CSE 231 Project Report

Default (LLVM) Project

Prateek Khurana, Mansi Malik

1. Overview

The goal of our project was to demonstrate the principles of dataflow analysis that could be used to implement various useful optimizations. The framework operates on a control flow graph, based on a given program's intermediate representation provided by LLVM. For each set of optimizations, we run an optimistic worklist algorithm to apply flow functions to each instruction over and over again until a fixed point is reached. The framework is generic to the optimization that is being run.

Being a team of two, we implemented 3 of the 4 required optimizations. The output for Constant Propagation and Common Subexpression Elimination optimization is a transformed, optimized version of the program. For Range Analysis, no transformation takes place, and the output is a set of warnings shown to the user if there is a possibility of accessing an array index that is out of bounds. Also, as a common output for all three analysis, we display the following output: The lattice information at the beginning and end of each Basic Block; The lattice information at the beginning and end of each instruction.

2. Interface Design

The framework consists of an abstract *Lattice* class and an abstract *FlowFunction* class that can be derived by the various optimizations to define their lattice and flow functions. The algorithm is implemented in a *WorkList* class that is called by any given optimization by passing to it the type of analysis that has to be executed. Based on the type of analysis, the algorithm chooses the type of Lattice and Flow Functions to use in the algorithm. An important point to note here is that the entire algorithm runs on the abstract Lattice and uses abstract Flow Functions. Only the instantiation of lattice and flow functions differ for each analysis, making the algorithm a generic one.

All our analysis require that LLVM's *mem2reg* pass be run on the input program representation before running the analysis, to reduce the number of instructions by promoting memory locations to registers wherever possible, converting the input program into a pure SSA. This pass removes the set of load and store instructions allowing us to run our analysis on the new representation.

One interesting challenge during the interface design was when we realized we cannot use dynamic_cast directly in LLVM to cast a base class into one of its derived classes. LLVM avoids using C++'s built in RTTI (Runtime Type Information). And therefore, we had to set up **LLVM-style RTTI** for our class hierarchy as provided in the programmer tutorial here. This allowed us to use LLVM's isa and dyn_cast methods to check the type and typecast base Lattice class into one of the derived classes at runtime.

Lattice: The abstract Lattice class has two members, bottom and top. These if set represent that the lattice point is bottom or top respectively. It also contains the following virtual methods:

a. *Join(LatticePoint)*: A virtual method to join the lattice point to the input lattice point.

b. *Equals(LatticePoint)*: Returns a bool representing if the lattice point is equal to input lattice point.
c. *printLattice()*: Prints the representation of the lattice point.

All analysis derive this abstract class to define their own representation of lattice, and give a definition to each of the virtual methods above. This representation is described in each analysis discussed.

Flow Function: The abstract FlowFunction class contains virtual methods for the different types of Flow Functions that we aim to apply in our analysis. Each of the analysis derive their flow functions from this class and provide their own implementation of every function.

WorkList: The worklist algorithm works on the abstract Lattice and Flow Functions. The type of Lattice and Flow Function is determined at runtime based on the analysis being run. The algorithm runs on each instruction in a basic block, based on the information at the beginning of each basic block. Flow Functions are called for the various basic block instructions. The updated lattice information at the end of Basic Block updated the information at the each successor edges. If the new information is different from the already stored edge information, the successor is again put on the worklist. This process continues will we reach a fixed point. We had concerns where the height of the lattice was infinite in certain analysis. We discuss such cases, and the solutions with each analysis later in the report.

A pseudo code for the worklist algorithm is presented below:

```
let m: map from edge to computed value at edge
let worklist: work list of nodes
for each edge e in CFG do
     m(e) := initial info
for each node n do
     worklist.add(n)
while (worklist.empty.not) do
     let n := worklist.remove_any;
     let info_in := m(n.incoming_edges);
     let info_out := F(n, info_in);
     for i := 0 .. info_out.length do
       let new_info := m(n.outgoing_edges[i]) join
       info_out[i];
       if (m(n.outgoing_edges[i]) != new_info])
            m(n.outgoing_edges[i]) := new_info;
            worklist.add(n.outgoing_edges[i].dst);
```

The initial info is an empty lattice if no information is available at the beginning of the algorithm.

We considered using templates for Lattice and Flow Functions instead of inheritance. But soon we realized that it unnecessarily made the framework more complex, and the worklist was no longer generic enough. Thus, we switched to inheritance from templates.

3. Clients of Analysis Framework

3.1. Constant Propagation

In constant propagation, we look for variables with known integer values at each point in the program, and propagate them throughout the program. The worklist algorithm returns a map of instructions and the Lattice information at the instruction point. The constant propagation pass uses this information to substitute instructions with constants and remove the resulting dead instructions.

3.1.1. Assumptions

a. Mem2reg pass was run on the IR before running constant propagation
b. The representation of the program is a pure SSA, which is to say that each register in LLVM can have only one assignment.

3.1.2. Lattice Definition

Our implementation of constant propagation is a must-be analysis. That is, for any program point, each variable can have only one value, as opposed to a super-set of all possible values in a may-be analysis. Therefore, our most conservative (top) is an empty set while most optimistic (bottom) is a full set.

Lattice:

$$(D, \bot, \top, \sqcup, \prod, \sqsubseteq) = (2^{\{u \to v \mid u \in Vars \wedge v \in A\}}, \Phi, \{u \to N \mid u \in Vars\}, \cap, \cup, \supseteq)$$

The problem with this lattice is that it can be infinitely high. To tackle this problem we have used a map from each variable/instruction to a single lattice point, and each lattice point has a special representation to imply bottom and top.

3.1.3. Flow Functions Definition

Our input program representation is an SSA that guarantees that all assignments to X are an assignment to a new register. This helps simplifying our flow functions, such that we do not have to remove X in our implementation when its value changes.

Flow Functions:

1. $F_{X:=N}(in) = in - \{X \to *\} \cup \{X \to N\}$
2. $F_{X:=Y}(in) = in - \{X \to *\} \cup \{X \to c \mid Y \to c \in in\}$
3. $F_{X:=Y\ op\ Z}(in) = in - \{X \to *\} \cup \{X \to c \mid Y \to a \in in \wedge Z \to b \in in \wedge c = a\ op\ b\}$
4. $F_{X:=\Phi(Y,Z)}(in) = in - \{X \to *\} \cup \{X \to c \mid Y \to c \in in \wedge Z \to c \in in\}$
5. $F_{merge}(in_1, in_2) = in_1 \cap in_2$

Note: Even though we have included removal {X->*} in our Flow Functions, we have skipped it from our implementation because SSA guarantees that every assignment to X will be performed in a new register. Hence, there would be no previous value while performing a new assignment.
Also, we currently handle only +,-,* and / Binary Operations. Adding more binary operations is a mere task of adding more switch cases in the implementation which is straight forward.

3.1.4. Implementation of Analysis

a. Data Structures

Our lattice element for Constant Propagation is a std: map with (Value*, ConstantInt*) as the (key, value) pair: - map<Value*, ConstantInt*>. The worklist algorithm stored one such lattice point for the beginning and end of each instruction and basic block.

b. Data Types

We have currently designed our analysis to handle only integer constants. However, the analysis can be extended to handle other constants, e.g. float, double, etc. in future, by using a more generic class instead of ConstantInt. The flow function logic would remain the same. The only change would be to look for constants instead of ConstantInts.

c. Handling Special Cases

To implement Constant Propagation, we needed to take into consideration the scenario of a potential "division by zero". Therefore, in our Flow Function to handle binary operations, we have added a check if the denominator is 0 and the operand is divide. We chose not to fold such an operation, and throw a warning to the user for a division by zero. Optimizations for other instructions are not affected.

3.1.5. Transformation

The lattice information computed for every instruction is finally passed on to the Constant Propagation Client. In the client code, we go through the Lattice Point at each instruction, and replace all instances of the key of the lattice point with its value, which is to say that if an instruction has been folded, we replace the instruction/register usage with its corresponding constant value. Since we cannot remove instructions while replacing their instances, we add all instructions whose instances have been replaced in a set, and finally delete those instructions from the transformed program.

3.1.6. Benchmarks

a. divideByZero.cpp

Consider the following C++ code

```
int a = 10;
int b = a+5;
int c = b * 10;
int d = c/0;
printf("%d\n", d);
```

In this code, after running mem2reg followed by Constant Propagation pass, the code is first transformed to fold b to 15 and c to 150. Then the assignment instructions for a and b are removed from the transformed program. Moreover, the pass throws a warning for a divide by zero, and does not attempt to fold the assignment to d.

b. ifTest.cpp

Consider the following C++ code

```
int a;
int x = 10;
int y = 20;
int z = 30;
```

```
if (y>0){
        a = z - y;
}
else{
        a = y - x;
}
printf("%d\n", a);
```

The partial LLVM representation for the above LLVM code is as follows:

```
entry:
%cmp = icmp sgt i32 20, 0
br i1 %cmp, label %if.then, label %if.else
if.then:                                          ; preds = %entry
  %sub = sub nsw i32 30, 20
  br label %if.end
if.else:                                          ; preds = %entry
  %sub1 = sub nsw i32 20, 10
  br label %if.end
if.end:                                           ; preds = %if.else, %if.then
  %a.0 = phi i32 [ %sub, %if.then ], [ %sub1, %if.else ]
```

A particularly interesting transformation to discuss here, after running the Constant Propagation pass on the code above is the transformation of the PHI node, which is as follows:

```
%a.0 = phi i32 [ 10, %if.then ], [ 10, %if.else ]
```

Notice that as per the Flow Function for PHI code, the value for %a.0 should have been replaced by 10, since the constants at both the incoming nodes of PHI are the same. However, we fail to do so in our analysis because the information is not available to us while analyzing the PHI node.

It is interesting to note that when we reach the PHI node, merging of information from the predecessor branches has already taken place, and since the values came from two different branches, the merged information, being the intersection of the two, no longer holds the values.

However, it is save to replace %sub and %sub1 with their corresponding folded value, because any new assignment is guaranteed to take place in a new register, considering the SSA property.

3.1.7. Lessons Learnt

a. We cannot delete an instruction from a representation while replacing its instances in a program. This results in the corruption of the internal representation of the program. Therefore, we need to keep a track of instructions and delete them all later.
b. After spending a lot of time to figure out why the PHI node was behaving the way it was in benchmark b above, we realized we need to consider merging information differently in SSA represented programs. This is a potential future work for our program.

3.2. Available Expressions

In Available Expressions, we look for available expressions of the form A op B, so that the value of such expressions need not be computed again and again.

3.2.1. Assumptions

a. As in other analysis, we assume mem2reg pass is run on the input program representation before running this pass.
b. We do not look for a division by 0, and do not warn the user, because we are not actually folding any expression.

3.2.2. Lattice Definition

The Available Expressions analysis is a must-be analysis. Therefore, as with Constant Propagation, our top is an empty set while bottom is the full set. We handle the possibility of an infinite lattice in a similar fashion, that is, by having a mapping of each instruction with an element of the lattice, and keeping special representations to signify if a lattice element is bottom or top.

Lattice:

$(D, \perp, \top, \sqcup, \prod, \sqsubseteq) = (2^{\{x \to y \mid x \in Vars \wedge y \in Exprs\}}, \Phi, \{x \to Exprs \mid x \in Vars\}, \cap, \cup, \supseteq)$

3.2.3. Flow Function Definition

Our input program representation is an SSA that guarantees that all assignments to X are an assignment to a new register. This helps simplifying our flow functions, such that we do not have to remove X in our implementation when its value changes.

Flow Functions:

a. $F_{X:=N}(in) = in - \{X \to *\} \cup \{ X \to N\}$
b. $F_{X:=Y}(in) = in - \{X \to *\} - \{* \to ...X... \} \cup \{ X \to E \mid Y \to E \in in \}$
c. $F_{X:=Y\ op\ Z}(in) = in - \{X \to *\} - \{* \to ...X... \} \cup \{ X \to Y\ op\ Z \mid X \neq Y \wedge X \neq Z \}$
d. $F_{merge}(in_1, in_2) = in_1 \cap in_2$

Note: Even though we have included removal {X->*} in our Flow Functions, we have skipped it from our implementation because SSA guarantees that every assignment to X will be performed in a new register. Hence, there would be no previous value while performing a new assignment.

3.2.4. Implementation of Analysis

a. Data Structures

Our lattice element for Available Expressions is a std: map with (Value*, Instruction*) as the (key, value) pair: - map<Value*, Instruction*>. The worklist algorithm stored one such lattice point for the beginning and end of each instruction and basic block.

b. Data Types

As opposed to Constant Propagation, we will be able to handle a wide range of data types in Available Expressions because the analysis is not data type specific. It just looks for instructions of the form A op B, no matter what the datatypes for A and B are, and no matter what the operand is.

3.2.5. Transformation

The lattice information computed for every instruction is finally passed on to the client that uses the information to perform Common Sub-Expression Elimination. In the client code, we go through the Lattice Point at each instruction, and replace all instances of the key of the lattice point with its value, which is to say that if an expression has already been computed, we replace the instruction usage with its value, that is the instruction where the expression was first computed. Since we cannot remove instructions while replacing their instances, we add all instructions whose instances have been replaced in a set, and finally delete those instructions from the transformed program.

3.2.6. Benchmarks

cse.cpp

Consider the following IR representation after applying mem2reg pass:

```
%add = add nsw i32 10, 20
%add1 = add nsw i32 10, 20
%add2 = add nsw i32 20, 10
%add3 = add nsw i32 %add2, 10
%add4 = add nsw i32 30, 10
%add5 = add nsw i32 %add4, 20
```

The transformation to the above program results in:

```
%add = add nsw i32 10, 20
%add2 = add nsw i32 20, 10
%add3 = add nsw i32 %add2, 10
%add4 = add nsw i32 30, 10
%add5 = add nsw i32 %add4, 20
```

Note that we were able to find that %add and %add1 are equivalent, and hence can replace all instances of add1 with add. However, we are not able to identify that **%add2 is commutatively equal to add** too.

We used LLVM's *isEquivalentTo* function to identify if an instruction is equivalent to another instruction, which unfortunately does not handle commutativity and associativity. Also, an expression of the form x = a+b+c is broken down in the form such that a+b is computed first, and the result is then added to c in another step. So, even if we have an available expression for b+c, we would be unable to perform any optimization in such a case.

3.2.7. Lessons Learnt

We need to handle commutative expressions in a better way. LLVM provides functions to check if a given instruction is commutative/associative. Using these functions to further optimize cases missed in the above benchmark is something we can take a look in future.

3.3. Range Analysis

In Range Analysis, we compute at each program point a range [a,b] for variable X. We then use this analysis to warn a user is there is a possibility of accessing array indexes out of bounds.

3.3.1. Assumptions:

a. As in other analysis, we assume mem2reg pass is run on the input program representation before running this pass.
b. Since we are not doing pointer analysis, we assume that any array present in the program representation has been allocated statically, and not on the heap.
c. If we do not know the lower bound of a variable, we set it to negative infinity. Similarly, if we do not know the upper bound, we set it to infinity. However, it is important for at least one bound to be known at some point of time for a variable to include it in our analysis.

3.3.2.Lattice Definition

Unlike Constant Propagation and Available Expressions, our Range Analysis is a may-be analysis.

Lattice:

$(D, \bot, \top, \sqcup, \prod, \sqsubseteq) = (2^{\{u \to v \mid u \in Vars \wedge v \in A\}}, \{u \to I \mid u \in Vars\}, \phi, \cup, \cap, \subseteq), I = (-\infty, +\infty)$

3.3.3. Flow Function Definition

a. $F_{X:=N}(in) = in - \{X \to *\} \cup \{X \to N\}$
b. $F_{X:=Y}(in) = in - \{X \to *\} \cup \{X \to R \mid Y \to R \in in\}$
c. $F_{x:=y\ op\ z}(in) = in - \{x \to *\}$
$\cup \{x \to (a, b) \mid y \to (ay, by) \in in$
$\wedge\ z \to (az, bz) \in in$
$\wedge\ a = min(ay\ op\ az, ay\ op\ bz, by\ op\ az, by\ op\ bz)$
$\wedge\ b = max(ay\ op\ az, ay\ op\ bz, by\ op\ az, by\ op\ bz)\}$

Note: We did not actually require to handle all of the above cases in case of binary operations because the data structure that we used to represent our range (ConstantRange) had functions to provide the updated range, given the range of operands and the operator.

d. $F_{X:=\Phi(Y,Z)}(in) = in - \{X \to *\} \cup \{X \to (l,u) \mid Y \to (l_1,u_1) \in in \wedge Z \to (l_2,u_2) \in in \wedge l = \min(l1, l_2) \wedge u = \max(u_1, u_2)$
e. $F_{merge}(in_1, in_2) = in_1 \cup in_2$

For branch functions, we have limited our scope to comparison of a variable with a constant. Thus, we do not handle the case where a variable is compared with another variable, both having their own range.
We currently handle only +,-,* and / Binary Operations. Adding more binary operations is a mere task of adding more switch cases in the implementation which is straight forward.

3.3.4. Implementation of Analysis

a. Data Structures

Our lattice element for Available Expressions is a std: map with (Value*, ConstantRange*) as the (key, value) pair: - map<Value*, ConstantRange*>. The worklist algorithm stored one such lattice point for the beginning and end of each instruction and basic block.

In ConstantRange, there are special representations to imply full set and empty set. These values can be enquired using *isFullSet* and *isEmptySet* function calls.

b. Data Types

For the current implementation, we have only considered range of integers. We limited the scope of the analysis to simplify the Flow Functions for Range Analysis.

3.3.5. Potential Bug Finding

Once the analysis has found potential range for different variables in a program, the client uses this range to determine if there is a possibility of accessing an index out of bounds at any point in the program. We iterate over all GetElementPtr instructions. This instruction is called to retrieve the pointer to a particular array position. We obtain the length of the array and convert the pointer operand into a PointerType, before casting it into an ArrayType and invoking the getNumElement method. We compare the range [0, size] to the range of the index variable. If the accessed index's range is not a subset of the accessible range, we throw a possible index out of bound warning.

3.3.6. Benchmarks

Consider the following C++ program:

rangeForTest.cpp

```
i.   int arr[100];
ii.  int i;
iii. int x = 0;
iv.  for (i=0;i<100;i++)
v.   {
vi.  arr[i] = 50;
vii. arr[x] = 100;
viii. x = x + i;
ix.  }
x.   arr[i] = 10;
```

In the above program, we are able to determine that inside the for loop, the value of i is guaranteed to be between 0 and 100, while outside the loop it is guaranteed to be in the range [100, ∞). So we display a warning to the user at line (x) but not at line (vi).

Ensuring Termination: Line (viii) is particularly interesting. Notice that the value of x keeps changing every time for.body is analyzed in the worklist, causing its successors to be added to the worklist. This can lead to infinite execution. We prevented this condition by setting an upper bound to the number of times the value of a variable can change if it is monotonically increasing. So, after running a certain number of times (set to 1000 in the code), the value of x is set to a full set. Therefore we display a warning to the user at line vii as well, because our analysis computes the value of x to be a full set at that point.

3.3.7. Lessons Learnt

a. The above benchmark made us realize that running into a non-terminating program is much easier in range analysis, and a deeper insight into the problem showed that optimizing termination in such cases is still an area of active research. Although we have succeeded in ensuring termination, there can potentially be better ways to reach termination in such cases.
b. We were trying to implement special cases to prevent a potential divide by zero. Later we realized that ConstantRange handles such cases by itself. This turned out to be positive for us and simplified the implementation.

4. Conclusion and Future Work

In the course of the project, we implemented dataflow analysis, and clients for three types of optimizations: Constant Propagation, Available Expressions and Range Analysis. We realized that Pointer Analysis could have improved the results of all the above analysis. We lost precision at a few places, as described in the benchmarks in the report. Nevertheless, our analysis was able to produce optimized results for several examples. Moreover, a thorough knowledge of SSA at the beginning of the project could have helped designing the merge operations differently, to further optimize results that lost precision at PHI nodes. We have provided a limited set of benchmarks to show possible areas where our analysis would break, along with the reason for it to break, and the potential fix that can be applied to optimize those benchmarks. Some features, such as handling branch statements in Constant Propagation and Available Expressions, where the register value folded/ expression computed is inside the if statement, could not be completed due to time constraints. The results, though correct, would again be imprecise for such scenarios. These features can be added to the analysis in future.