CSC410

Assignment 2

Due on October 19, 2023

Submit on Markus.

This homework is shorter than it looks. The solutions are mostly straightforward and short. The two-week timeline is just there for your convenience and is not an indication of how much time is required.
On Markus, submit:

- A file called `a2.pdf`, which will contain the cleanly typeset solutions to Problem 1, and the pen-and-paper description of the analyses for all the other problems as indicated. In each case, the description should be complete in a way that another person can read it and implement the analysis.
- The files that are listed under **Implementations** at the end of this document.

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 (typed) 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, i.e. the transfer functions are distributive.

(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 that you have seen before, but rather to make up a small example that satisfies this condition. The example does not have to necessarily correspond to a real-world analysis. Anything that fits the mathematical definition of an analysis over a program will be acceptable.

Problem 2

In many programming languages, variables can be declared and assigned values separately. The first assignment to a previously declared variable is referred to as initialization. Accessing uninitialized variables in a program usually indicates a bug. The purpose of this problem is to implement a dataflow analysis that enables the detection of uninitialized variable accesses. More precisely, your analysis will identify *definitely initialized variables*for each program label.

A variable v is definitely initialized at label ℓ if all CFG paths to ℓ contain an assignment to v.

If a label uses a variable that is not in the *definitely initialized* set, then it may be using an uninitialized variable.

- (5 points) Fully specify your analysis in the PDF file by describing the domain (the lattice) and the transfer functions.

- (20 points) Implement your designed analysis in Tundra.

Hint: the pass should be conservative, and collect the smallest set of *definitely initialized* variables. There may be false positives. Pay careful attention to what this implies for loop headers, which have predecessors from both outside the loop and from the loop body.

Problem 3

In a *Integer Signs Analysis* , we model all negative integers by the symbol $-$, zero by the symbol 0, and all positive integers by the symbol $+$. Integer sign analysis then tracks whether each integer in the program is positive, negative, or zero at a given program label. The results of this analysis can be used to optimize a program or to circumvent errors like using a negative index into an array or division by zero. Below, D and D' stand for the set of datafow facts, and V stands for the set of program variables.

(a) (5 points) First, we need to define the corresponding lattice (in PDF only). Take D to be $V \to \mathcal{P}(\{-, 0, +\})$ and, based on this D, fully specify the lattice for the property space by giving the meet operation, the top, the bottom, and the corresponding partial order relation.

(b) (5 points) We are now ready to define sign analysis as an instance of a *monotone framework* by defining transfer functions. Start by defining your transfer functions in the PDF.

(c) (15 points) Implement the entire framework you have defined on paper in parts (a) and (b) in Tundra.

You should create three functions, `abstract`, `merge`, and `sign`. The `abstract` function takes an expression and `AnalysisType` as input, and recursively visits the operands of each binary or unary expression before applying `merge`. The `merge` function takes the operation and operands as input, and uses sign to calculate the proper lattice element: for example, `1 * 1 == + * + == +`. Think carefully about choosing the correct `AnalysisType`. Additionally, consider how `*`, `+`, and unary `-` can be overloaded methods in Python, and think about how overloading can help merge lattice elements. Finally, abstract has two important base cases that rely on `AnalysisType` and `sign`.

(d) (5 points) Now that you have your analysis working, let us try to tweak it a bit to get a more precise analysis out of this. Take D' to be $\mathcal{P}(V \times \{-, 0, +\})$ and, using D', redo parts (a) and (b). Think carefully about what the change of domain means for the type of facts that we can infer for the program now before you start on the path of analysis design.

(e) (10 points) Modify your implementation from part (c) to get an running implementation in Tundra for this case.

(f) (5 points) Is there any difference in precision obtained by the two approaches? Can you justify your answer? (answer in the PDF)

Notation reminder: $\mathcal{P}(S)$ is the powerset of set S, which is the set of all subsets of S.

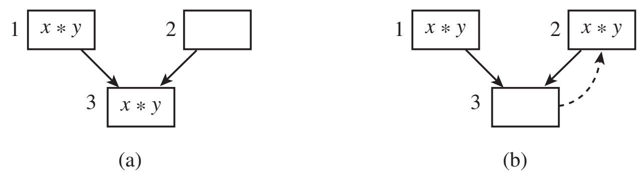
Graduate Problem

This problem is long, because it is written in a way that it would walk you through the solution steps in detail, and give some examples. The solution to each part is short and simple, and parts (a) and (b) are already known to you and are given as part of the roadmap to complete the solution, and opportunity for partial credit.

The Analysis

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 will be using the parts we built in **Problem 2** in completing this exercise.

Available expressions analysis is a simple analysis that captures the redundancy of an expression along all paths. However, in some situations, an expression may be available through some path(s), but not all paths. Consider the diagrams below:



The expression $x * y$ is available 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 available in node 3 and does not have to be recomputed. We can then eliminate its computation as an optimization. The precise criterion for this optimization is quite subtle; intuitively, we identify an expression that is being computed at a node i, and is available on some paths to it, and then by adding its computation to some remaining predecessor(s) of i, we make the expression fully available. This should all be done without changing the semantics of the program.

Let us combine a few simple (basic) analyses through the following steps to perform this task.

(a) (5 points) Recall the formal definition of *available expressions analysis* and either write the analysis yourself or copy it from your tutorial material. Note, you need to slightly extend the tutorial version to get it working for you here. No need to submit anything in the PDF for this one.

(b) (10 points) An expression is partially available at a program point ℓ if 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, and implement *partially available expressions*. Do not overthink this. It is straightforward. 5 out of 10 points are dedicated to what you submit in the PDF.

(c) (30 points) Ultimately, we would like to decide where we can add new computation of some existing expressions, so that we can then delete some redundant computations. To prepare to define what to add and what to delete, we perform the main analysis that identifies all candidate locations for the placements of these new computations.

Potential Placement Analysis aims to determine all locations where an expression may be added. To do this, we need to make precise what it means to "add the computation of an expression to a location". Let us fix this to mean that we add a second assignment statement to that program location, **after** the existing statement at that location, which computes the expression and stores it in a new (fresh) program variable.

The idea is that an expression **may** be added to location ℓ'' if it is partially available at location ℓ' and there is control flow path from ℓ'' to ℓ'. Additionally, we need to consider the following:

- What does the statement in (arbitrary location) ℓ contribute in terms of the information about placement of possible expression?
- If a computation **may** be placed at (the end of) location ℓ, then under what conditions can it *move up* to the beginning of ℓ?
- Considering that we have decided to only place new computations at the end, if a computation *can be placed at the beginning of a location* ℓ, then we really want to pass it on to some of its predecessors. Which predecessors of ℓ may also (possibly) compute it at the end of their locations? or need not compute it because it is already available to them?

Define a data flow analysis for *potential placement* by designing the property space, the transfer functions, and the initialization information. The idea is that you will use *available expressions* and *partially available expressions* in your data flow equations for *potential placement* analysis. Like any other analysis, you may use local sets to capture the effect of the statement at location ℓ on the results as well. Implement your analysis. 5 out of 30 points are dedicated to what you submit in the PDF.

Hint: If you carefully consider the items above, you will realize that unlike the classic analyses that you have seen in class and tutorials, this analysis does not solely move forward or backward. It sort of moves both ways, because it updates the information in each node by looking at some information from its predecessors and some information from its successors. But, that is fine. Convergence is still guaranteed if the transfer functions are monotone.

(d) (15) 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 the statement in the node). Loosely put, we prefer to insert things earlier rather than later, and we only recompute expressions that are not available.
- *delete*$_\ell$: the set of redundant expressions that may now (after the inserts above) be deleted from the location ℓ. An expression is redundant if it is computed here but it could be computed earlier.

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 (d)), 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. 5 out of 15 points are dedicated to what you submit in the PDF.

Discussion

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:

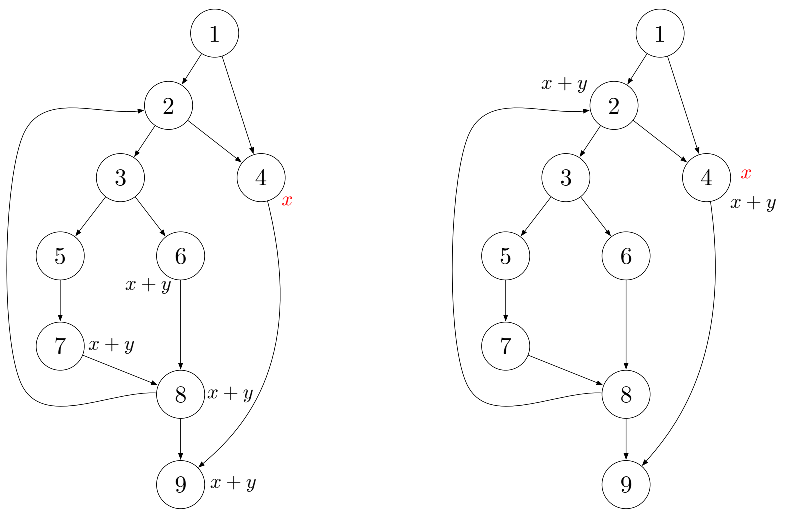
$$\begin{tikzcd}
& a \arrow[dl] \arrow[dr] & & b \arrow[dl] & x+y \\
d & & c & x+y &
\end{tikzcd}$$

  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.

- Your insertions would naturally create new candidates to consider for redundancies, and this process seems recursive in nature. You do not need to worry about this. As long as you are maintaining the property mentioned in the previous bullet point, you are free not to consider your "inserts" as part of the code that has to be re-analyzed.

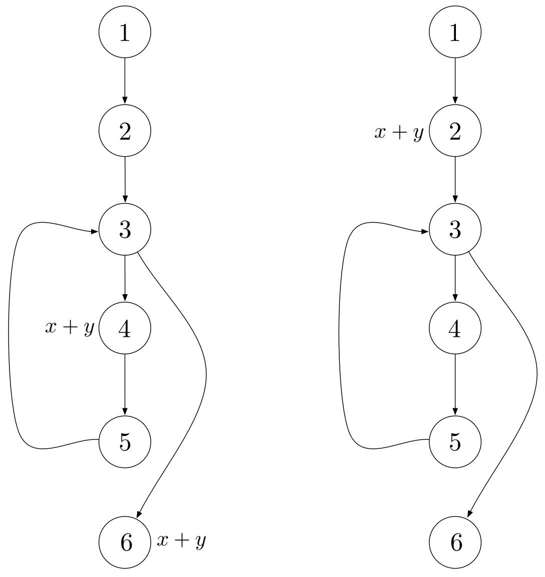
- Your analysis does not have to always insert the minimal number of expressions/computations. Following to the point above, new inserts may lead to decisions about fewer inserts in another round of this algorithm. Do not worry about minimality.

- One can design the analysis to do nothing and still satisfy the two conditions above: no new inserts means "no harm done" and minimality is not a concern! 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, but no other node does):



Moreover, the expression $x + y$ should be in the potential placement *in* sets of nodes 3, 5, 6, 7, 8, and 9, and in the potential placement *out* sets of nodes 2, 3, 4, 5, 6, and 7. Naturally, the resulting graph says that the *insert* set of only nodes 2 and 4 shall end up including the expression $x + y$, and the *delete* sets of 6, 7, 8, and 9 should include $x + y$. Note that the specific control flow graph illustrated above is does not correspond to a program with clean branching and looping. So, in Tundra, you will not get this exact graph to correspond to any program. It just serves as an example here.

Another example is the following generic optimization which yours should subsume. In the classical optimization, computation of an expression is removed from a 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) (under the assumption that no node modifies the variables x or y):



the expression $x + y$ should be in the potential placement *in* sets of nodes 3, 4, 5, and 6, and in the potential placement *out* sets of nodes 2, 3, 4, and 5.

Implementations

Before starting on your implementation, you are strongly encouraged to complete the data flow analysis tutorial exercises and review the tutorial slides. The starter code for the tutorial is very similar to the code used in this assignment; the exercises and slides will teach you how to create and test a new analysis in this framework.

Getting Started

To complete the implementation, you will need Python $\geq$ 3.9. Once you download and unzip the starter code, install the Python dependencies: `pip3 install -r requirements.txt`. Check if it's working: `pytest test/test_cfg.py`. You should see that 9 tests pass.

Analysis Implementation

For each of the analyses you are asked to implement in the problems above, you will submit a separate implementation. For example, for problem 2, create the file `analysis/definitely_initialized.py`. There, define the type `AnalysisType` and the class `DefinitelyInitialized`, a subclass of `Analysis[AnalysisType]`. This subclass should implement the methods `initial_in`, `initial_out`, and `get_new_values`. You are free to add imports to this file, to implement helper methods, and to implement the `__init__` method on this new class, provided that you do not change the class's interface. Test your analysis by running `pytest` on the file `test/test_definitely_initialized.py`. Note that this test file only contains a couple of small tests; you are strongly encouraged to add more tests.

You will follow a similar procedure for the rest of the implementations. The table below lists what we expect to see when you hand your assignment in. Naturally, undergraduates can leave out the files related to the graduate problem.

Class Name	**Analysis File** in `analysis/`	**Test File** in `test/`
DefinitelyInitialized	`definitely_initialized.py`	`test_definitely_initialized.py`
SignAnalysis	`sign_analysis.py`	`test_sign_analysis.py`
PreciseSignAnalysis	`precise_sign_analysis.py`	`test_precise_sign_analysis.py`
AvailableExpressions	`available_expressions.py`	`test_available_expressions.py`
PartiallyAvailableExpressions	`partially_available_expressions.py`	`test_partially_available_expressions.py`
PotentialPlacement	`potential_placement.py`	`test_potential_placement.py`

Deliverables and Grading

Submit the following files:

- `definitely_initialized.py`
- `sign_analysis.py`
- `precise_sign_analysis.py`
- `available_expressions.py`
- `partially_available_expressions.py`
- `potential_placement.py`

You will be graded via automated testing using pytest. Some of the tests have been provided to you already and some tests have been withheld. Your grade in the implementation part of each question will correspond to the percentage of tests that you pass.