# CSE 231 Project Report

Danny Y. Huang, John Mangan, Matthias Springer

Department of Computer Science and Engineering
University of California, San Diego

## ABSTRACT

In this work, we present an LLVM-based code analysis and optimization framework. The framework builds and operates on a control flow graph, based on the input program's intermediate LLVM representation. It runs an optimistic worklist algorithm until a fixed point is reached by applying flow functions over and over again.

We also present a *meta-lattice* pattern which abstracts complexity out of our four analyses implemented: constant analysis, range analysis, intra-procedural pointer analysis, and available expressions analysis. Furthermore, we show how to use the results from the constant analysis for constant folding and constant propagation, and how to use the results from range analysis to throw errors if an array index is definitively out-of-bounds.

## 1. INTRODUCTION

Lattices and flow functions are abstract and often obscure concepts in graduate computer science education. This project attempts to bridge such theoretical knowledge with hands-on experience, in which we implemented a number of known compiler techniques from scratch, evaluated their precision, and gained an in-depth understanding of LLVM internals. As part of this intellectual exercise, we also constructed a general control flow analysis framework (§2) designed for optimistic top-down analyses and optimations over programs represented in LLVM IR.

In this paper, we describe four such analyses that leverage this framework. In particular, we implement code optimization for the first two analyses (§4 and §5). Throughout the paper, we discuss the benefits and limitations that SSA/`mem2reg` introduces, as well as how these affect some of our design decisions. Moreover, we explain the unique challenges about the lattice design, and how our "meta-lattice" is able to address some of the issues (§3).

## 2. ANALYSIS FRAMEWORK

In this section, we define the API of our analysis framework and how it can be used to write new analyses.

### 2.1 Framework Architecture

Our framework consists of three important classes, shown in Figure 1. Two of them are abstract and must be implemented when writing a new analysis.

`Lattice<T>` This abstract class declares the interface for lattice operations such as join and meet. It also stores flow functions. The template parameter `T` defines the datatype for lattice elements.

`AnalysisPass` This abstract class contains the methods for building and printing the control flow graph, as well as the worklist algorithm. Program optimization code should be implemented in this class.

`BlockNode<T>` This class stores an LLVM basic block and the lattice elements after applying the flow functions on each instruction. It also contains pointers to the successor nodes.

### 2.2 Framework Setup and Execution

Figure 2 gives a high-level overview of our framework and shows how we can define and register flow functions and optimization procedures.

Flow functions are defined in the `Lattice` class as static functions that transform a lattice element/state into another lattice element by analyzing an instruction. They are registered in the constructor and added to a dictionary that maps instruction opcodes to flow function pointers. If the algorithm encounters an operation without a flow function during the analysis, the default flow function is used.

Optimization procedures are defined in the pass class as static functions that get passed an instruction and the lattice element after evaluating the flow function on this instruction. The registration of optimization procedures is similar to the registration of flow functions. Optimization procedures are executed only after the worklist algorithm is done and a fixpoint was reached[1]. They operate on a per-instruction basis, i.e. optimization procedures are called for single instructions as opposed to single basic blocks or functions. We can easily extend our framework to support per-basic block or per-function optimization procedures, however, we do not need them for the optimizations presented in this work. Optimization procedures are executed in the same sequence as they were registered. This is important in case some optimzations depend on other optimizations being executed first.

## 3. THE META-LATTICE

In this section, we introduce an abstract concept that will be applied in every analysis implementation explored in the remainder of this paper. This single cross-cutting concept is the hardest notion to grasp when convincing oneself that

[1]Intermediate results might be incorrect; therefore, we can only do optimizations once a fixpoint has been reached.

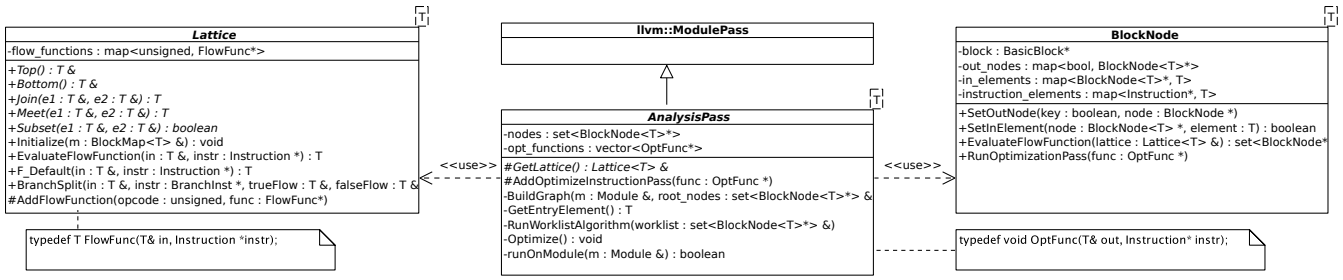



Figure 1: Framework class diagram.

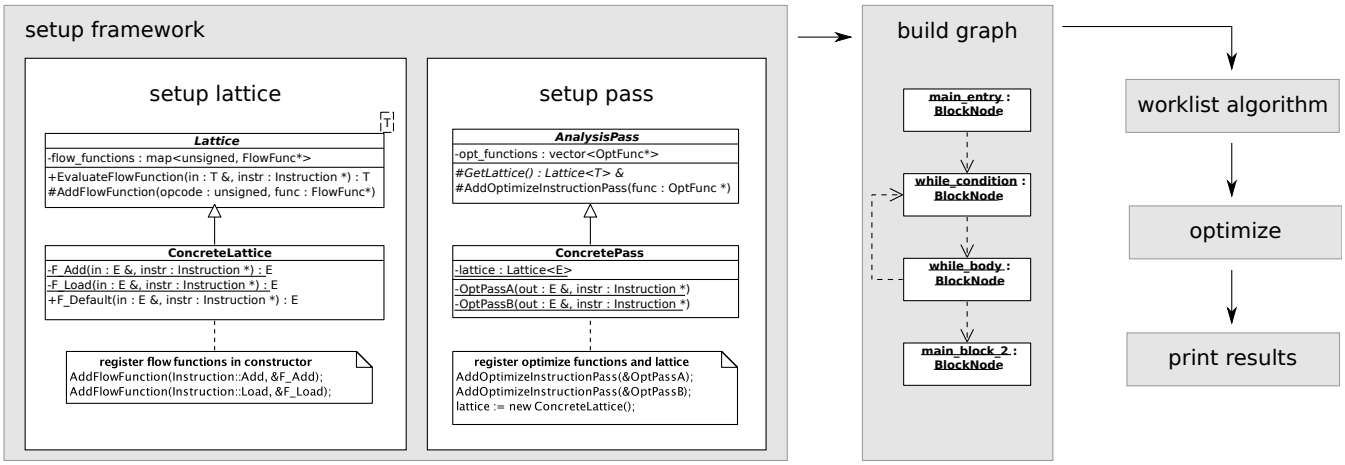



Figure 2: Setting up the framework and running the analysis and the optimization.

each analysis will terminate. It is the goal of this section to encapsulate this concern such that understanding and defending its correctness may be done once and mitigated everywhere else; not unlike aspects which modularize cross-cutting concerns in a software's architecture.

### *Goal.*

To show that our analyses will terminate, we present lattices for which numerous iterations of a work-list algorithm will always converge to a fixed point. Our worklist algorithm iterates over basic blocks, therefore we must first conceptually raise our flow functions from the granularity of instructions to basic blocks. We must then show progress towards termination with each iteration over a basic block. This is guaranteed if flow functions are monotonic and the lattice has a fixed height, as a fixed point will then eventually be reached. The complex notion this section aims to address is how our abnormal lattices may formally (mathematically) be shown to have a fixed height, while we leave monotonicity of flow functions as a concern for the reader during later sections.

### *Flow Function Granularity.*

Our flow functions act at the granularity of a single instruction. Since a basic block consists of an ordered set of instructions for which control flow cannot diverge, we can imagine a big flow function F which "top-folds" (similar to a left-fold) the instruction level flow functions together; thus, F operates at the granularity of basic blocks.

As we will be running optimistic analyses, our flow functions (if properly monotonic) must output a lattice element that is "higher" in the lattice than the input lattice element provided as input, or the equivalent lattice element if it has reached a fixed point.

### *Must-be vs. May-be.*

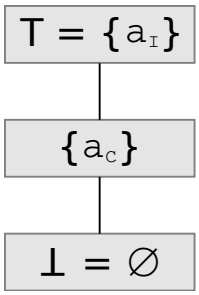
Before continuing, it is worth noting that each analysis in this paper is a *must-be* analysis, as opposed to a *may-be*. That is to claim that for any program point the output produced *must-be* singularly representative, as opposed to a super-set of all possible values that *may-be* representative.

It is worth clarifying that despite being a *must-be* analysis, it can still be an infinite lattice. In fact, each analysis in this paper will make use of an infinite width lattice. Any variable output from an analysis *must-be* a single value, however there are infinite potential values. By the end of this section, the reader should be able to convince themself that it has no affect on the correctness nor scale of our constructed lattices. Any further thoughts of infinite lattices are therefore omitted.

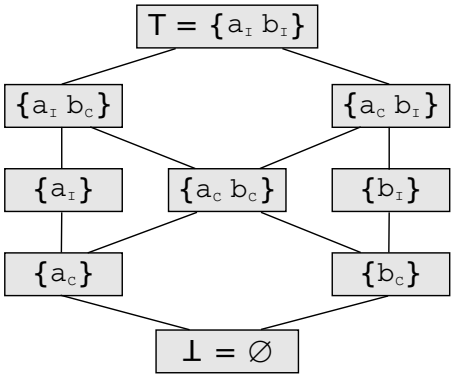
### *Resulting Lattice Elements.*

For every analysis in this paper, each lattice element is a mapping of variables to their analysis-specific state (ASS). An ASS either stores the respective data that *must-be* representative of the variable, or it represents an *incomparable* state. It is also stateful for a variable to not map to an ASS.

Incomparable ASSs conservatively represent unusable data for any optimization that would make use of the prod-




(a) Single variable meta-lattice with fixed height of 2.




(b) Double variable meta-lattice with fixed height of 4.

Figure 3: Two examples of meta-lattices with differing variable counts. $v_C$ & $v_I$ denote variable $v$ mapped to a comparable ASS or incomparable ASS, respectively.

uct of the analysis. Incomparable ASSs are generally constructed as the aggregation of a variable that *must-be* at least two nonequivalent ASSs, where a contradiction would be raised should the analysis imply it is one value but not the other[2].

A lattice element that does not have an ASS mapped for a specific variable is implying that there is no knowledge (yet) for said variable. It is liberal, in that it is willing to accept any knowledge provided by a flow function for said variable, and will optimistically accept it[3].

### *Fixed Height Lattices.*

To determine the fixed height of any lattice, we must first determine the $\bot$ (bottom) and $\top$ (top) lattice elements. For all of this paper's analyses, $\bot$ will always be an empty map, such that no variable has an associated ASS. Symmetrically, $\top$ will consist of a mapping for every variable in the program to an incomparable ASS (implying no optimizations can be made from this analysis, except an optimization to avoid executing any optimization passes). Since any program has a finite quantity of variables ($Vars$), $\top$ will be of finite size.

At this point, we claim that every analysis in this paper has a fixed height $fh = 2 \cdot \| Vars \|$ and provide a construction argument, leaving a formal proof up to aspiring readers.

### *Lattice Construction.*

Let us define a variable promotion as the variable's mapping changing from non-existent ASS to a comparable ASS *or* from a comparable ASS to an incomparable ASS. We consider a non-existent ASS changing to an incomparable ASS as two independent promotions.

A lattice element's parent has exactly one promoted variable differentiating it from the child. That is:

$$p_e = parent(e) \iff \exists v \in e : promoted(v) \in p_e \land\ e - \{v\} = p_e - \{promoted(v)\}$$

Following from this definition, each lattice element will have $n - i$ parents where $n = \| Vars \|$ and $i =$ "count of incomparable ASSs mapped to variables". By iterating over all variables that a lattice element does not map to an incomparable ASS, a parent lattice element can be constructed by promoting the variable (independently per parent). Note that some lattice elements will share some parents.

Finally, for any lattice element $e$ there exist quantities $i$ and $c$ of incomparable and comparable ASSs, respectively, mapped to variables. The height of any lattice element $e$ is $h(e) = 2i + c$. Note that $h(\bot) = 0$ and $h(\top) = 2\| Vars \|$. □

For the remainder of this paper, we omit any discussion about fixed lattice heights as the above has shown them to be so. As long as flow functions are monotonic, iterating over a basic block in our work-list algorithm will progress towards a fixed point (trivially $\top$ if a more precise point does not exist).

## 4. CONSTANT PROPAGATION

In this section, we present our must-be constant analysis implementation and our constant propagation optimization implementation.

### 4.1 Assumptions

The analysis runs on the output of `mem2reg`. Also, the fact that registers in LLVM are only assigned once (SSA) simplifies some flow functions.

### 4.2 Lattice Definition

With $\mathbb{A} = \mathbb{Z} \cup \{\aleph\}$, we define the lattice for constant analysis as follows.

$$(D \quad ,\top \quad ,\bot, \sqcup, \sqcap, \sqsubseteq) \quad = \quad (2^{\{u \to v \,|\, u \in Vars \land v \in \mathbb{A}\}} \quad ,\{u \to \aleph \,|\, u \in Vars\} \quad ,\emptyset, \cup, \cap, \subseteq)$$

[2]For instance, consider the case where, at a merge, one branch assumes that variable $x$ must be the constant value 5 and the other branch assumes that $x$ must be the constant value 4.

[3]We will come back to this when handling PHI nodes.

Our domain is a mapping from variables to either a constant integer value or $\aleph$ (not a constant). Our implementation can easily be extended to support more data types. $\top$ is the most conservative lattice element, which is always safe to assume: every variable is not a constant. $\bot$ is the most optimistic lattice element: every variable can be any constant.

### *Example.*

$\{x \to 4, y \to \aleph\}$ implies that the value of $x$ is the constant 4 and $y$ is not constant. If there are more variables in the program code, e.g. $z$, then we do not have any knowledge about them at this time.

### *Flow Functions.*

The flow functions for assignment, addition and PHI nodes are defined as follows. Flow functions for other arithmetic operations are similar to addition.

$$F_{X:=Y}(in) = in \quad - \{X \to *\} \cup \{X \to c \mid Y \to c \in in\}$$

Note, that we do not have a flow function for an assignment $X := Y$ in our implementation, since LLVM would use $Y$ at any point where $X$ is needed.

$$F_{X:=Y+Z}(in) = in \quad - \{X \to *\} \cup \{X \to c \mid Y \to a \in in \wedge Z \to b \in in \wedge c = a + b\}$$

Note, that $\aleph + x = x + \aleph = \aleph$ for any $x \in \mathbb{Z}$.

$$F_{X:=\Phi(Y,Z)}(in) = in \quad - \{X \to *\} \cup \{X \to c \mid Y \to c \in in \wedge Z \to c \in in\} \cup \{X \to \aleph \mid Y \to c_1 \in in \wedge Z \to c_2 \in in \wedge c_1 \neq c_2\}$$

Note that for readability reasons, all three flow functions do not handle constants directly. In the actual implementation, we check first whether a value is a constant, otherwise, we do the lookup in $in$.

$$F_{X:=Unknown}(in) = in \cup \{X \to \aleph\}$$

If an instruction unknown, i.e. no flow function is defined for it (e.g. function calls, floating point operations in our implementation), we set the result to not constant.

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

Due to SSA, all registers are only assigned once. In case a variable is assigned twice at different positions and the two control flows merge, LLVM creates two different registers that are merged using a PHI node. It is safe to union $in_1$ and $in_2$ without checking for duplicate registers, because every control flow must assign different registers. In mathematical terms, the following statement always holds true.

$$\forall v \in Vars : v \to c_1 \in in_1 \wedge v \to c_2 \in in_2 \Rightarrow c_1 = c_2$$

## 4.3 Implementation of Analysis

### *Data Structures.*

We represent lattice elements by a mapping from `llvm::Value*` to `MaybeInt64`, which is a tuple/C++ class representing an ASS (§3) over constant integer values. `MaybeInt64` contains an integer value and an `is_const` flag. Before the analysis begins, all nodes in the CFG are initialized with $\bot = \emptyset$.

### *Data Types.*

To simplify our implementation, we decided to handle only integer constants. We can easily extend our implementation to support floats and other data types as follows. Instead of integers, we store `llvm::Constant*` pointers, where `Constant` is an abstract LLVM class that could be a constant integer, a constant float, or any other kind of constant. We then simply create flow functions for differing types, as well as mitigate type differences as appropriate within flow functions for instructions that act on numerous types. LLVM guarantees that equivalent constants have the same pointer, making it easy to check if two registers contain the same constant.

### *Flow Function for Division.*

While folding a division or modulo operation, we check if the divisor is zero. In that case we exit with a divide-by-zero error.

### *Folding Complex Expressions.*

In LLVM, all arithmetic operations take two operands. When we compose a more complex expression, `clang` generates code that stores intermediate results in registers. LLVM has an infinite amount of registers and they are only assigned once (SSA). Since we assume that we run `mem2reg` first, we only handle registers in our analysis instead of memory locations[4]. Since complex expressions are split into multiple smallers ones, our analysis can infer their constant value by folding the smaller subexpressions.

## 4.4 Implementation of Transformation

Our program transformation consists of two steps: propagating constants and removing unnecessary instructions, i.e. instructions that compute constants.

### *Propagating Constants.*

Our framework invokes the optimization function for every instruction and the result of the flow function after this instruction (*out*). We iterate through all operands of the instruction and check if they are in *out* and comparable (constant). In such a case, we create a new LLVM constant object and replace the previous operand with it.

[4]Therefore, *Vars* does not denote the set variables but the set of registers, i.e. instructions in the program code.

*Removing Unnecessary Instructions.*

Our implementation assumes that we operate on the result of `mem2reg`. Therefore, all constants that were propagated in the previous step result from constant folding, because `mem2reg` puts constants directly into the instruction as operands instead of loading them from memory. All propagated values (i.e. the original operands) can now be removed from the IR completely. This effectively results in removing dynamic computation that was completed in constant folding, since these values *are* the instructions themselves in LLVM.

## 4.5 Benchmarks

In this section, we present an interesting case where a constant could be folded in theory, but it is not considered constant by our implementation. Consider the following C-style pseudo code.

```
int a = 12, b = 10;

if (*)
    a += 10;
else
    b += 10;

int c = a + b;
```

In this example, we do not detect that `c` is a constant. It is interesting to see the optimized LLVM IR code.

```
if.end:
    %a.0 = phi i32 [22, %then], [12, %else]
    %b.0 = phi i32 [10, %then], [20, %else]
    %add = add nsw i32 %a.0, %b.0
```

Although we do the constant folding for `a` and `b` independently, we cannot detect that their sum is always 32, because the PHI node is not constant and can, therefore, not be folded.

The solution is to adapt our flow functions. Instead of just checking whether the operands are both constant, also check whether both one or both operands are PHI nodes.

- Both operands are constant: fold the constants directly.
- One operand is constant, the other one is a PHI node: if constant folding was possible here, both operands of the PHI node would have to be the same constant. But then, we would already have folded the PHI node itself. Therefore, no constant folding is possible.
- Both operands are PHI nodes: if both PHI nodes share at least one common label in the operands (e.g. `[22, %then]` and `[10, %then]`) and the PHI nodes' operands are constant, fold the operands for both cases (e.g. $22 + 10$, $12 + 20$). If we get the same constant result in both cases (e.g. 32 and 32), fold the constants. If all labels are different, we have to check four different cases, but the same logic applies.

Now assume that at least one operand of the PHI nodes is not constant but another PHI node. In that case, we could consider both operands of this PHI node. But every time we add another PHI node to the calculation, the number of different paths doubles.

## 4.6 Lessons Learned

*Unknown Values in PHI Nodes.*

Although not shown above, we optimistically implement the PHI flow function by skipping unmapped operands for which we do not (yet) have any knowledge. The assumption is that they are either dead variables and therefore will never be chosen by the PHI node, or that they will eventually be known and thus mapped to a value which will force the worklist algorithm to eventually reiterate over the current basic block (with the PHI instruction) which will correct any misplaced optimism and any derived values from it.

*Constant Propagation during Analysis.*

The goal of the optimization step is to replace all known variables with the corresponding constants. Initially, we decided to do the replacement after analyzing an instruction (except for PHI nodes). For example, suppose we knew that $x \to 5$, and that we encountered the addition instruction $y = x + 2$. We were going to immediately replace any references of $y$ with $y = 5+2 = 7$. This, however, would introduce errors stemming from the prior paragraph. We did the replacement *before* the worklist algorithm completed. Thus, any replacement could potentially be incorrect, as the algorithm guarantees correctness only after it completes. To solve this problem, we ran the constant propagation step after the worklist algorithm was done. Similarly, throwing an error for divide-by-zero must be carefully implemented in case a later iteration updates the mapping of the variable to a non-zero value.

*Removing Instructions.*

We made another mistake while propagating constants, in which we replaced variables with constants within expressions. In other words, if $x \to 5$ and we encountered $y = x+2$ later, we would replace $x$ with 5 within the addition instruction. Effectively, the program would contain the instruction $y = 5 + 2$. Since $y$ is propagated to whereever it is used, the addition is redundant; removing it will not affect the program's result at all.

Given this assumption, we started removing instructions. Unfortunately, this violated two aspects of the program logic.

The first violation concerned the integrity of C++ iterators. Recall that an instruction was obtained as an iterator from basic blocks. While we went through all the instructions, we pre-incremented the iterator (i.e. `++iterator`) for performance. Removing an instruction would break the chain of iterator pointers; by the time the iterator was to be incremented, it had already been set to NULL, thus causing the application to seg-fault.

The solution was to post-increment the iterator instead. This actually stored the iterator value in a temporary register, incremented the original, and continued using the temporary value even though the original one had already been nullified. The chain of iterator pointers was broken. Nonetheless, we would not run into segmentation faults.

The second violation was related to references. We were removing instructions that were deemed redundant, but there could still be instructions in later basic blocks that

would refer to these redundant instructions[5]. These dependent instructions would face an invalid def-use chain—an actual error that LLVM would throw. The solution was to drop these references by calling `removeFromParent` and `dropAllReferences` on the redundant instructions.

These two solutions allowed us to safely remove instructions whose values had been propagated into the rest of the code. The optimization step reduced code size and could potentially enhance performance.

# 5. RANGE ANALYSIS

In this section, we present our must-be in range analysis that assigns every register a range/interval of possible values.

## 5.1 Assumptions

We developed and implemented this analysis under the following assumptions.

- For simplicity, each variable is of type `unsigned long`. We do not handle floats, and we do not handle the `sext` instruction.
- If we do not know a variable's upper bound, we set it to positive infinity. Likewise, an unknown lower bound is represented as negative infinity.
- We operate on the result of `mem2reg` and do not handle memory access.
- This is a must-be analysis. The analysis throws a warning only when we are absolutely certain of an impending out-of-bounds error.

## 5.2 Lattice Definition

Our lattice for the range analysis is a more general case of the constant analysis lattice and is defined as follows.

$$\begin{array}{llll} (D & ,\top & ,\bot,\sqcup,\sqcap,\sqsubseteq) & = \\ 2^{\{u\to v\,|\,u\in Vars\wedge v\in\mathbb{A}\}} & ,\{u\to\mathcal{I}\,|\,u\in Vars\} & ,\emptyset,\cup,\cap,\subseteq) & \end{array}$$

We define $\mathbb{A} = (\mathbb{Z}\cup\{-\infty,+\infty\})^2$ and $\mathcal{I} = (-\infty,+\infty)$. Note, that all ranges are considered bounds-inclusive.

### *Flow Functions.*

$$\begin{aligned} F_{X:=Y}(in) = in \quad & - \{X\to *\} \\ & \cup \{X\to r\,|\,Y\to r\in in\} \end{aligned}$$

$$\begin{aligned} F_{X:=Y+Z}(in) = \quad & in - \{X\to *\} \\ & \cup \{X\to(l,u)\,|Y\to(l_1,u_1)\in in \\ & \qquad \wedge\ Z\to(l_2,u_2)\in in \\ & \qquad \wedge\ l = l_1 + l_2 \\ & \qquad \wedge\ u = u_1 + u_2 \quad \} \end{aligned}$$

[5]This happens even though we replace the instruction in all operands of all instructions with a constant value in the first optimization function (constant propagation).

When we encounter an addition, we add the lower and upper ranges of both operands.

$$\begin{aligned} F_{X:=Y-Z}(in) = \quad & in - \{X\to *\} \\ & \cup \{X\to(l,u)\,|Y\to(l_1,u_1)\in in \\ & \qquad \wedge\ Z\to(l_2,u_2)\in in \\ & \qquad \wedge\ l = l_1 - u_2 \\ & \qquad \wedge\ u = u_1 - l_2 \quad \} \end{aligned}$$

Similarly to the addition case, we can come up with a simple scheme that computes the new lower and upper bound.

$$\begin{aligned} F_{X:=Y*Z}(in) = \quad & in - \{X\to *\} \\ & \cup \{X\to(l,u)\,|Y\to(l_1,u_1)\in in \\ & \qquad \wedge\ Z\to(l_2,u_2)\in in \\ & \qquad \wedge\ p_1 = l_1 * l_2 \\ & \qquad \wedge\ p_2 = l_1 * u_2 \\ & \qquad \wedge\ p_3 = u_1 * l_2 \\ & \qquad \wedge\ p_4 = u_1 * u_2 \\ & \qquad \wedge\ l = \min(p) \\ & \qquad \wedge\ u = \max(p) \quad \} \end{aligned}$$

For the multiplication case, have to consider all four possibilities of multiplying the lower and upper bounds. For this flow function, we assume that lower and upper bounds can also be negative.

$$\begin{aligned} F_{X:=\Phi(Y,Z)}(in) = \quad & in - \{X\to *\} \\ & \cup \{X\to(l,u)\,|Y\to(l_1,u_1)\in in \\ & \qquad \wedge\ Z\to(l_2,u_2)\in in \\ & \qquad \wedge\ l = \min(l_1,l_2) \\ & \qquad \wedge\ u = \max(u_1,u_2) \quad \} \end{aligned}$$

Note, that, for readability reasons, all three flow functions do not handle constants directly. In the actual implementation, we treat a constant $c$ as $(c,c)$.

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

## 5.3 Implementation of Analysis

### *Data Structures.*

We represent lattice elements by a mapping from `llvm::Value*` to `MaybeInt64Range`, which is a tuple/C++ class containing a constant integer range. Before the analysis begins, all nodes in the CFG are initialized with $\bot = \emptyset$. This is similar to the data structure in our constant analysis.

### *Data Types.*

To simplify our implementation, we decided to handle only integers. We can easily extend our implementation to support floats and other data types as follows. The same arguments as in the constant analysis also applies here.

## 5.4 Out-of-bounds Warnings

This step is an optimization that is run after the analysis is complete, i.e. after we can be sure that the result of the analysis is correct.

We iterate over all *GetElementPtr* instructions. This instruction is called to retrieve the pointer to a particular array position. We first obtain the length of the array and convert the pointer operand into a `CompositeType`, before casting it into an `ArrayType` and invoking the `getNumElements` method.

Given that we have obtained the length of the array ($[0, n]$ for instance), we must make sure that the index is within the bounds of the length. Using the map (lattice element) we have built during the analysis, we find the range of the array index ($[x, y]$ for instance). If neither ranges overlap (i.e. $y < 0$ or $n < x$), the array is guaranteed to be out of bounds, and we throw a stern warning at the user.

## 5.5 Branch Analysis

Our analysis takes into account simple branch statements where a variable is checked against a value with a known range. Consider the following C-style pseudo code.

```
int a = range(10, 90);
int b = range(5, 20);

if (a < b)
  ...
else
  ...
```

We are able to detect that in the *true* branch $a \rightarrow (10, 19), b \rightarrow (11, 20)$ and in the *false* branch $a \rightarrow (10, 90), b \rightarrow (5, 20)$. The following flow functions generate different lattice elements, depending on if the $<$[6] branch is taken or not.

$$\begin{aligned} F_{br\_true(Y<Z)}(in) = \quad & in - \{Y \rightarrow *\} - \{Z \rightarrow *\} \\ & \cup \ \{Y \rightarrow (l_y, a) \\ & \cup \ \{Z \rightarrow (b, u_z) \mid Y \rightarrow (l_y, u_y) \in in \\ & \qquad \wedge Z \rightarrow (l_z, u_z) \in in \\ & \qquad \wedge a = \min(u_y, u_z - 1) \\ & \qquad \wedge b = \max(l_y + 1, l_z)\} \end{aligned}$$

$$\begin{aligned} F_{br\_false(Y<Z)}(in) = \quad & in - \{Y \rightarrow *\} - \{Z \rightarrow *\} \\ & \cup \ \{Y \rightarrow (a, u_y) \\ & \cup \ \{Z \rightarrow (l_y, b) \mid Y \rightarrow (l_y, u_y) \in in \\ & \qquad \wedge Z \rightarrow (l_z, u_z) \in in \\ & \qquad \wedge a = \max(l_y, l_z) \\ & \qquad \wedge b = \min(u_y, u_z)\} \end{aligned}$$

## 5.6 Benchmarks

Our analysis was able to generate a warning for the program below.

[6]Other comparisons can be implemented in a similar way.

```
int arr[10];

int a = 0;
unsigned long i = 0;

while (i < 100) {
  a = a + i;
  i = i + 1;
}

return arr[i];
```

At the end of the *while* loop, our analysis concludes that $i$ will be in the range $[100, \infty)$, which clearly does not overlap with $[0, 9]$ of the array. On the other hand, if we replace `arr[i]` with `arr[a]`, the analysis will not produce any warnings. Based on our implementation, we do not have sufficient information about $a$; for all we know, $a$ could start from zero and go all the way to infinity. Thus, $a$'s range intersects with the array's range. The analysis has no definitive evidence to throw a nasty warning in this case.

## 5.7 Lessons

Only toward the end of implementing range analysis did we realize that constant analysis was, in fact, a special case of range analysis. Effectively, a constant $c$ is equivalent to a range $[c, c]$. For coding efficiency, we could have designed the range analysis first, before making constant analysis a special case of the range analysis.

# 6. INTRA-PROCEDURAL POINTER ANALYSIS

In this section, we present our must-point-to analysis.

## 6.1 Assumptions

LLVM's default `mem2reg` pass already implements the optimization for the analysis. If, for instance, $a$ points to $b$ in the C++ file, then all occurrences of $a$ are automatically replaced with $b$ in the byte-code. In order to implement our own must-point-to analysis, we must therefore disable the `mem2reg` pass.

The goal of the analysis is to output a mapping from one memory location, $a$, to another memory location, $b$, if $a$ must point to $b$.

## 6.2 Lattice Definition

We define the lattice for the pointer analysis as follows. $x \rightarrow\!\!\!\!\!\!\diagup$ means that $x$ does not point to one thing, but it may point to multiple things.

$$\begin{array}{lll} (D & ,\top & ,\bot, \sqcup, \sqcap, \sqsubseteq\ ) \\ (2^{\{x \rightarrow y \mid x, y \in Vars\}} & ,\{x \rightarrow\!\!\!\!\!\!\diagup \mid x \in Vars\} & ,\emptyset, \cup, \cap, \subseteq\ ) \end{array} =$$

We define flow functions at the IR level as follows.

$$F_{store(X,Y)}(in) = in - \{Y \rightarrow *\} \cup \{Y \rightarrow X\}$$

$$F_{X:=load(Y)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow Y\}$$

$$F_{Default}(in) = in$$

We argue that by defining the flow functions for `store` and `load` instructions alone, we are able to capture the behaviors of all four types of pointer manipulations: `X = Y`, `X = &Y`, `X = *Y`, and `*X = Y`.

$$F_{merge}(in_1, in_2) = in_1 \cup \{X \to Y \in in_2 \mid X \to * \notin in_1\} \cup \{X \to \not\curvearrowright \mid X \to a \in in_1 \wedge X \to b \in in_2 \wedge a \neq b\}$$

## 6.3 Implementation of Analysis

### *Data Structures.*

Our lattice element is represented by a map that maps `llvm::Value*` to instances of `PointsTo`, which is a C++ class that stores another `llvm::Value*` and an `is_null` flag.

### *Initialization.*

First, we start with a high level view of the analysis. At the initialization stage, we assign $\perp$ to each of the instructions in the control-flow graph. Internally, we also maintain a map from variables to variables (as defined in the domain of the lattice). If a variable $a$ is mapped to another variable $b$, we conclude that $a$ points to $b$. However, if $a \to \not\curvearrowright$[7], then we conclude that $a$ points to a value that we cannot determine, and that its value is incomparable to those of the others in the lattice. On the other hand, a variable absent from the map suggests a lack of evidence about this variable. Note that variables in this map can be either memory locations or registers in the byte-code. We will later explain how this difference is resolved.

### *Worklist algorithm.*

Now that the map is initialized, we can start the worklist algorithm. Our analysis changes the state of the lattice only upon `store` and `load` instructions. Otherwise, the output of the flow function is exactly the input.

When we are loading $X$ into $Y$, the mapping from $Y$ to $X$ is added. Similarly, when the value of $X$ is stored into $Y$, we add $Y \to X$ into the map. Any existing keys in the map will be replaced, as the corresponding variable now points to a new value.

The maps are intersected when two basic blocks merge. For example, we may have $X \to Y$ in the first basic block, and $X \to Z$ in the second. The result of the merge depends on $Y$ and $Z$. If they are the same, the map after the merge will have $X \to Y$. Otherwise, we require that $X \to \not\curvearrowright$, as $X$ must point to an incomparable value.

This process is repeated until we reach a fixed point. The worklist algorithm terminates. Our analysis prints out the contents of the map. Unfortunately, we cannot hand the map over to the optimization pass yet, as the map contains both memory locations and temporary registers that LLVM creates. For example, a memory location may be dereferenced first before being stored to another value. LLVM would automatically create a new register to temporarily store the dereferenced value.

[7]Represented in the code by a flag `is_null`.

Ultimately, the goal of the pointer analysis is to output a mapping of memory locations, which the original program creates. We should ignore intermediate registers in the analysis. If, for instance, $a \to \%1 \to b$ (where %1 is a temporary register, following LLVM's naming convention), then our analysis should output $a \to b$.

### *Eliminating temporary registers.*

Recall that our map maintains relations such as $a \to b$, where $a$ and $b$ could each be a memory location or a temporary register. Our objective is to follow this chain of must-pointers, until we can determine that a memory location points to another memory location.

We illustrate the problem with the following C++ program.

```
int x = 666;
int* a; int* b;
b = &x; a = b;
```

In the LLVM intermediate representation, we observe that $b \to x$, but $\%0 \to b$ while $a \to \%0$. The goal is to eliminate temporary registers such as %0, such that the analysis would eventually conclude that both $a$ and $b$ point to $x$.

To this end, we transform the map into a special pointer graph[8]. Each node in the graph is a key in the original map; thus, a node can be either a memory location (e.g. $a$) or a temporary register (e.g. %1). Each node has a type. If we follow the example above, the $x$ node would have type $int$, $b$ would have type $int*$, %0 type $int$, and $b$ type $int*$. A directed edge from node $b$ to node $x$ suggests that there is a mapping from variable $b$ to variable $x$, which flow functions constructed during the worklist algorithm.

We can establish must-point-to relationships by traversing the graph. Suppose we are interested in what $x$ must point to, such that $x$ is a memory location of type $int*^n$. This notation means that the pointer type has $n$ asterisks. A pointer-pointer is expressed as $int*^2$.

We recursively follow the outgoing edges from node $x$, until we arrive at some special node $y$, such that:

- $y$ is a memory location, and
- $y$ is of type $int*^{(n-1)}$.

We can subsequently conclude that $y$ must point to $x$. If no such $y$ can be found or if $x$ is encountered along our path, then $y$ is incomparable, there is insufficient information to establish must-point-to relationship about $y$.

For each memory location in the map, we carry out the graph traversal above. Our analysis outputs all such $x \to y$ pairs, which subsequent passes can potentially leverage for optimization. For a full example of how we eliminate registers with the graph, refer to §8.2.

## 6.4 Benchmarks

We design two benchmark programs to test our analysis.

The first benchmark verifies that our analysis can handle all possible pointer operations. As shown in §8.2, the C program attempts various combinations of pointer operations. Furthermore, the program creates two integer pointer types:

[8]We have to traverse this graph to produce the correct output during the printing phase.

$int*$ and $int**$. It constantly converts between the two, in a way that is representative of the difficult corner cases.

Our analysis can handle these situations, because they all boil down to the `load` and `store` instructions at the LLVM level. The graph-traversal algorithm is able to follow memory locations and temporary registers with ease. It can also keep track of various $int*^n$ pointer types, so that we can easily pinpoint the must-point-to relations between any two memory locations.

The second benchmark examines how must-point-to relations are propagated between basic blocks. Shown below is a snippet of the program:

```
     int y = 4; int* c;
L2:  if (y > 0) { c = &y; } else { c = &x; }

     int* d = &y;
L4:  while (y > 0) { d = &x; d = &y; }
```

The analysis is able to show that both $c$ must point to $\not\curlywedge$ (incomparable), as $c$ points to two different memory locations at the merge point on L2. Hence, we have insufficient information on what $c$ must point to. In contrast, the analysis concludes that $d$ must point to $y$, since $d$ points to $y$ at both merged points on L4.

However, our analysis will not work with arrays, where `getElementPtr` is used to handle reading. We believe it an easy task to add support for this new instruction.

## 6.5 Lessons Learned

Initially, we started designing the analysis with `mem2reg` in mind, similar to the two analyses that we had done so far. This built-in pass offers some undeniable perks, such as SSA, that would significantly simplify our analysis passes. In the case of pointer analysis, however, the `mem2reg` pass would be too smart—it would do the optimization for us.

Again, consider the following code snippet:

```
int x = 666;
int* a; int* b;
b = &x; a = b;
```

The `mem2reg` pass would simply render our analysis obsolete, as it already takes care of constant propagation and pointer analysis:

```
%add = add nsw i32 666, 666
%add1 = add nsw i32 %add, 666
```

If we replaced the initial assignment of $x$ with a random number generator (i.e. a `call` instruction), textttmem2reg would be smart enough to replace all occurrences of 666 with the `call` instruction. Effectively, `mem2reg` was replacing available expression for us as well. Therefore, we had to disable `mem2reg`.

As SSA disappeared, we could only do *join* when two basic blocks merged rather than at Phi nodes.

# 7. AVAILABLE EXPRESSIONS ANALYSIS

In this section, we present our available expression analysis.

## 7.1 Overview

We once again enable LLVM's default `mem2reg` pass to provide SSA for non-memory locations and build our design atop of such assumptions.

The goal of this analysis is to output the expression that each variable is assigned a value from, which could enable optimizations such as Common Sub-expression Elimination. From a high-level, our analysis constructs a mapping of variables to expressions for which any pair of expressions can be checked for equivalency. We chose to support three primary forms of expressions based upon operation formats: unary, unordered binary, and ordered binary. Respective examples of such operations would be: bangs/not operators; addition and multiplication; and subtraction and division.

Our analysis constructs custom expressions composed of the analyzed LLVM instruction's operation and the value(s)/variable(s) passed to it. Equivalency between expressions is defined via equivalent values (respecting ordering as appropriate) with equivalent operations.

SSA guarantees that registers are immutable, so long as they are not assigned values that derive from a PHI node. To this extent, expressions can also be set to incomparable (§3). This is commonly the expression of a PHI node or any expressions that contain values descendent of such PHI nodes.

Once the mapping pairs are created from variables to expressions, any pair with comparable expressions can be used in succeeding optimization passes. The obvious example would pass pairs with equivalent expressions onto a CSE optimizer, which could replace all dominated occurrences of such expressions with the variable corresponding to the dominating pair. This could then be optimized further via another pass, Copy Propagation.

The rest of the section will focus specifically on Available Expressions Analysis, leaving optimization passes as future consumers of our products. To do so, the analysis syntax and lattice will be formalized, followed by details of our specific implementation's design of the requirements, easy-to-follow benchmarks, and finally lessons and minor obstacles encountered during implementation.

## 7.2 Syntax

All *variables*, like in other passes with `mem2reg` enabled, are values stored in an infinite set of registers with uniquely identifiable names. Allocating of, storing to and loading from memory locations, despite being representative of variables in higher level languages, are not so in this analysis. The register holding the value returned by an allocate or load, however, is considered a variable like any other register. All *expressions* are represented as described in the Overview (§7.1). The following syntax will be used throughout the section to describe all variables and expressions:

**Var** a variable

**Vars** full set of all variables

$\mathbf{Expr}_{\emptyset}$ a non-existent expression (not yet known)

$\mathbf{Expr}_{C}$ a comparable expression

$\mathbf{Expr}_{I}$ an incomparable expression

$\mathbf{Expr}^{O}$ an ordered expression (discussed below)

$\mathbf{Expr}^{U}$ an unordered expression (discussed below)

**Expr** base type for any expression

## 7.3 Lattice Definition

We define the lattice for available expressions as follows.

$$(D, \top, \bot, \sqcup, \sqcap, \sqsubseteq\ ) =$$
$$(2^{\{x \to y \,|\, x \in Vars \wedge y \in Exprs\}}, \{x \to Expr_I \,|\, x \in Vars\}, \emptyset, \cup, \cap, \subseteq\ )$$

We define the flow functions at the IR level as follows.

$$\begin{aligned} F_{X:=Y+Z}(in) = \quad & in \\ & \cup\ \{X \to Expr^U_C(+, Y, Z)\ | \\ & \quad Y \to Expr_I \notin in \wedge Z \to Expr_I \notin in\} \\ & \cup\ \{X \to Expr_I\ | \\ & \quad Y \to Expr_I \in in \vee Z \to Expr_I \in in\} \end{aligned}$$

$$\begin{aligned} F_{X:=Y-Z}(in) = \quad & in \\ & \cup\ \{X \to Expr^O_C(-, Y, Z)\ | \\ & \quad Y \to Expr_I \notin in \wedge Z \to Expr_I \notin in\} \\ & \cup\ \{X \to Expr_I\ | \\ & \quad Y \to Expr_I \in in \vee Z \to Expr_I \in in\} \end{aligned}$$

$$\begin{aligned} F_{X:=\Phi(Y_1,\ldots,Y_n)}(in) = \quad & in - \{X \to *\} \\ & \cup\ \{X \to c \,|\, \exists c\, \forall i \in \{1 \ldots n\} : \\ & \quad Y_i \to c \in in \vee Y_i \to Expr_\emptyset \in in\} \\ & \cup\ \{X \to Expr_I \,|\, \exists i, j \in \{1 \ldots n\} : \\ & \quad Y_i \to e_1 \in in \wedge Y_j \to e_2 \in in \\ & \qquad \wedge\, e_1 \neq e_2\} \end{aligned}$$

$$F_{Unknown}(in) = in \cup \{Unknown \to Expr_I\}$$

We argue that by defining flow functions for the `add` and `sub` instructions alone, we can demonstrate expression equivalence for all binary operations on signed integers: ordered and unordered. Unfortunately, we did not locate a unary operator that was logical for signed integers. This is further discussed in the implementation below.

## 7.4 Implementation of Analysis

### *Data Structures.*

We represent expressions with an `Expression` class composed of two `llvm::Value*` operands and a `std::string` representation of the instruction operation. Two `Expression` objects are equal if their respective member fields are equal.

Within the `Expression` class, the operands are always considered ordered, despite our declaration of unordered-binary and unary expressions. This was done for ease of design, and generalized via three static factory methods: `Unary(...)`, `BinaryOrdered(...)` and `BinaryUnordered(...)`. Both `Unary` and `BinaryUnordered` include minimal setup and then delegate actual creation to `BinaryOrdered`. `BinaryOrdered` stores the values in the order provided. `Unary` passes the *NULL* value as the second operand. `BinaryUnordered` sorts the two operands (via less-than) and then passes them to `BinaryOrdered`; this internal ordering is acceptable so long as external users need not care about the order they pass the values in. This reordering is further acceptable since an unordered instruction and and ordered instruction should never share the same string representation, thus mutual exclusivity of instructions remains.

Specifically, the following two LLVM instructions would equate to different variables (registers) that point to equivalent expressions:

```
%add2 = add nsw i32 %1, %2
...
%add3 = add nsw i32 %2, %1
```

Of particular interest regarding these instructions is that the operands are ordered differently, however the expression portions are still equivalent, so long as the values are not modified between the two instructions, as guaranteed via SSA.

### *Data Types.*

To simplify our implementation, we limited our scope to data types that are passed to `add` and `sub` LLVM instructions. We can easily extend our implementation to support floats and other data types by adding additional flow functions for them. Our mapping of variables is done via `llvm::Value*`, thus it can handle any data type permissible in LLVM IR.

### *Φ Nodes and SSA.*

Φ instructions have a unique flow function in this analysis since we may not yet have seen all the instructions that define its use. We optimistically ignore any unknown values, knowing that we will eventually know some information for them and will therefore revisit this Φ instruction again via the worklist algorithm. If all the known values map to equivalent *Expr*s, we may map an equivalent *Expr* to the *var* storing the Φ instruction result. If any two *Expr*s are nonequivalent, we instead map $Expr_I$ to this instruction's respective *var*.

SSA allows us our catch-all default flow function to return the lattice element with the only modification being that the current instruction is set to a conservative, incomparable ASS. This is not a lack of precision since assignment can only happen in a single location, and thus nothing else could ever override this value. Further, this is neccessary in the event that this instruction is later used as a Φ operand, in which case we should not optimistically ignore the value.

## 7.5 Benchmarks

For Available Expression Analysis we will demonstrate two simplistic examples that cover: equivalent expressions over the generalized unordered operation `add`, and two non-equivalent expressions that result from a Φ instruction. Other cases were also tested but omitted for brevity (e.g. false-positive `sub` expressions due to ordering).

For brevity in the benchmarks below, we omitted declarations of integers *a*, *b*, *y*, *yyy*, and *bbb* which are initialized to random values if read before assignment. (Also for clarity, as described in §7.6.) Returns have also been omitted.

*Unordered Equivalence.*

```
int x = a + b;
if (a > 0) {
  y = b + a;
} else {
  y = a + b;
}
```

Although $x$ and the $y$ within the *false* branch would map to equivalent expressions in an ordered equivalence check, the expression mapped to $y$ in the *true* branch would not. In this case, a later reading of $y$ would result in a $\Phi$ node to aggregate the two values, and result in $y \rightarrow Expr_I$.

Our analysis, however, goes further as it will correctly map all three assignments to equivalent expressions. A $\Phi$ node further down can then precisely aggregate the two $y$ assignments and result in an $Expr_C$ as opposed to $Expr_I$. The $x$ value could also be aggregated should such a $\Phi$ node exist with operands $x$ and $y$, or for optimization passes.

*False-Positive Equivalence.*

```
int aaa = yyy + bbb;
while (aaa > 0) {
  yyy = yyy + bbb;
}
```

At first glance, it may appear that *aaa* and *yyy* map to equivalent expressions, but that would be incorrect, as *yyy* may mutate to other values. Our analysis properly distinguishes the difference due to the *yyy* within the *while* loop being a $\Phi$ node with two operands: the initial *yyy* initialization, and the *var* storing the result of the addition of the $\Phi$ instruction and *bbb*. On the first iteration, the $\Phi$ node will only have knowledge about the initialization of *yyy*. On the next iteration, it will recognize the inequivalence of the two operands and thus map $Expr_I$ to the *var* holding the result of the $\Phi$ instruction.

## 7.6 Lessons Learned

Our analysis does not take care of nested equivalence. For example, the analysis does not yield a precise, meaningful result for the following pseudo-IR code:

```
a = add 66, 77
b = add 77, 66
c = add a, 88
d = add b, 88
```

As expected, the analysis considers `add 66, 77` and `add 77, 66` to be the same expression; recall that the ordering does not matter for unordered instructions like `add`. However, $a$ and $b$ are fundamentally different memory locations (or registers). Thus, `add a, 88` and `add b, 88` are considered different expressions even though $a$ and $b$ practically refer to the same expression. Effectively, our analysis fails for nested equivalence.

Adding support for this case is trivial. The idea is similar to our constant folding and propagation: for Available Expression Analysis, we simply replace the equivalence check of $a$ and $b$ with the corresponding equivalence check on their mapped expressions. In this way, $a$ and $b$ are equivalent; so are $c$ and $d$. We believe that implementing this feature will introduce diminishing intellectual merits, and that we shall leave this to future generations of enthusiastic readers to implement. Be aware that a naive implementation of this improvement will make numerous recursive calls each time nested expressions are checked for equivalence.

We have also learned another lesson that involves `mem2reg`. Earlier, we have found that `mem2reg` introduces SSA, but it also optimizes away variables, pointers and expressions that we wished to analyze. For this analysis, `mem2reg` does the constant folding and propagation for us. The code above will not have $a$ or $b$; `mem2reg` will have replaced them with their constant values.

To fool `mem2reg`, we introduce "constant" values through `call` instructions. Where constants should have occurred, we use the `call` instructions. The code above would read:

```
a = call ...
b = call ...
c = add a, b
d = add b, a
```

For our analysis to work with `call`, we need one additional hack. We introduce a flow function for `call`, where we ensure that the assignment is considered a different value at each line (even to the same function, as we do not have side-effect free guarantees); otherwise, $a$ and $b$ may inadvertently become equivalent. To do this, we first observe that every instruction has a unique memory address. An expression involving `call` can be identified by the memory address of the `call` instruction. In this way, $a$ and $b$ are expressions with unique identifiers. Essentially, this hack forces $a$ and $b$ to behave as different constants, while we are able to keep `mem2reg` for the benefits of SSA. Equivalently, we could have used function parameters.

# 8. APPENDIX

## 8.1 Branch Analysis Example

In §5.5, we show an example for our branch analysis as part of the range analysis. Figure 4 shows the the ranges of the variables graphically.

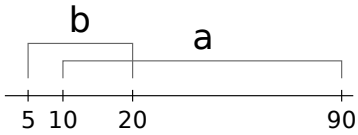



Figure 4: Visualization of the ranges of $a$ and $b$.

## 8.2 Eliminating temporary registers

In §6, we describe our must-point-to analysis, where we eliminate temporary registers that LLVM creates in the byte-code. These temporary registers prevent us from directly establishing must-point-to relationships among memory locations. In this section, we delineate the register-elimination algorithm using a full example.

Suppose we have the following C code:

```
int x = 666;
int* a; int* b; int** c; int* d; int** e;

L1: b = &x;
```

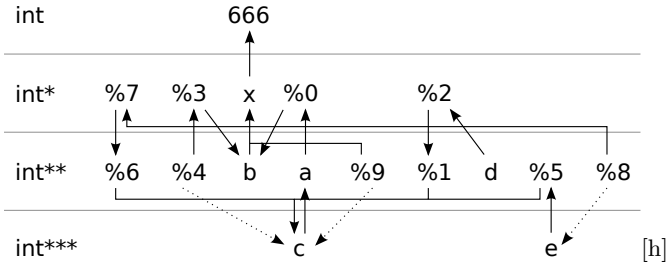



Figure 5: Points-to example graph. Note that even though the C type of $x$ is $int$, it becomes $int*$ when compiled down to the LLVM intermediate representation.

```
L2: a = b;
L3: c = &a;
L4: d = *c;
L5: *c = b;
L6: e = c;
L7: *e = *c;
L8: *c = &x;
```

LLVM translates L1 into the following code:

```
store i32* %x, i32** %b, align 8
```

We create an edge from $b$ to $x$. Similarly, L2 is translated into the following intermediate representation:

```
%0 = load i32** %b, align 8
store i32* %0, i32** %a, align 8
```

We create an edge from %0 to $b$, and from $a$ to %0. In this way, we can construct the full must-point-to graph as shown in Figure 5.

Should the must-point-to relationships change, we update the edge accordingly. For example, %4 points to $c$ initially, but it later points to %3 instead. Thus, we mark the edge from %4 to $c$ as a thin, dashed line to be removed, while the edge from %4 to %3 is a solid line.

With the complete graph, we can output the result for further optimization. Suppose we are interested in what $e$ points to. Using Figure 5, we first trace from $d$ (whose type is $int*$) to %2. Even though %2 has one fewer pointer asterisk than $d$, it is not a memory location. Hence, we keep traversing the graph, passing through %1, $c$, $a$, %0, $b$, and ultimately $x$, which is both a memory location and of type $int$. Therefore, $d$ must point to $x$.