CSE 231 Project 2 Report

Brian Anderson, Nathan Heisey, Pavan Kumar
UC San Diego
{b3anders, nheisey, pavan}@eng.ucsd.edu

June 12, 2014

1 Overview

The goal of our design was to demonstrate the principles of dataflow analysis that could be used to implement a useful set of optimizations. We designed the framework so that various dataflow analyses and optimizations could implement a simple set of functions that allow for as much control as possible. Each analysis implements merge and flow functions, as well as flow information to be stored at the output of each node. The work of creating the control flow graph and running the worklist algorithm is delegated to the framework, which calls upon specific analyses to determine which instructions are relevant and should store dataflow information. We did not add explicit support for analyses that store global information, but such global analyses could be implemented using our framework at the expense of more memory than might normally be required. Our design was not meant to allow for reverse-flow analyses such as variable liveness or interprocedural analyses such as the lock/unlock analysis described in lecture; such support could be added, but would require changes to the underlying framework that would introduce additional complexity and test cases.

While our goal was to demonstrate the principles of dataflow analysis, it is entirely possible to implement useful optimizations using our framework. Different analyses can be chained together so that succeeding analyses may use the results of preceding ones. For example, we used the results of the may-point-to analysis to improve constant propagation analysis. However, such chains operate on a per-function basis; each function is analyzed individually and all analyses are run on each function in succession, meaning a global analysis would be incomplete at any given time if other analyses attempt to use it. This is an artifact of llvm optimization ordering, however, and could be fixed by changing the basic framework from using FunctionPass to using ModulePass.

2 Interface design

In order to run, our analyses require that -instnamer is run on the input files first. This pass provides ‘real’ names to variables and temporaries, which otherwise would be nameless and collide with each other when storing information about them. The option -mem2reg is useful but not actually required; our pointer analysis assigns names to allocated memory locations when safe, allowing us to track when these locations have constant values, known ranges, or particular expressions. However, we used -mem2reg due to the benefits of it cleaning up the code and reducing the number of instructions. Our analyses assume that provided instructions are already in SSA form, which llvm appears to do automatically.

The framework is designed to make analysis implementation as simple as possible. The framework accepts the type of flow function information to be stored as a template parameter; analyses are expected to inherit from the framework and implement several pure virtual functions that operate on this information.

Two functions, createEntryInfo() and createBottom(), return information objects. The first accepts a list of function parameters, which the analysis may initialize in the new info object. This info object is then passed to the entry node of the function. It is assumed that requested info that does not exist will be initialized to top, although if the analysis requires a different result it will not affect the framework. The second function returns an information object where all entries are bottom, e.g. if the requested information is not available, bottom should be returned.

validInstruction() accepts a pointer to an LLVM instruction and should return true if the instruction is relevant to the particular analysis or optimization. The framework uses this information to build the dataflow graph; if the instruction is not relevant, no node will be created (although a node id will be reserved for it, in order to maintain consistency with the other analyses for comparison purposes).

flowFunction() is used to run the actual flow functions on the data. It accepts a pointer to an LLVM instruction, the unqiue id of the current node in the dataflow graph, and the input information from the previous node. The node id can be used to find the corresponding input flow information from another analysis, as well as the previous output information from the current node. The resulting information is returned, and the framework uses it to determine which nodes need to be added to the worklist. The information is stored in the graph and is accessible at any time if the id of that node is known, even after the function has finished being implemented.

merge() is used to handle merge nodes. It does not provide an instruction, but instead offers a list of inputs to be merged. One of the major problems we ran into is that PHI nodes are a separate instruction, so they are not automatically integrated into the merge. This could result in lost information, since a phi node cannot know which information object its parameters arrive from, as they have already been merged by the time we arrive at the phi node. We discuss this further in the future work section.

In contrast to the functions implemented by an analysis, the information object is expected to implement only one function, hasChanged(). This accepts a pointer to another information object and should return true if the information is different, thus indicating that the input information to a node changed. If it is true then the node and any immediate successors of the node need to be added to the worklist.

The framework handles all memory deallocation via the delete operator. Analyses are expected to use new to allocate information objects and to create a new information object at each flow function, merge node, and call to createBottom() or createEntryInfo(). Otherwise, the analyses know nothing about memory allocation and usage unless they have additional internal requirements.

Note that in general, if an analysis is accessed but has not yet been run, it will have a NULL instance and cause errors. As such, the analysis either must require certain other analyses are run before it, or make a check before using them. For example, our constant propagation analysis assumes that our pointer analysis has been called before it and does not verify that the pointer analysis exists before attempting to use it.

We considered using reference counting to handle memory management for graph nodes and information objects, but we decided against this idea because we couldn't think of an easy way to implement this code in one convenient place. Graph nodes would have been especially problematic because they store references to both their parents and their children. We experimented with replacing inheritance with template parameters and vice versa, but found that more parameters made the underlying framework unnecessarily complex while pure inheritance required annoying casts in order to get the data we needed.

3 Client Analyses for the Framework

For each section, the flow functions listed correspond to the instructions we support.

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. Optimizations using constant propagation can then substitute these constants and remove dead variables.

3.1.1 Lattice and flow functions in mathematical notation

Generally, we never remove X because SSA guarantees that X is a new variable. For safety and debugging purposes, however, we implemented most flow functions in non-SSA form. Since we did not have a must-point-to analysis, we assumed that if the may-point-to set contained a single variable, then it was equivalent to must-point-to. In the flow functions below, variables are lower case and constants/special functions are upper case.

Domain:

$$D = P(\{x \to N \,|x \in Vars\})$$

Flow functions:

Operation: Used for Add, Sub, Mul, SDiv

$$\begin{aligned} F_{x:=y\ op\ z}(in) = in &- \{x \to *\} \\ &\cup \{x \to C \mid y \to C_1 \in in \\ &\wedge\ z \to C_2 \in in \\ &\wedge\ C = C_1\, op\, C_2\} \end{aligned}$$

$$\begin{aligned} F_{x:=y\ op\ C_1}(in) = in &- \{x \to *\} \\ &\cup \{x \to C \mid y \to C_2 \in in \\ &\wedge\ C = C_2\, op\, C_1\} \end{aligned}$$

$$\begin{aligned}F_{x:=C_1\ op\ y}(in) = in - \{x \to *\} \\ \cup \{x \to C \mid y \to C_2 \in in \\ \wedge\ C = C_1\ op\, C_2\}\end{aligned}$$

$$\begin{aligned}F_{x:=C_1\ op\ C_2}(in) = in - \{x \to *\} \\ \cup \{x \to C \mid C = C_1\ op\, C_2\}\end{aligned}$$

Merge:

$$F_{merge}(in_1, in_2) = in_1 \cap in_2$$

PHI:

$$\begin{aligned}F_{x:=PHI[y,z]}(in) = in - \{x \to *\} \\ \cup \{x \to C \mid y \to C \in in \\ \wedge\ z \to C \in in\}\end{aligned}$$

Assign: Equivalent to Bitcast, SExt

$$\begin{aligned}F_{x:=y}(in) = in - \{x \to *\} \\ \cup \{x \to C\ \mid y \to C \in in\}\end{aligned}$$

Assign-dereference: Equivalent to Load

$$\begin{aligned}F_{x:=*y}(in) = in - \{z \to * \mid x \to z \in may_p t_t o(x)\} \\ \cup \{x \to C\ \mid z \to C \in in \\ \wedge\, y \to z \in may\_pt\_to(y)\}\end{aligned}$$

Dereference-assign: Equivalent to Store

$$\begin{aligned}F_{*x:=y}(in) = in - \{z \to * \mid x \to z \in may\_pt\_to(x)\} \\ \cup \{z \to C\ \mid z \in must\_pt\_to(x) \\ \wedge\, y \to C \in in\}\end{aligned}$$

3.1.2 Lattice and flow function implementation

In constant propagation we stored flow information in a map from string to int to represent variables and their values. We stored a map for each node in the framework. The method we used to handle the add, sub, mul, and SDiv is the same except for the operator associated with each instruction. We first we get the values of both operands. Each operand will either be: a constant value, a variable in the input map, or some non-constant. If there is a non-constant operand we gain no information from the instruction. If both operands are either constant or some known constant variable, we compute the result of the expression by using the value or by looking it up in the input map. We then assign the variable associated with the operation the result of said operation within our control flow information. PHI instructions are similar to the above expressions. First we have to make sure both operands of a phi node are constants or refer to some variable in the input map. If both are constant we can compare the two values and can propagate them if they are equal. For both Load and Store instructions we use the results of the pointer analysis. How we did this is described in the section after pointer analysis.

3.1.3 Instances where your analyses loses precision

When we encounter a phi instruction and there is a variable that has not been initialized, our program effectively assumes top. When it reads a phi node it will look to see the values of both operands. If one of the operands is a variable and it is not in the input map, then our implementation adds no new information to the output map. However, if the second operand is a constant that appears later in the code, we could've used that value instead if we had assess to it. For example if we had the program:

```
loop:  phi i = [0, inc]
...
inc = i
phi node should give us {i -> 0}
```

Running constant prop should give inc rightarrow 0, i rightarrow 0 as output. For our analysis we output the empty set since we gain no information about i from the phi node and then subsequently gain no information from the inc assignment. However, we couldn't create this case as LLVM would simply remove the phi node then the values were equal and the inc variable was constant.

3.1.4 Observations

The mem2reg pass leaves us with few cases where we can apply our analysis since most constant values are already propagated. The few places we could apply our analysis were on expressions. Additionally, our constant propagation only supports integer operations. It could be easily extended to floating point operations. However, this would add additional overhead since we would duplicate the implementation for integer operations and add type casting to handle constantInt or constantFP values.

3.2 Available Expressions

For this type of analysis, we identify certain expressions in the program whose value need not be recomputed.

3.2.1 Lattice and flow functions in mathematical notation

Unlike in constant propagation, we assumed SSA as provided by llvm. This allowed us to avoid some of the complexities introduced into the flow functions when removing old expressions and guarding against scenarios where a var x is a function of itself. We do not support checks for associativity and other mathematical equivalence metrics of the operations.

Domain:

$$D = Power\{x \rightarrow E \ \mid x \in Vars \land E \in Expressions\}$$

Operation: Used for Add, Sub, Mul, SDiv

$$F_{x:=y\ op\ z}(in) = in \cup \{x \rightarrow y\ op\ z\}$$

$$F_{x:=y\ op\ C}(in) = in \cup \{x \rightarrow y\ op\ C\}$$

$$F_{x:=C\ op\ y}(in) = in \cup \{x \rightarrow C\ op\ y\}$$

$$F_{x:=C_1\ op\ C_2}(in) = in \cup \{x \rightarrow C_1\ op\ C_2\}$$

Merge:

$$F_{merge}(in_1, in_2) = in_1 \cap in_2$$

3.2.2 Lattice and flow function implementation

In available expressions we stored flow information in a map from string to strong to represent variables and their expressions. The expressions were compressed to a string with the format "operand operator operand". We stored a map for each node in the framework. The implementation for avaiable expressions is near identical to the constant propagation implementation for add/sub/mul/sdiv. However, instead of propagating the result of the expression we propagate the expression itself. We chose not to handle PHI nodes since the only time PHI nodes were created in testing was when the two values were different.

3.2.3 Instances where your analyses loses precision

While we could add the addition of Load and Store instructions, we chose not to since they copy information rather than add new expresssions. For example, if an expression was loaded into the variable x and x was then part of a supported operation, we would not be able to determine which expression x is a copy of. This would prevent us from reducing the list of avaiable expressions when optimizing. For example, if x was a copy of y and y = 1+1, if there was an expression z = x+1, we could not remove the load instruction and propagate z = y+1 when optimizing. The exclusion of PHI nodes could result in some lose of information if the two operands contained expressions that resulted in identical values. For instance if x = phi[y,z] and y = 1+1 and z = 1+1, then we could have propagated some information for x as either of the two expression. Although technically different expressions themselves, they are identical. However, we decided against adding functionality.

3.3 Observations

One of the tough design decisions we had made for available expressions was whether to examine the information map or not. While running our benchmarks, we noticed that some expressions evaluated to the same result even though they used different variables. For example, if we had the information: add = 1+1, add1 = 1+1, add3 = 1+1, add4 = add1+add2, add5 = add1+add3, add5 could have refered to add4 instead. It is possible to examine the map for identical instructions on each assignment, but we decided to keep the additional information. We figured that, if there is some identical expressions, the optimizer could sift through any extra information. However, since phi nodes can make us lose information, such optimizations may be impossible.

3.4 Range Analysis

In Range Analysis, we try to identify different values a variable can take using which we can determine the range of a given variable.

3.4.1 Lattice and flow functions in mathematical notation

Range analysis was more complex because of the effect of performing operations on ranges that extended to infinity.

Domain:

$$D = P(\{x \to (a,b) \mid x \in Vars \wedge a, b \in \{Z, +\infty, -\infty\}\})$$

Flow functions:

Operation: Used for Add, Sub, and Mul

$$\begin{aligned} F_{x:=y\ op\ z}(in) = in - \{x \to *\} \\ \cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\ \wedge\ z \to (a_z, b_z) \in in \\ \wedge\ a = min(a_y\ op\ a_z, a_y\ op\ b_z, b_y\ op\ a_z, b_y\ op\ b_z) \\ \wedge\ b = max(a_y\ op\ a_z, a_y\ op\ b_z, b_y\ op\ a_z, b_y\ op\ b_z)\} \end{aligned}$$

$$\begin{aligned} F_{x:=y\ op\ C}(in) = in - \{x \to *\} \\ \cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\ \wedge\ a = min(a_y\ op\ C, b_y\ op\ C) \\ \wedge\ b = max(a_y\ op\ C, b_y\ op\ C)\} \end{aligned}$$

$$\begin{aligned} F_{x:=C\ op\ y}(in) = in - \{x \to *\} \\ \cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\ \wedge\ a = min(C\ op\ a_y, C\ op\ b_y) \\ \wedge\ b = max(C\ op\ a_y, C\ op\ b_y)\} \end{aligned}$$

$$\begin{aligned} F_{x:=C_1\ op\ C_1}(in) = in - \{x \to *\} \\ \cup \{x \to (a,b) \mid a = b = C_1\ op\ C_2\} \end{aligned}$$

Merge:

$$\begin{aligned} F_{merge}(in_1, in_2) = \{x \to (a,b) \mid x \to (a_1, b_1) \in \in_1 \\ \wedge\, x \to (a_2, b_2) \in in_2 \\ \wedge\, a = min(a_1, a_2) \\ \wedge\, b = max(b_1, b_2)\} \end{aligned}$$

PHI:

$$\begin{aligned}F_{x:=PHI[y,z]}(in) = in \cup \{x \rightarrow (a,b) \mid\, & y \rightarrow (a_y, b_y) \in in \\ & \wedge\ z \rightarrow (a_z, b_z) \in in \\ & \wedge\, a = min(a_y, a_z) \\ & \wedge\, b = max(b_y, b_z)\}\end{aligned}$$

Assign: Equivalent to Bitcast, SExt

$$F_{x:=y}(in) = in \cup x \rightarrow (a,b) \mid y \rightarrow (a,b) \in in$$

We also support alloca and getelementptr, which we use to implement array bounds checking. alloca records the size of allocated arrays, and getelementptr checks that the created address is within the bounds of the original array. If the array is dynamically allocated with new or malloc, we do nothing, even if the array size is constant.

3.4.2 Lattice and flow function implementation

For Range Analysis, we store the range information in each variable in a struct which includes minimum range, maximum range, Top/ Bottom initialization methods, and positive and negative infinity bias variables to assist in achieving fixed point in loops. In the flow function implementation, we first look up the opcode for the desired instruction. The instructions supported include: Alloca, Add, Sub, Mul, SDiv, PHI Bitcast, SExt, getelementptr.

We get the logic to handle add, sub and mul directly from the mathematical definition of flow functions described above. In these functions, we determine the maximum and minimum range of the variables at the end of the operation. While handling the mul instruction, if we see that any of the operands are infinity, we set the variable to top since depending on the other operand, the range can be either negative or positive infinity. Division is an extension of the above operations, which we do not handle in our implementation.

PHI instructions are essentially a merge operation where we choose the widest range possible depending on the input variables to the PHI node.

Since we also perform array size detection along with range analysis, we use alloca to get the array size.

The getElementPtr instruction operands includes array index and pointer index. From this, we calculate the array index range and the pointer index range. The array index range information is used to see provide 'array index out of range warnings'. We currently do not use the pointer information to provide warnings.

3.4.3 Instances where your analyses loses precision

We had not implemented the case for SDiv for range analysis. Although we could extend our implementation to include it, we felt that it was similiar to the others save for the operator when it comes to extending the range of a variable. Additionally, in loops we have two bias variables, one for positive and one for negative increases. When we increase the range of a variable we increment the associated bias variable. When a bias is equal to 3 we set either the max value to positive infinite or the min value to negative infinity, depending on if it was the pos or neg bias variable. So if we have the loop for(a = 0; a ¡ 10; a++) a will only exist in the range between 0 and 9 at runtime. However, each cycle of the analysis over the loop makes a increase in range, therefore we increment the positive bias variable. Once the positive bias is equal to 3 we set the range for a between 0 and infinity, losing small-range information. We could've looked up the compare instruction for the loop to get more precise range information to reduce this loss of precision. We chose not to implement this due to the complexity required to determine the number of loop iterations.

3.5 Intra-Procedural Pointer Analysis

3.5.1 Lattice and flow functions in mathematical notation

We chose to implement the midterm's pointer analysis, with some support for dynamic variable allocation. While this prevented us from operating on any large programs, it allowed us to perform more interesting optimizations for constant propagation.

Domain:

$$D = P(\{x \to y \mid x, y \in Vars\})$$

Flow functions:

Assign address: Equivalent to Alloca

$$\begin{aligned} F_{x:=\&y}(in) = in - \{x \to *\} \\ \cup \{x \to y\} \end{aligned}$$

Assign: Equivalent to Bitcast

$$\begin{aligned} F_{x:=y}(in) = in - \{x \to *\} \\ \cup \{x \to z \mid y \to z \in in\} \end{aligned}$$

Assign-dereference: Equivalent to Load

$$\begin{aligned} F_{x:=*y}(in) = in - \{x \to *\} \\ \cup \{x \to w \mid z \to w \in in \\ \wedge y \to z \in in\} \end{aligned}$$

Dereference-assign: Equivalent to Store

$$F_{*x:=y}(in) = in \cup \{z \to y \mid x \to z \in in\}$$

Merge:

$$F_{merge}(in_1, in_2) = in_1 \cup in_2$$

PHI:

$$\begin{aligned} F_{x:=PHI[y,z]}(in) = in \cup \{x \to w \mid y \to w \in in \\ \vee\ z \to w \in in\} \end{aligned}$$

3.5.2 Lattice and flow function implementation

For Pointer analysis we had a map from string to string to represent variables pointing to other variables. We used a special form of a map called a multimap so that one variable could be mapped to multiple variables. We stored a map for each node in the framework.

The Alloca and Call instruction to new (in-code as _Zn** where ** are the return type and parameter type characters) both add new pointers. So the instructions X = Alloca or X = call let us add new pointer information $\{X \to val@X\}$. We assume that neither alloca nor calls to new are unsafe.

We had also determined the instruction ”store x, p” is equivalent to *p = x. The information we add to the map is that everything p now points to will also point to all of the things x may point to.

We also determined the instruction ”x = load p” is equivalent to x = *p. In our information map we have x point to all the things that p may point to.

For a PHI node ”x = phi[y,z]” we have it so that x points to the union of the sets of what y may point to and what z may point to.

3.5.3 Instances where your analyses loses precision

There is a possibilitiy of dangerous information loss on new. It was theorized that calling new could erase information if there was a pointer to a new constant. However, we were unable to demonstrate this case. We assumed that since SSA makes each assignment have a new variable that this case does not happen.

3.6 Applying Pointer Analysis to Improve Constant Propagation

3.6.1 Implementation

We configured our constant propagation to take the output information from pointer analysis. For a load or store instruction in constant propagation we would traverse through all points to information in the equivalent node output from the pointer analysis. If everything that the loaded variable may point to results into a set of variables and every variable in that set has the same constant value, we can assign that value to the variable associated with the load/store instruction. For clarification, ”x = load y” is equivalent to ”store y x”; if everything y points to has the same resulting value, then we can propagate x with that value. Eg if y pointed to w and z, whereas both w and z resulted with the value of 5, then we can propagate x as 5.

3.6.2 Observations

Adding may-point-to lookup in the constant propagation analysis resulted in constant propagation being unable to run in isolation. In our pointer test, the addition of may-point-to information allowed us to find two additional constants to propagate when compared to the constant propagation anaylsis that did not accept may-point-to information. Branch folding is designed to use whichever constant propagation method is available so that any improvements to constant propagation also improves branch folding.

One of the problems we had with pointer analysis was the lack of real instruction names. We had tested several small benchmarks, and in each one we had a no-name variable pointing to a massive body of other variables. When we examined the llvm code everything had a unique name. This lead us to try changing some of our implementation to include additional string information in the variable name, but this proved to be complicated and prone to errors. Eventually we discovered that llvm does not physically store the name of the temporary registers, which lead us to the discovery of the -instnamer command to assign unique names to temporary variables.

Another difficulty with pointer analysis was implementing the flow functions for load and store. In the flow functions presented in class, all variables are addressable. In llvm-IR, registers are non-addressable named variables, while memory locations are unnamed but addressable. This dichotomy between dataflow analysis and llvm-IR concepts of variables caused some problems, since a variable named i0 and a pointer named p0 would both be represented in llvm as registers storing addresses, making i0 a pointer and p0 a double pointer. Originally, we simply stored the information $\{p1 \rightarrow i0\}$, which was true for the original C code but was incorrect for the llvm bitcode. To fix this, we added support for alloca instructions to pointer analysis by naming the allocated memory locations, where an alloca of i0 would result in $\{i0 \rightarrow val@i0\}$. This means our original example (involving two alloca's and one store) becomes $\{i0 \rightarrow val@i0, p1 \rightarrow val@p1, val@p1 \rightarrow val@i0\}$. While it is not as easy to connect to the original C code, it is internally consistent with the llvm bitcode and allows for more accurate pointer analysis.

Naming memory locations led to its own set of interesting side effects. For one, since our names are unique to the variable that stores them, revisiting the same alloca does not create new variables, or even notice that one alloca may have been called multiple times. In effect, this means that some of our pointers actually refer to multiple memory locations. This can cause problems for constant propagation, particularly if you have a loop building up some sort of linked list. Accessing the first or last element in the list is probably fine, but if you attempt to chain calls to access some element n items from the first or last element, our pointer analysis won't know about the intermediate values and may cause failures.

Arrays and structures have their own set of problems; we decided to ignore these for pointer analysis. The main issue comes from the getelementptr instruction, which calculates an address. Our pointer analysis does not recognize this instruction, so it cannot know that the result of getelementptr calculates an offset to a provided pointer. As a result, our pointer analysis generally cannot conclude anything about an array or structure dereference. While we probably could extract address information from getelementptr for greater accuracy, we decided that the subtleties of getelementptr would be far too complex for the time we had available.

4 Benchmarks

We created five basic tests, each meant to test particular functions of C/C++. None of the benchmarks are actually useful programs, but they represent important aspects of C and C++ where our analyses should be able to detect information. We also found a command line chess program on the internet and tried running our analyses on this, so that we could have a more complex test program; while not especially successful, it did provide some insight into the limitations of our analyses. We did not use the benchmarks from part 1 of the project due to the number of arrays they used (a known issue with our implemention) or the difficulty of verifying the results.

4.1 arrayTest.cpp

In this test file, we create 2 arrays of different sizes and a pointer array. We try to simulate an array index out of bounds scenario where our range analysis must provide warnings. The common instructions present in the .reg.bc.ll file include: load, store, alloca, getelementptr, sext, add, sub, mul, br, phi etc.

When we executed our Range Analysis on this program, we noticed that we were able to generate out of bounds warnings and valid accesses for all the expected cases. However, the warnings in a few of the cases were due to a loss of precision when we accessed arrays inside of loops. We expected these out of bounds warnings because of the bias we implemented, which can make a variable have an infinite range even though the number of iterations is finite.

4.2 assignTest.cpp

This is a basic test program which is used to test available expressions analysis and has multiple integer assignments, expressions and a function call. The instructions encountered in this file include add, sub, mul, div, call and ret. As noted earlier, in this test case we saw that we are left with very few cases where we need to add the available expressions on our map. Running it had added all of the expressions in the program to the output information map.

4.3 branchFoldTest.cpp

This is another basic program where we use the results of constant propagation to preform simple branch folding. We first create several variables by directly assigning a constant value or computing with a short expression. Running constant prop gave us a reduced list of variables after mem2reg. After the assignments was a short list of if statements who conditions were comparisons or constant booleans.

Running branch folding showed that we could use the constant information to reduce all of the branches that relied on integer comparisons. The branches with boolean constants were removed by llvm mem2reg optimizations.

4.4 ptrTest.cpp

This was our primary pointer test, containing multiple levels of pointers and nested if/else statements. We also took advantage of the opportunity to test constant propagation and branch folding to see if constants buried under multiple layers of pointers could be detected and used. The pointers are set to point to different values in the if/else, then afterward are used to calculate a simple expression that amounts to adding two constant values. We also tried changing the constant values and setting them inside of the branches to verify that constant propagation could detect the appropriate cases.

Running pointer analysis resulted in a massive may-point-to list, since the nested if/else resulted in multiple possibilities for each variable. While large, the results were accurate and sufficiently optimistic for constant propagation to detect that the values referenced by the pointers were constant. As with branchFoldTest, branch folding successfully detected foldable and unfoldable branches. Range analysis was not able to do much, since we did not include support for using pointer analysis.

4.5 structTest.cpp

This test was intended to break everything. In structTest, we create a linked list in a loop, then print specific values from the linked list. Pointer analysis got lost at the calls to new within the loop, especially when attempting to access elements of the internal structure. Constant propagation was unable to conclude anything, although it somehow resulted in several temporaries getting set to bottom. The reason for this is unclear, although we suspect a bug in our pointer analysis may-point-to set and constant propagation's initialize to top or bottom information.

4.6 chessTest.cpp

We found this program in an online tutorial. The program generates a chessboard and lets a player move pieces around the board; it does not keep score or verify that movements are legal, but it is still a fairly complex program. When we ran our analyses on it, we found that there was very little that could be done. The chess program used a lot of subroutines and arrays which are not handled very well by our pointer analysis.

5 Future Work

There are several important ways we would improve the framework and analyses using the information we have now. For instance, we implemented most of the analyses so that the entire node was top, bottom, or somewhere in between. In general, individual variables in the map should be listed as top or bottom, and the overall information object should know whether to initialize variables not found in the map to top or bottom. We corrected this for constant propagation and found that this makes our analysis slightly more optimistic, although we still lose information due to PHI nodes coming after the merge nodes as discussed in the constant propagation section. Range analysis already has this feature built in, but pointer analysis and available expressions may benefit from adding this feature.

Adding support for different types would greatly improve constant propagation and the optimizations that use it. For example, adding support for booleans with the cmp instructions would allow branch folding to simply check if the branch loop condition is a constant, rather than having to perform some tests on the cmp instruction itself, which might not be easy to link to a specific branch. Adding floating point support has obvious benefits, although adding casting support would create additional complexities.

Probably the most important missing feature in our pointer analysis is array and structure support. Whether arrays and structures are supported by allowing one may-point-to value to represent multiple memory locations, or by giving each element of an array or structure a unique name, such support would go a long way towards improving our analyses. Arrays and structures featured heavily in most of the benchmarks we tested, so this proved to be a major limitation.

Both range analysis and available expressions could benefit from using pointer analysis; pointers may point to a value with a known range (or multiple values with known ranges), and avaliable expressions could include variable addressing and pointer dereferences in its repertoire of supported operations. This is especially important for common subexpression elimination, since memory operations can be expensive and knowing a variable has been dereferenced before could result in many important optimizations.

Available expressions could also benefit from storing expressions as structures instead of strings. This would allow for deep comparisons of common subexpressions and use of associativity and commutativity properties of some operations.

6 Conclusion

We perform various dataflow analyses, two place holder optimizations, and one analysiis improvement by incorporating pointer analysis. The two place holder optimizations are: range analysis out of bounds warnings, and branch folding from constant prop. Our constant propagation analysis is improved with pointer information and we enabled the framework to add pointer information to our other analyses. We lose precision in several known cases, some tested and some theoretical (due to not being able to develop a test case to prove it). However, most of these could be fixed with additional features to our analyses. We did not implement these features due to time and complexity constraints. Our benchmarks went through several different versions and we designed them to test cases where we believed our analysis would break. Some of the simpler benchmarks existed to test functionality only.