CSC410

HW 4

Due on Monday October 30, 2017 before 11am

Submit on Markus.

About Homework 4

Homework 4 is the second part of the homework on data flow analysis, which focuses on its theory. For this part, all answers should be written in a PDF file and submitted on Markus. You will submit a single PDF file named `hw4.pdf`.

This homework looks long, but it is rather simple and has a short solution (around 2 pages). My estimate for the time required to do this exercise is under 4 hours on average (done by 1 student who understands the data flow analyses lecture material). Naturally, this excludes the time that you may need reminding yourselves of what you have learned in the past two lectures. That said, do not start on the Sunday before the deadline. Especially, if you may have questions. And, do not expect answers from the teaching team on Piazza during the weekend. We have families, laundry, and shopping to do over the weekends!

Problem 1

A `Bit Vector Framework` is a special instance of a *monotone* framework where

- $L = (\mathcal{P}(D), \sqcap)$ for some *finite* set D and where $\mathcal{P}(D)$ is the powerset of D and $\sqcap$ is either set union ($\cup$) or set intersection ($\cap$), and
- $\mathcal{F} = \{f : \mathcal{P}(D) \to \mathcal{P}(D) |\ \exists G, K \subseteq D :\ \forall S \subseteq D,\ f(S) = (S \cap K) \cup G\}$.

You will hand in a pdf containing the written answer to the following questions:

(a) (15 points) Show that *Live Variable Analysis* is a bit vector framework (this is mostly trivial).

(b) (25 points) Show that all bit vector frameworks are distributive frameworks.

(c) (10 points) Provide an example of a simple distributive framework that is **not** a bit vector framework to prove (by counterexample) that the converse of (b) is not true.

Note that you are not being asked to remember something you have seen for this before, but to make up a small example that satisfies this condition. The example does not have necessarily have to correspond to a real widely used analysis. Anything that fits the definition of an analysis over a program will be acceptable.

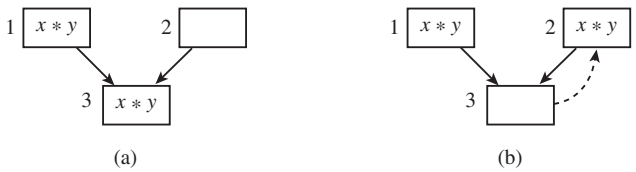
This is a variation of exercise 2.9 in the book.

Problem 2

This problem is long, but it is very simple and has a rather short solution. It is long, because I wanted to simplify it by walking you through the solution steps, and give some examples. The solution to each part is short and simple. You just need to be careful with part (c).

Simpler data flow analyses can be composed to make a more sophisticated one. Let us try to do this together to learn how it is done. We pick a (more) sophisticated data flow analysis, called *partial redundancy elimination* (PRE) as our target.

Available expressions analysis (see page 39 of your assigned reading) is a simple analysis that captures the redundancy of an expression along all paths (i.e. complete redundancy). However, in some situations, an expression may be only partially redundant along, that is redundant along some path, but not redundant along some other. Consider the diagrams below:



The expression $x * y$ is redundant on the path from 1 to 3, but not from the path from 2 to 3 (in (a)). The expression is clearly not *available* (in the sense of available expressions) at node 3. But, if we insert $x * y$ into node 2, then it becomes completely redundant in node 3. The precise criterion for PRE is quite subtle; intuitively, the expression must be partially redundant at a node (say i) and some predecessors of i must exist in which the expression can be inserted without changing the semantics of the program.

The principle of performing partial redundancy elimination is to introduce new computations of the expression at points of the program chosen in such a way that the partially redundant computations become redundant and can hence be deleted. In the example above, the identification of node 2 is the key to deleting the redundancy at node 3.

To make up a *partial redundancy eliminator*, we introduce, and then combine a few simple (basic) analyses through the following steps.

(a) (10) An expression is partially available at a program point ℓ if it there is at least one control flow path to location ℓ along which the expression is computed, and none of its operands have been overwritten since. Note the contrast against available expressions. With a minor change to the formal definition of *available expressions analysis*, you can get a definition for *partially available expressions*. Make this precise, by providing the constraints (equations) that formally define *partially available expressions*. Note that we are asking you to cut-and-paste the formal definition form *available expressions* and make a few simple changes to them to get the new analysis. Do not overthink this. It is trivial.

(b) *Anticipability* is the dual of *availability*. An expression is *anticipated* at a location ℓ if it computed along all (future) paths from ℓ, and none of its operands are modified on each path until the first computation of the expression. This is what is named as *very busy expressions analysis* in your text book. There is nothing to do for part (b), other than reminding yourself of this definition to use it in (c).

(c) (30) *Placement Possible Analysis* aims to determine predecessors of a statement, which contains partial redundancies, where a new computation may be introduced. Define a data flow analysis for *placement possible* by providing the property space, the transfer functions, and the initialization information. The idea is that you will use *available expressions*, *very busy expressions*, and *partially available expressions* in your data flow equations for *placement possible* analysis.

Note: this is intentionally loosely defined. Making this definition precise is part of the exercise.

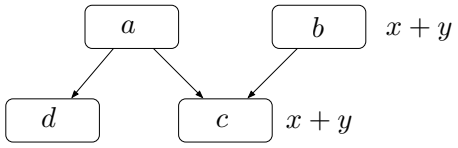
(d) (10) To wrap this up, provide two equations defining the two key sets:

- $insert_\ell$: the set of the expressions that must be added as new computations at location ℓ. The new computation would be inserted at the exit from the node (i.e. after all the statement(s) already there in the node).
- $delete_\ell$: the set of redundant expressions that may now (after the inserts above) be deleted from the location ℓ.

Note that the two sets above should be computable from the result of analyses in the previous part. We are not setting up new data flow analyses to compute them in part (d). At this point (specially after part (c)), all the ingredients should be there for you to compute the *insert* and *delete* sets using simple set operations over the results of the analyses that you have already defined.

For those following things carefully, there should be a question on your mind at this point: are the answers to this analysis really unique? Well, not exactly. This is a compiler optimization technique that was introduced to subsume a few other simpler optimizations. The goal with its design was to cover the simpler cases, and do more but not "do everything!". So, your design should have the following properties:

- It should do no harm. At the end of the transformation no path of the graph can contain **more** computations of an expression than it contained before. This may seem sensible, but has serious consequences. For example, an expression cannot be inserted in a place where now it will become known to a path where it was not known before. For example, in the figure below:

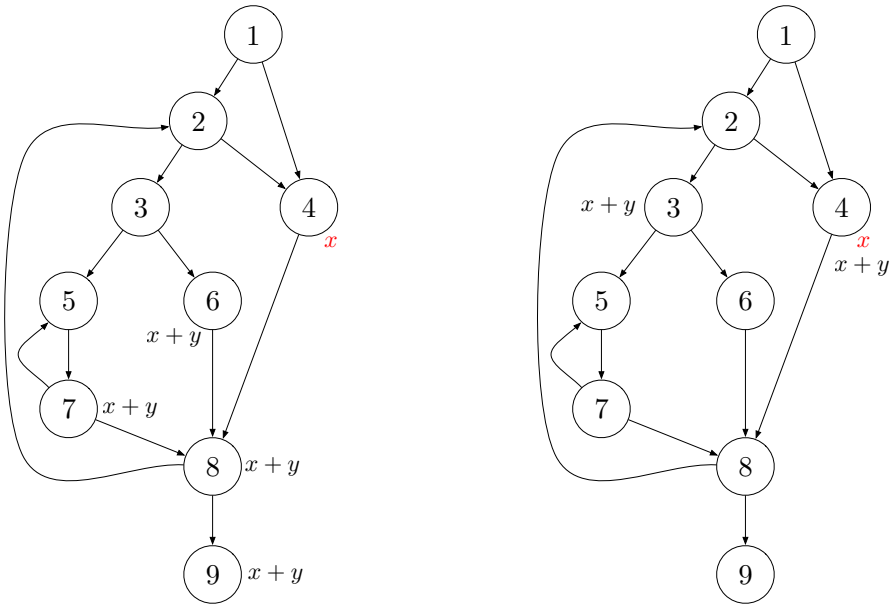


  the redundancy at node c cannot be eliminated because moving the computation to a will introduce the expression $x + y$ to the path $a \to d$ where it was not available before.

- A partial redundancy is being suppressed by possibly introducing new partial dependencies. The recursive nature of the problem means that a global optimization is best performed during runtime. You do not have to worry about the new partial dependencies that your inserts are creating, as long as you are maintaining the property from the previous bullet point.

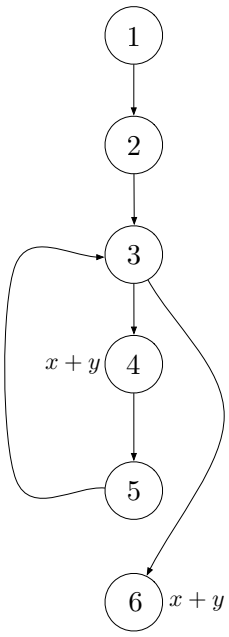
- Your analysis does not have to always insert the minimal number of computations. Following to the point above, new inserts may be redundant or partially redundant and can be replaced by fewer inserts in another round of this algorithm. Do not worry about minimality.

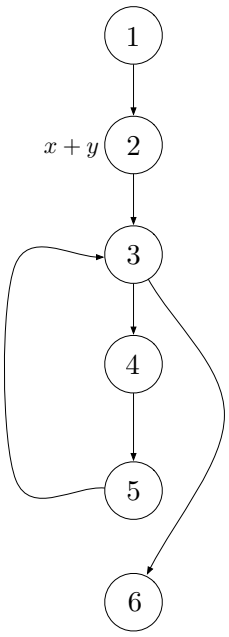
- One can design the analysis to do nothing and still satisfy the two conditions above. So, here are a couple of examples that your analysis should be able to handle by performing the illustrated transformations. Below, the graph on the left should be transformed to the graph on the right (the red x indicates that node 4 modifies x):



  Moreover, the expression $x + y$ should be in the placement possible *in* sets of nodes 1, 2, 3, and 4, and in the placement possible *out* sets of nodes 3, 4, 5, 6, 7. Naturally, the resulting graph says that the *insert* set of only nodes 3 and 4 shall end up including the expression $x + y$, and the *delete* sets of 6, 7, 8, and 9 should include $x + y$.

  Another example is the following generic optimization which is referred to a *loop invariant* optimization, which your partial redundancy elimination should subsume. In the classical optimization, computation of an expression is removed from a (nested) loop while it is both invariant in the loop and can be anticipated on entry. The diagram below graphically illustrates this transformation from the redundant version (left) to the optimized version (right):


1
2
3
$x+y$
4
5
6
$x+y$


1
$x+y$
2
3
4
5
6