressions or union for liveness.

The transfer function is defined as `std::function<BitVector(BitVector, Instruction*)>`, which sets the bits for elements in the GEN set and unsets elements in the kill set. Aside from this, we also support passing in block-level transfer functions for more flexibility.

5. Iterative Algorithm Structure

The core of the framework lies in the `analyze` function, which performs the standard **iterative dataflow algorithm**. First, it computes a traversal order over the basic blocks using LLVM's postorder iterator. For forward analyses, this traversal order is reversed to give reverse postorder (RPO). For backward analyses, the traversal is left as-is. The framework maintains a `ResultMap` from each instruction to its current OUT set (IN set if backward analyze), initialized to top of semi-lattice depending on the analysis. For forward analysis, it then enters a fixed-point loop where, in each iteration, it recomputes IN sets for each block by applying the meet operator to the OUT sets of its predecessors (or successors in backward mode). After computing IN, it applies the transfer function to obtain the new OUT. If any block's OUT set changes compared to its previous value, the algorithm marks that progress has been made and continues to another round. This process repeats until convergence.

2.1 Code Implementation 1: Available Expressions `available.cpp`

Here are the steps that I did to implement the availability analysis using our framework.

1. Collecting All Expressions

Available expressions are defined over expressions, so I traversed all instructions and created `Expression` objects for each `BinaryOperator`. Using `createBitVectorOffsetMap`, I assigned each unique expression a bit offset, producing both `elementToOffset` and its reverse. This mapping fixes the dimension of bitvectors for GEN, KILL, IN, and OUT.

2. Defining the Meet Operator

In available expressions analysis, an expression is available only if it appears on all paths. Thus, the meet operator is set intersection, implemented as a bitwise AND across predecessor OUT sets.

3. Implementing GEN and KILL Functions

The next component of the analysis was to compute the GEN and KILL sets for each instruction. In the GEN function, I canonicalized the expressions' operands using `findRepresentative`, and looked up its index in the universal set. Before generating the expression, I checked whether the current instruction's LHS variable invalidated the expression, for instance in cases like `B = B + C`. In addition, I scanned subsequent instructions in the same basic block to determine whether a **later assignment** to one of the operands would kill the expression. If neither of these conditions was true, the expression was marked as generated by setting its corresponding bit. For the KILL function, for a given instruction, I identified its LHS variable, canonicalized it, and if the LHS appeared as an operand in any expression, that expression would be in the KILL set.

4. Defining the Transfer Function

Once the GEN and KILL functions were defined, I implemented the transfer function for basic blocks. I began with `out = in`, initializing the block's OUT set to its IN set. Then, for **each instruction** in the block, I computed its GEN and KILL sets, and updated the OUT set according to the standard equation: `OUT = (IN - KILL) ∪ GEN`. The important detail in my implementation was that I applied this update at the granularity of **individual instructions** rather than at the whole-block level. This allowed me to print the availability information immediately after each instruction, which the assignment required, instead of only once per basic block.

5. Setting up the Dataflow Analysis

With all the components ready, I instantiated the `DataflowAnalysis<Expression>` object provided in the framework. I passed in my meet operator (intersection), transfer function (instruction-level OUT updates), the bitvector size, and the initialization values. Specifically, **the entry block was initialized with the empty set**, since no expressions are available before the start of the program, while **the OUT sets of all blocks were initialized to the universal set**. I then invoked `analyze(F, offsetToElement)`, which executed the forward dataflow analysis iteratively until convergence.

6. Printing the Results

Finally, once the analysis **reached a fixed point**, I printed the results. For each basic block, I first computed its **IN** set as t**he meet (intersection) of the OUT sets of all its predecessors**. This captures the availability of expressions at the block's entry. I printed this IN set, then iterated through each instruction, applying its GEN and KILL sets to update OUT. After each instruction, I printed the resulting OUT set. This way, the final output showed the availability of expressions at every program point, **both at block entry and immediately after every instruction**, as required by the assignment.

2.2 Liveness

2.2.1 Pass Description

- Meet Operator: Union
- Direction: Backward
- `IN` of each basic block: Empty Set.
- `Exit`: Empty Set

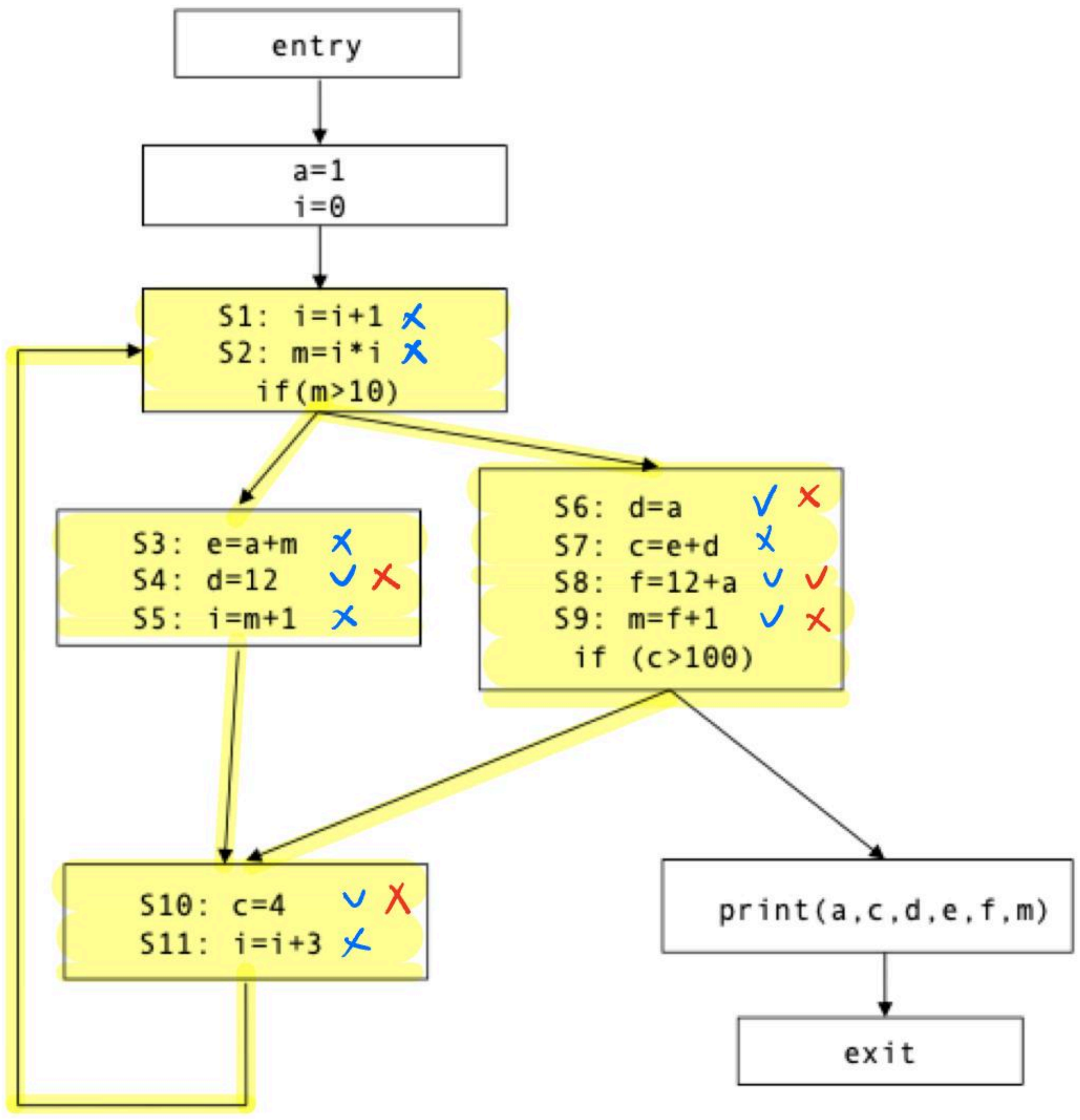
In the code, the data flow analysis is initialized as:
`DataflowAnalysis<Var,false>(setUnion,transferFunction,numVar,false,false)`
Here, `Var` is a wrapper around `Instruction` that provides better formatted printing.

2.2.2. Transfer Function

- Kill Set: If an instruction defines a variable, that variable is removed from the live set.
- Gen Set: Variables used by an instruction are added to the live set. There are some special cases:
    - PHI nodes: Handle incoming values specially since they represent different values from different predecessorsConditional branches:
    - The branching condition variable must be live.

The transfer function is given by `IN[inst] = (OUT[inst] - KILL[inst]) ∪ GEN[inst]`, where KILL removes defined variables and GEN adds used variables.The analysis runs backward through the CFG using reverse post-order traversal, iterating until convergence to find the fixed-point solution for liveness information. Note that since the IR is in SSA form, we don't need to handle cases like `a=a+1`.

3.1 Loop Invariant Code Motion



(a) List the instructions that have loop-invariant computations in their RHS.

Firstly, we need to find the header, the tail and the back edge.

Header: the basic block containing S1 and S2

Tail: the basic block containing S10 and S11

Back edge: From S10, S11 to S1, S2

So the natural loop will be {S1-S11}.

The results of loop-invariants are: {S4, S6, S8, S9, S10}. Below is the analysis:

S1: “i” is both on the LHS and the RHS, so “i” is changing and being redefined. Not loop invariant.

S2: the operand is “i” which is changing (see S1). Not loop invariant.

S3: Since the operand “m” is defined based on “i” (see S2), it is not loop invariant

S4: 12 is constant, not changing. It is loop invariant.

S5: Since the operand “m” is defined based on “i” (see S2), it is not loop invariant

S6: Since the operand “a” is defined outside the loop and does not change in the loop, it is loop invariant.
S7: Since the operand “e” is defined based on changing “m” (see S3), it is not loop invariant.
S8: Since “a” is defined outside the loop and does not change, and 12 is constant, it is loop invariant.
S9: “f” is not defined in the code (unspecified behavior) and does not change. It is loop invariant.
S10: 4 is constant, not changing. It is loop invariant.
S11: Since the operand “i” is changing, it is not loop invariant.

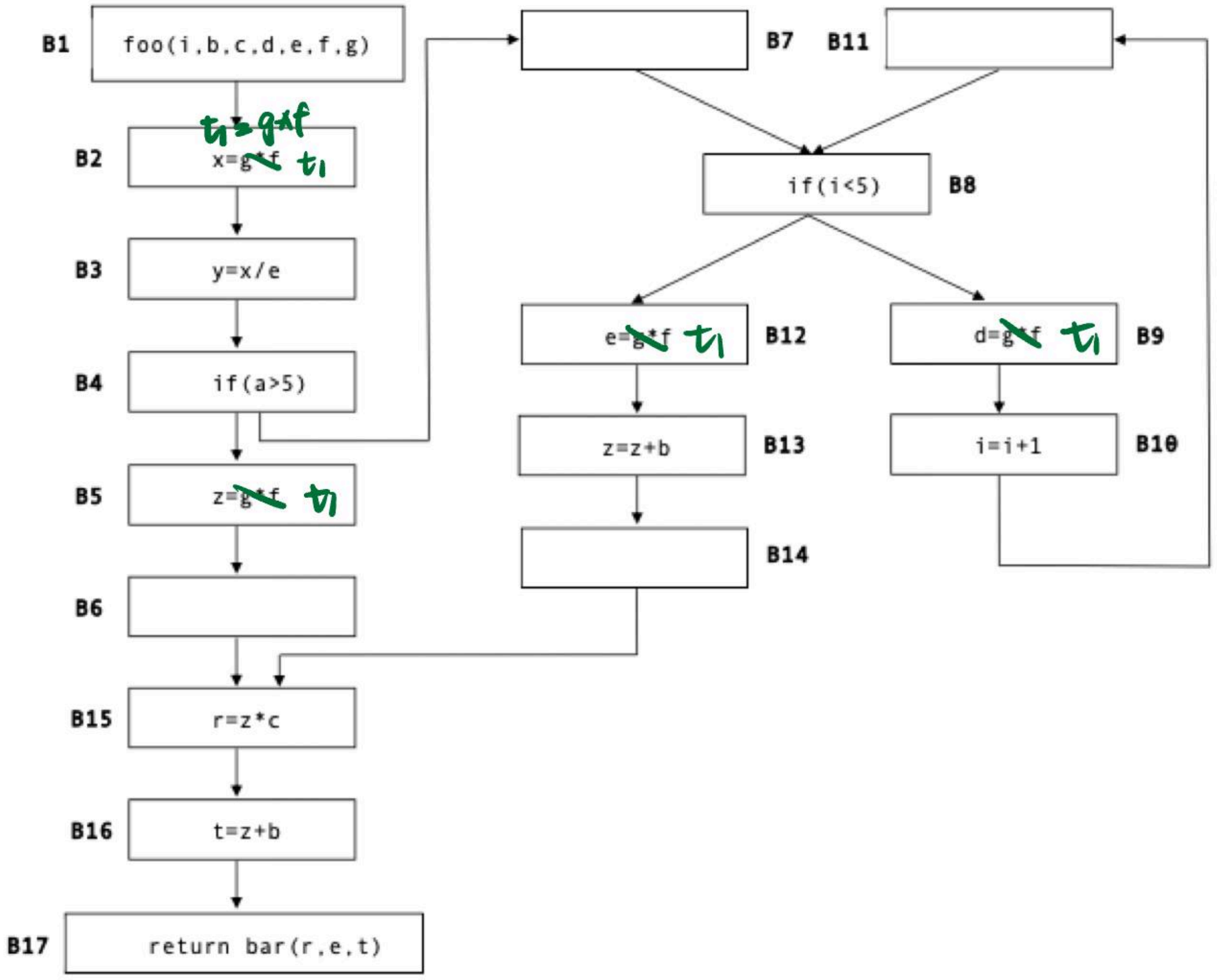
(b) Indicate if each loop-invariant computation can be moved to the loop preheader, and give a brief justification.
Only S8: f = 12 + a can be moved to the loop preheader. Below is the analysis.
S4: On the LHS, “d” is defined both in S4 and S6. If we move d=12 to the loop preheader, and then go inside the loop {S1, S2} and follow the path on the right {S6-S9}, and then we go to the tail {S10, S11}, and then go inside the loop again, and follow the path on the left {S3-S5}, here “d” will become “a”, which is not 12. So it violates the rule of “in blocks that dominate all the exits of the loop”. This will go wrong, so we cannot move S4 to the preheader.
S6: Same reason as S4. Because “d” is defined differently in both S4 and S6. It is not in blocks that dominate all blocks in the loop that use the variable “d”. It violates the rule of “in blocks that dominate all the exits of the loop”.
S8: Since “f” is not re-defined inside the loop (assign to variable not assigned to elsewhere in the loop), and both operands “a” and 12 are unchanged. It means that it satisfies the rule of “in blocks that dominate all the exits of the loop” and “in blocks that dominate all blocks in the loop that use the variable assigned”.
S9: Because on the LHS, “m” is changing within the loop. If we move it to the loop preheader, the value of “m” will be overwritten in other blocks (e.g., S2: m=i*i) inside the loop.
S10: Because “c” is defined twice {S7, S10} inside the loop. If we move S10 to the loop preheader, c=4 might be overwritten by S7 (c=e+d) if we follow the path on the right.

3.2 Lazy Code Motion

Basic Block	Location	Anticipated	Available	Earliest	Postponable	Used	Latest
B1	IN	{g*f}	{}	{g*f}	{g*f}	{}	{}
	OUT	{g*f}	{}	/	{g*f}		
B2	IN	{g*f}	{}	{g*f}	{g*f}	{g*f}	{g*f}
	OUT	{g*f, x/e}	{g*f}	/	{}		
B3	IN	{g*f, x/e}	{g*f}	{x/e}	{}	{x/e}	{x/e}
	OUT	{g*f}	{g*f, x/e}	/	{}		
B4	IN	{g*f}	{g*f, x/e}	{}	{}	{}	{}
	OUT	{g*f}	{g*f, x/e}	/	{}		
B5	IN	{g*f}	{g*f, x/e}	{}	{}	{g*f}	{}

	OUT	{z+b, z*c}	{g*f, x/e}	/	{}		
B6	IN	{z+b, z*c}	{g*f, x/e}	{z+b, z*c}	{}	{}	{}
	OUT	{z+b, z*c}	{g*f, x/e}	/	{z+b, z*c}		
B7	IN	{z+b, g*f}	{g*f, x/e}	{z+b}	{}	{}	{}
	OUT	{z+b, g*f}	{g*f, x/e}	/	{z+b}		
B8	IN	{z+b, g*f}	{g*f, x/e}	{z+b}	{z+b}	{}	{}
	OUT	{z+b, g*f}	{g*f, x/e}	/	{z+b}		
B9	IN	{z+b, g*f, i+1}	{g*f, x/e}	{z+b, i+1}	{z+b}	{g*f}	{}
	OUT	{z+b, g*f, i+1}	{g*f, x/e}	/	{z+b, i+1}		
B10	IN	{z+b, g*f, i+1}	{g*f, x/e}	{z+b, i+1}	{z+b, i+1}	{i+1}	{i+1}
	OUT	{z+b, g*f}	{g*f, x/e}	/	{z+b}		
B11	IN	{z+b, g*f}	{g*f, x/e}	{z+b}	{z+b}	{}	{}
	OUT	{z+b, g*f}	{g*f, x/e}	/	{z+b}		
B12	IN	{z+b, g*f}	{g*f, x/e}	{z+b}	{z+b}	{g*f}	{}
	OUT	{z+b}	{g*f}	/	{z+b}		
B13	IN	{z+b}	{g*f}	{z+b}	{z+b}	{z+b}	{z+b}
	OUT	{z+b, z*c}	{g*f}	/	{}		
B14	IN	{z+b, z*c}	{g*f}	{z+b, z*c}	{}	{}	
	OUT	{z+b, z*c}	{g*f}	/	{z+b, z*c}		
B15	IN	{z+b, z*c}	{g*f}	{z+b, z*c}	{z+b, z*c}	{z*c}	{z*c}
	OUT	{z+b}	{g*f, z*c}	/	{z+b}		
B16	IN	{z+b}	{g*f, z*c}	{z+b}	{z+b}	{z+b}	{z+b}
	OUT	{}	{g*f, z+b, z*c}	/	{}		
B17	IN	{}	{g*f, z+b, z*c}	{}	{}	{}	{}
	OUT	{}	{g*f, z+b, z*c}	/	{}		

(d) Complete the final pass of lazy code motion by inserting and replacing expressions. Provide the finished control-flow graph, and label each basic block with its instruction(s).


B1
foo(i,b,c,d,e,f,g)
t1 = g*f
B2
x=g*f t1
B3
y=x/e
B4
if(a>5)
B5
z=g*f t1
B6
B7
B11
B8
if(i<5)
B12
e=g*f t1
B9
d=g*f t1
B13
z=z+b
B10
i=i+1
B14
B15
r=z*c
B16
t=z+b
B17
return bar(r,e,t)