

1 Intermediate Representation

Here begins the code optimization part. The compiler pipe line begins with source code which is unoptimized, having specifications of the program logic by human programmer. On the other end of the pipeline, we have assembly code which actually gets executed on machine. The question we want to address here is: where to do the optimization? Here are three different options along with their pros and cons:

- **Optimize at source code level** : For e.g, we can run optimization on the abstract syntax tree(AST) of the programming language. But then, we will have to deal with a lot of features which are very specific to a programming language. For e.g., C has *for* and *while* loops but optimization does not care whether loops are written using *for* or *while*, it just needs to know that there is a loop. Also, sometimes we need information about the architecture to do architecture specific optimizations but if we do the optimization at source code level, we do not have visibility into the architecture.
- **Optimize at assembly language**: This level has much less features, usually nothing specific to any programming language. But here the optimization space is very limited because most of the decisions are already taken by the time we reach the assembly code.
- **Optimize the intermediate representation(IR)** : Most compilers have intermediate representation(IR). The AST is translated to IR and then optimized IR is converted to assembly code. What is a good intermediate representation? The IR should be simple enough, should not have unnecessary program features(e.g., *for* vs *while* loop differentiation). Also we want to make minimum commitments when designing IR so that we have large optimization space. Essentially there is a tradeoff in designing the IR:
 - IR too close to assembly - there is smaller optimization space
 - IR too close to source - loose architecture specification optimizations.

More Software Engineering advantages of IR:

We can have n number of source code languages on the source code side and similarly, we can have m number of instruction set architectures(ISA) like x86 etc on the architecture side. If we write one IR that can serve both different source code languages and architectures, then we just need to write optimizers once for that IR. We won't have to write different optimizers for each language or each ISA. But then there is one more issue here: The types of constructs needed to support C(lower level language than python) program are different from the type of constructs needed to support python program. IR design can be considered more art than science and is constantly evolving.

In reality, its difficult to design one IR, so there are multiple IRs -(in the range of 3-5). Each IR has its own level of abstraction. Each level will have different optimizations, because different types of optimizations require different abstractions. This can be explained with an example: Consider *Tensorflow* :

- In *Tensorflow*, source program is represented as graph.
- Nodes are operators. Edges represent data flow.
- Datatypes are tensors(multi-Dimensional arrays).

Some example optimizations are:

- **Matrix level rewrites** : $A * A^{-1} = I$. (For this, we need to know that A is a matrix and $*$ is multiplication operation). Such operations are easier closer to the source code because by the time we reach assembly code, we lose track of this high level information.
- **Parallelization operation**: This needs dependency information which is in the form of matrices, so better to do this optimization at higher level than lower level.
- **Locality** : There is fixed size architecture, we need to organize data so that the cache is maximum utilized. Optimizations like these are done in higher level IRs.
- **Register Allocation** : These need to be done at lower level of abstractions, so these are done at lower level IRs.

2 Three Address Code

2.1 IR for Assembly Code Generation

This is low level IR and is meant for assembly code generation. Typically this IR is present in compiler tool chains of almost all languages. IR resembles high-level assembly. Some properties for such IR are:

- They have register names similar to assembly language. Unlike assembly code, they have unlimited numbers of registers.
- They have control structures like conditional and unconditional branches(similar to assembly). They are lower level than if-else(in C) etc.
- They have opcodes. They have higher level opcodes, something like doing vector addition. These opcodes are very similar to the ones in assembly which make their conversion to assembly opcodes easier. Sometimes IR opcodes could be lower level as well. For eg., some assembly opcodes can do multiple operations (*add*, *multiply*) in one instruction which might not be the case with the IR opcode. These need to be kept into mind while converting from IR to assembly code. An opcode is higher level if its translation to assembly code results in more than 1 instruction. If more than one opcodes translate to one instruction in assembly code, then it is considered lower level opcode.

2.2 IR Characteristics

Each instruction is of the form

$$x = y \text{ op } z \text{ (binary operation)}$$

or

$$x = \text{op } y \text{ (unary operation)}$$

where y and z are registers or constants, can be scalars or vectors. This type of IR is called *Three-Address Code*. This is because, for each operation, there are at max 3 addresses (x, y, z). Addresses are registers or memory addresses.

2.3 Three Address Code Property

Expression $x + y * z$ translates to

$$t1 = y * z$$
$$t2 = x + t1$$

Property : Each sub-expression has a name which makes compilation easier later on from the perspective of optimization. Assembly operations such as *load* and *store* are also supported.

2.4 IR Code Generation

This is very similar to assembly code generation.

Define function $igen(e, t)$ as a function which

(INPUT) : takes an expression e and a value t . t represents the register name in which the value of expression should be computed.

(OUTPUT) : $igen(e, t)$ generates code to compute the value of expression e in register t . (Currently assuming that there are unlimited number of registers)

2.4.1 IR CodeGen Example

$$\begin{aligned} igen(e_1 + e_2, t) = & \\ & igen(e_1, t_1) \\ & igen(e_2, t_2) \\ & t = t_1 + t_2 \end{aligned}$$

t_1 and t_2 are fresh registers.

2.4.2 Three Address Code IR

LLVM (Low level Virtual Machine) is the one of most popular *Three Address Code* IRs.

Types in LLVM are :

(integer) $i1, i8, i32..$ ($i1$ stands for 1-bit integer and so on..)

(float)

(pointers) $i32^*, i32^{**}..$

(vector types) $\langle 4 * i32 \rangle$

(struct types) $\{i32, i16\}$

The size of the integer is not consequential to optimization, so we make this decision by the time we reach IR.

Risc-like opcodes : *LLVM* has opcodes like add, load, store, call, ret, branch, conditional branch... Unlike assembly code, these opcodes can have unbounded number of arguments.

3 Module 70 : Static Single Assignment IR

Consider the following example

$$x = y + z$$

$$x = x + 1$$

$$w = y + z$$

$$z = x + 3$$

In this example, the computation $y+z$ could be reused. In SSA, we assign versions to variables(as below) and each version has only one assignment to it. SSA stands for single assignment in a static program.

$$x_1 = y + z$$

$$x_2 = x_1 + 1$$

$$w_1 = y + z$$

$$v_1 = x_2 + 3$$

Using SSA, we can not optimize the code by rewriting $w_1 = y + z$ as $w_1 = x_1$.

$$x_1 = y + z$$

$$x_2 = x_1 + 1$$

$$w_1 = x_1$$

$$v_1 = x_2 + 3$$

Why SSA? Advantages of SSA:

- Optimization algorithms become simpler if each variable has only one definition.
- Unrelated uses of same variable become independent.
- More values become available at each program point.

Therefore, SSA is a very popular method of IR design. LLVM IR is also an SSA IR.

3.1 Converting to SSA

- Replace the target of each assignment with a new variable.
- Replace each use of a variable with the version of the variable reaching that point.



Figure 1: Converting to SSA Example

Again, taking an example:

$$x = y + z$$

$$a = b + x$$

$$x = a + 3$$

$$y = x - a$$

On applying the two rules for conversion to SSA, these statements change to :

$$x_1 = y + z$$

$$a = b + x_1$$

$$x_2 = a + 3$$

$$y = x_2 - a$$

Consider another example as shown in Figure 1:

In this example, there are 2 variables of interest. x is assigned thrice and y is assigned thrice. On applying the rules of SSA conversion, here is how the code snippet looks like.

The versioning is straightforward for variable x . For y in this example, there are two versions reaching at the bottom block from two different paths. The question is how we version the variable y at bottom block. To answer that , we need to understand about basic block

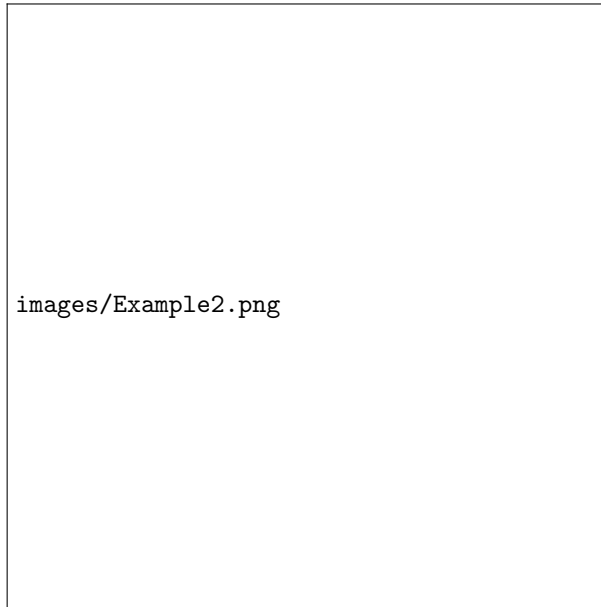


Figure 2: Converting to SSA Example - With Versions

3.2 Basic Block

: A basic block is a maximal set of instructions with

- no labels(except at the first instruction)
- no jumps (except in the last instruction)

In the example, each rectangle is a basic block. Idea:

- Cannot jump into a basic block (except at beginning)
- cannot jump out of a basic block (except at end)
- Single-entry single-exit straight line code segment

At the bottom block, there are two version reaching for variable y . To represent that we use Φ node or Φ function. So the version of y can be represented as a function of y_1 and y_2 as shown in figure XXX. This is an ordered set.

3.3 PHI(Φ) Nodes

- Φ function chooses the version depending on the incoming edge.
- Present only at the beginning of a basic block. Since this is the only place where there are multiple versions flowing through different incoming edges.

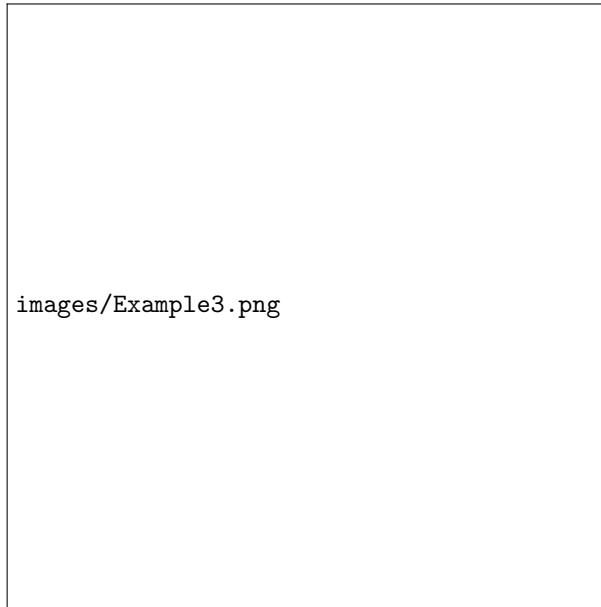


Figure 3: Converting to SSA Example - PHI Nodes

4 Phi Nodes

In SSA IR, we encountered the issue of figuring out which variable version to use after a join point where multiple paths were coming in with different versions of the same variable. As a solution to this, phi nodes or phi functions are extremely helpful that could be placed only at the beginning of a basic block with multiple edges coming in (join point).

These phi nodes can be placed at each join point for every variable in the program. This placement strategy is a bit wasteful as illustrated in the following diagram:

4.1 Path Convergence Criterion

Consider z to be node with multiple edges coming in (thus it is a join point). We need Φ node for variable a at node z if and only if

1. There is a block x containing the definition of a .
2. There is a block y (not x) containing the definition of a .
3. There are non empty paths P_{xz} and P_{yz} from x to z and y to z respectively.
4. P_{xz} and P_{yz} should not have any node in common except z .
5. The node z does not appear within both P_{xz} and P_{yz} , prior to the end. It is fine if z appears only in one of the path before end.



Figure 4: Placement of Φ nodes

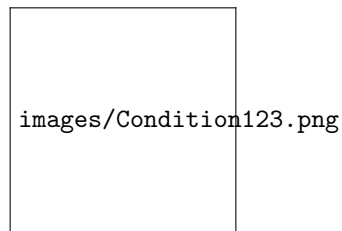


Figure 5: CFG for condition 1, 2, 3

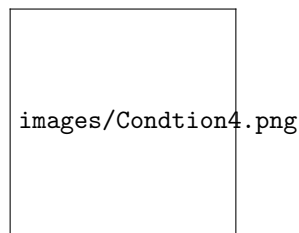


Figure 6: Violation of Condition 4

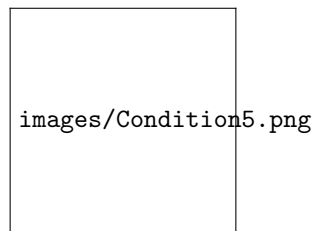


Figure 7: Condition 5: Left CFG does not need Φ node, right CFG needs Φ node

4.2 Iterative Fixed Point Algorithm

while {there are nodes x, y, z satisfying condition 1-5 and z does not contain a Φ node for a }
do {insert $a \leftarrow \Phi(a_1, a_2, \dots, a_j)$ }

The Φ function has as many 'a' arguments as there are predecessors of z . Since the conditions 1-5 are both sufficient and necessary, the above algorithm is sound and complete.

5 Optimization Overview

Most intermediate representations are organised as *control flow graphs (CFG)* over *basic blocks*.

5.1 Basic Blocks

A basic block is a single entry, single exit, straight line code segment. More formally, it is a maximal sequence of instructions with no labels (except at the first instruction) and no jumps (except at the last instruction).

In the following figure, (a) is a basic block. (b) is not a basic block because it has multiple exits; (c) is not a basic block because it has multiple entries; (d) is not a basic block because it does not represent a straight line code.

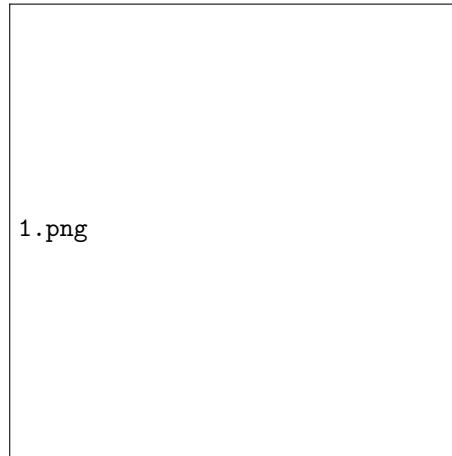


Figure 8: Examples and Counterexamples for basic block

Consider the following example of a single basic block:

1. L:
2. $t := 2 * x$
3. $w := t + x$
4. if $w > 3$ goto L'

- Is it ok to change (3) to $w := 3 * x$? It is ok. If addition of two numbers is more expensive as compared to multiplication of a number with a constant, the above change can be considered an optimization.
- Is it ok to change (4) to if $x > 1$ goto L' ? It is not ok. Because here x is a finite bounded integer and due to overflow, it is possible to have $3 * x > 3$ and not $x > 1$.
- Is it ok to remove (2) ? Depends on whether variable t is used later in the code or not.

5.2 Control Flow Graph

A control flow graph is a directed graph with

- basic blocks as nodes
- edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B, for example
 - the last instruction in A is: jump Lb
 - the last instruction in A is: if $id1 = id2$ then goto Lb
 - execution can fall-through from block A to block B

Consider the following example of a CFG:

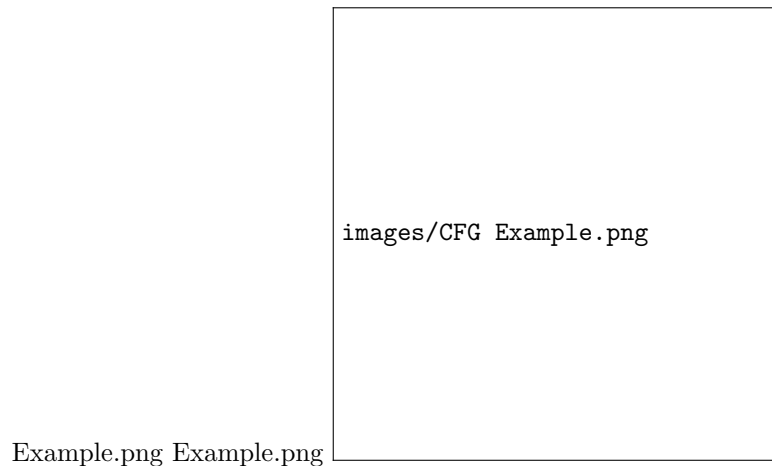


Figure 9: CFG Example

The body of a method (or procedure) can be represented as a control-flow graph. There is one initial node (entry node). All "return" nodes are terminal.

5.3 Optimization

The optimizations are performed on the control flow graph of the intermediate representation of the code to improve program's resource utilization:

- Execution Time
- Code Size
- Memory Usage
- Frequency of Disk I/O operations (or network operations)
- Power Consumption (Not same as energy)

It is important to remember that optimization should not alter the meaning of the program. It should not alter what the program computes.

For example, the following optimization changes the meaning of the program and thus, it is not a valid optimization.

```
w := 3 * x if w > 3 goto L
cannot be converted to: if x > 1 goto L
```

5.4 Typical Granularity of Optimization

1. Local Optimizations

- applied to a basic block in isolation
- easiest to implement

2. Global Optimizations

- applied to a CFG (method body) in isolation while crossing the boundaries of basic blocks

3. Inter-Procedural Optimizations

- applied across method (CFG) boundaries
- difficult to implement but usually most effective

5.5 Economics of the Optimization

Optimizations are more of an art rather than science. The current state of the art methods are based on the concept of "Maximum benefit for minimum cost" where cost can denote the development and integration costs of the optimization.

- Some optimizations are hard to implement
- Some optimizations require large compilation time
- Some optimizations have low payoff (the benefits) and it is often difficult to quantify payoff.

6 Local Optimizations

Local Optimizations are the simplest form of optimizations that can be performed by considering a single basic block in isolation. Some possible examples are:

- **Elimination of No-ops:** Some statements can be deleted. $x := x + 0$, $x := x * 1$, $x := x|0$
- **Algebraic Simplification:** Some statements can be simplified

$$x := x * 0 \rightarrow x := 0$$

$$y := x * * 2 \rightarrow y := x * x$$

$$x := x * 8 \rightarrow x := x << 3$$

$$x := x * 15 \rightarrow t := x << 4; x := t - x$$

The above code replacements are meaningful optimizations only if the RHS code would perform better than LHS code which also depends on the underlying hardware.

- **Constant Propagation or Constant Folding:** For statement $x := y \text{ op } z$ where y and z are constants, this statement can be computed at compile time.

$$x := 2 + 2 \rightarrow x := 4$$

$$\text{if } 2 < 0 \text{ jump } L \rightarrow \text{No-op}$$

$$\text{if } 2 > 0 \text{ jump } L \rightarrow \text{jump } L$$

More specifically, constant folding is about performing computations at compile time and constant propagation is about propagating constants throughout the program.

6.1 Dead Code Elimination

Dead code is the code that is unreachable from the initial block, for example, CFG node with no incoming edges. Removing unreachable code makes the program smaller (and sometimes faster due to fewer cache misses).

Why would unreachable block occur?

```
#define DEBUG 0
...
if (DEBUG) {
...
}
```

In the above example, after performing constant propagation, `DEBUG` would be replaced with `0`. Further constant folding would remove the jump statement to the 'if' block and it would be converted to dead code which can be eliminated.

Unreachable block may also occur when libraries are imported. Libraries might have a large number of functions but usually only a small fraction of functions are actually used. Then the code for other functions constitute dead code. Usually other optimizations may result in more dead code.

6.2 Common Subexpression Elimination

Consider the case of following code with the assumption that the value of x, y, and z remain unchanged after the first statement, then we can perform the following replacement to use the pre-computed value.

$x := y + z; \dots; \dots; w := y + z; \rightarrow x := y + z; \dots; \dots; w := x;$

In this case, SSA IR particularly helps in doing away with the initial assumption since SSA IR enforces the property that each variable can be assigned only once. In short, SSA IR makes more values available simultaneously.

$x_1 := y + z; x_2 := x_1 + 3; w := y + z \rightarrow x_1 := y + z; x_2 := x_1 + 3; w := x_1$

In the above example, the common subexpression being eliminated is $y + z$. The three address code also makes it easier to identify the uses of a subexpression in the code.

6.3 Copy Propagation

If we see $w := x$, replace subsequent uses of w with x (and eliminate this statement)

Before: $x := y + z; w := x + 3; v := x; u := v + 3;$

After: $x := y + z; w := x + 3; u := x + 3;$

This optimization leads to less number of copies leading to need of fewer registers. It also activates further optimizations like common subexpression elimination.

6.4 Example 1

Original Code: $x := y + z; w := x + 3; v := y + z; u := v + 3;$

After CSE: $x := y + z; w := x + 3; v := x; u := v + 3;$

After CP: $x := y + z; w := x + 3; u := x + 3;$

After CSE: $x := y + z; w := x + 3; u := w;$

Further copy propagation is possible due to the third statement $u := w$.

6.5 Example 2

$a := x ** 2$ $b := 3$

$c := x$

$d := c * c$

$e := b * 2$

$f := a + d$

$g := e + f$

Final form:

$a := x * x$ $f := a + a$ $g := 6 * f$

It is possible for the compiler to get stuck in a "local minima". In the above example, if f

$:= 2 * a$ was not replaced with $f := a + a$, there could have been possibility to use the shift operator ($f := a << 1$) or to use copy propagation followed by dead code elimination ($a := x * x$; $g := 12 * a$)

6.6 Summary

- Each local optimization does little by itself.
- Often optimizations interact: performing one optimization may enable or disable other optimizations.
- Optimizing compilers can be thought of as a big bag of tricks.

6.7 Typical Structure of an Optimizing Compiler

repeat {apply an optimization rule}
until {no improvement is possible}

- **Convergence** can be guaranteed by defining a metric for performance and ensuring that each iteration improves that metric. This monotonic behaviour would prevent any kind of oscillations that might be possible.
- **Optimality** is not guaranteed. The compiler can get stuck in a local minima.

7 Peephole Optimizations

A peephole optimization is a type of local optimization based on a pattern matching rule consisting of a pattern and a replacement. Here, both pattern and replacement are templates for a sequence of instructions.

$$pattern \rightarrow replacement$$

$$i_1, i_2, \dots, i_n \rightarrow j_1, j_2, \dots, j_m$$

The main idea here is to scan the code and look for the code matching the pattern template and replace it with the replacement code which might be better than the original code in utilization of one of the program's resources. Traditionally the peephole optimization has been quite successfully used on assembly code.

Some examples of peephole optimizations are as follows:

- `move $r1 $r2; move $r2, $r1` \rightarrow `move $r1 $r2`

Possible to do the above replacement because after the first instruction, second move is just a nop and can be removed. It is equivalent to dead code elimination. Do note that the registers mentioned in the instructions are simply placeholders and there can be some other registers as well.

- `addiu $r1, $r1, i; addiu $r1, $r1, i → addiu $r1, $r1, i + j`

Again registers and constants in the above instructions are simple placeholders and can take any arbitrary value. The above example can be considered as constant folding cast as peephole optimization.

- `addiu $r1, $r2, 0 → move $r1, $r2`
- `move $r1 $r1 → < empty >`

The peephole optimizations are implemented as a database of rules. An optimizer, then scans the code to find the code that matches any pattern in the database and then can be replaced by the replacement.

Many (but not all) local optimizations can be cast as peephole optimizations. Some examples and counterexamples are as follows:

- Algebraic simplification: If an instruction multiplies the value in a register with some power of 2, it can be replaced by a shift instruction.
- Copy Propagation: It cannot be cast as peephole optimization as the definition of a variable and its usage can be very far away with multiple instructions in between with various combinations making it difficult to match them to a pattern

$$\begin{array}{c} w := x \\ \dots \\ \dots \\ t := w + 1 \end{array}$$

Like local optimizations, peephole optimizations can be applied repeatedly. Applying one peephole optimization can activate as well as curb other usages of peephole optimizations. Thus, peephole optimizations are used iteratively a fixed number of times or until no further patterns can be found.

The idea of peephole optimizations becomes more attractive if these can be generated automatically instead of manually coded. Use of enumerative and stochastic methods have been there to automatically learn peephole optimizations.

”Program optimization” is grossly misnamed. Present day ”optimizers” have no intention or even pretense of working towards the optimal program. They work towards improving the performance of the code based on certain patterns. Therefore, code ”improvers” is a better term.

8 Global Optimizations

(Prepared by Aditya Senthilnathan)

Global optimisations go beyond the scope of a single basic block. It needs reasoning of the whole control flow graph which may be within the body of a method.

8.1 Global Constant Propagation

- This is similar to local constant propagation but now we are reasoning in a global level (across several basic blocks within the body of a method)
- In this optimisation, we basically ask the question "When can we replace the use of a variable x with a constant k

Ans: If on **every path** to the use of x , if the **last assignment** to x is $x:=k$, then we can do this transformation/optimisation.

The algorithm for global constant propagation requires a class of algorithms which are called "Global Dataflow Analysis". This kind of reasoning (with paths, etc.) is required for many different types of optimisation problems in compilers. And so, it helps to create a common framework in which we can implement the same logic but with different kinds of properties so that there is reuse of the same idea.

Let's identify some traits that global optimisations tasks share:

- Optimisation depends on knowing property X at a certain program point P
Eg. In global constant propagation, we were interested in knowing if $x:=k$ is the last assignment on all possible paths, at the program point of the last basic block (where x is used) in our example graph.
- Proving X at P requires knowledge of entire program
Eg. In our global constant propagation example, to be able to say $x:=k$ is the last assignment to x on all possible paths that can reach program point P , we need to know the characteristics of the entire program. As opposed to local optimisation where you only need info about the concerned basic block. Here, even though transformation is only made in a basic block, it relies on a property that requires knowledge of the entire program.
- It's OK to be conservative

We can either say,

X is definitely true

or say,

We don't know if X is true

We want to get "X is definitely true" as much as possible because only then can we apply the transformation but if we get the conservative answer "We don't know if X is true", then we don't apply the transformation to preserve program correctness. Knowing if X is definitely not true is useless because in this case we still don't apply the transformation and so this use case is clubbed with "We don't know if X is true".

9 DFA Values

(Prepared by Aditya Senthilnathan)

Recall that for global constant propagation, we need the following:

Optimisation: Replace a use of x by a constant k

Constraint: On every path to the use of x , the last assignment to x should be $x:=k$

We need to do this for every variable in the program. But first let's do this for a single variable x and then we can generalise it to more variables.

9.1 Dataflow Analysis Values

These are abstract values that are supposed to capture the set or category of concrete values that x (a variable) may take.

At each program point, we associate one of the following values with X (a variable),

Value	Explanation
\top (top)	This value represents either that <ul style="list-style-type: none">• This statement never executes• or we have not executed or seen it yet
\perp (bottom)	This value represents either that <ul style="list-style-type: none">• We can't say that X is a constant• X is definitely not a constant
C (constant)	This value represents that X is a constant C

Question: What are we trying to do here? What do these values mean?

Answer: We are trying to figure out what are the possible values of X (a variable) at every program point. The values we are talking about essentially tell us if the program reached here what is the value of X or all the possible values of X at this program point.

Now we will discuss each of the values in a little more detail,

- \top (top) - This value represents that this statement never executes i.e the program/control never reaches here. This might be because this part of the program is never reachable from the beginning of the program. So in this case we just say that the value is \top here because the statement never executes and therefore it is not meaningful to say what this value is.
- C (constant) - It could be any constant. Eg. 0,1,2,-2,etc. This value says that whenever this program point is reached, irrespective of the path taken, we are sure that X will be equal to constant value C at this point.

- \perp (bottom) - This value essentially says that we don't know if X is a constant or not at this point. We are not sure of its value. This might be happening because it's possible that on different paths X could take different values or on some path X is not a constant, etc. Even if we know for sure that X is not a constant, we still assign \perp to X. This is a conservative assignment. Technically, even if we (DFA algorithm) assign \perp to X at every program point conservatively, then this solution is also correct but it is trivially true and is not of much use to us.

Note: If we're still in the middle of execution of the DFA algorithm and at a particular point, we have value \top for one of the variables in the program then it could also mean that the DFA algorithm hasn't seen this point yet or hasn't reached this point yet. The above discussion and interpretation of \top only applies if a variable has \top value after the DFA algorithm has finished executing

9.2 Using DFA values for transformation

Given global constant information (\top , \perp , C)

- Inspect $x = ?$ (either \top or \perp or C) at the program point that just precedes the statement which you want to examine (the statement that uses x)
- If $x = C$, then replace the use of x with C

For some examples of how to assign DFA values see the lecture video

10 Dataflow Analysis Algorithm (DFA)

(Prepared by Aditya Senthilnathan)

In this section, we will now try to answer the question "How do we identify what the DFA value should be at every program point?"

We will only analyse one particular variable x for now. This can easily be generalised for arbitrary number of variables and this is left to the reader.

10.1 Insight

DFA of a complex and large program can be expressed as a combination of simple rules relating the change in information between adjacent statements.

We're just going to see

- What the value of x is just before this statement
- What is the current statement
- Based on the current statement, we then use rules (to be discussed) to compute what should be the value after the statement

One way to think about this is that we are "pushing" or "transferring" information from one statement to the next. It's almost just like program execution with the caveat that we are not dealing with concrete values anymore but rather abstract DFA values.

10.2 Function for "Constant Information"

We now define a function $C(x, s, in/out)$ where,

- x - variable for which we want to compute abstract values
- s - statement s in the program
- in/out - whether it is input to the statement or output to the statement i.e just before the statement or just after the statement.
 - $C(x, s, in) = \text{Value of } x \text{ just before } s$
 - $C(x, s, out) = \text{Value of } x \text{ just after } s \text{ i.e just after statement } s \text{ is executed}$

The idea is to calculate the value of this function $C(x, s, in/out)$ for every x and every s (both in and out)

10.3 Transfer Function

We need to define how information is transferred from one statement to another i.e how are we pushing information. For this we need to define a transfer function.

Transfer Function: transfers information from one statement to another

We are going to define rules for the following two cases:

- **Case 1:** There are one or more predecessor points with edges to the current program point. In this case, we will see how to combine information/values flowing in from the predecessor points to assign a value for the $C()$ function in the current program point
- **Case 2:** For a given statement s , we see how the information/value is modified as it flows through the statement

10.3.1 Transfer function for Global Constant Propagation

We will define the function by doing an exhaustive case analysis. We can do this kind of finite description of the function with an exhaustive case analysis because there are only a limited number of values and cases to consider.

$C(x, s, p)$ can take only one of 3 different unique values during and after execution of the DFA algorithm i.e \top, \perp, C . Also once the algorithm picks a constant value for x at a particular program point, the constant value is fixed and it cannot vary after that. Eg. If the DFA algorithm picks 3, it cannot assign 4 later on. It can only change to either \top or \perp later on in the execution.

Rules for the Transfer function:

1. If $C(p_i, x, out) = \perp$ for any i , then $C(x, s, in) = \perp$
2. If $C(p_i, x, out) = c$ and $C(p_j, x, out) = d$ and $c \neq d, i \neq j$ for some i, j then, $C(s, x, in) = \perp$
3. If $C(p_i, x, out) = c$ or $C(p_i, x, out) = \top$ for all i , then $C(s, x, in) = c$

4. If $C(p, x_i, out) = \top$ for all i , then $C(s, x, in) = \top$
5. If $C(s, x, in) = \top$, then $C(s, x, out) = \top$
6. If rule 5 does not apply i.e $C(s, x, in) \neq \top$, then $C(x := c, x, out) = c$ [where c is a constant]
7. If rule 5 does not apply i.e $C(s, x, in) \neq \top$, then $C(x := f(...), x, out) = \perp$
8. $C(y := ..., x, out) = C(y := ..., x, in)$ where $y \neq x$

We now have 8 rules relating $C(s, x, in/out)$ to the value of this function at a predecessor point. These rules are exhaustive.

But so far we have just stated the rules. How do we get the actual values?

Think of this as a system of equations. We are interested in solving and identifying values for this C function such that all 10 rules are satisfied.

How do we solve this system of equations?

Note that there is a trivial solution,

$$C(s, x, in/out) = \perp, \text{ for all statements } s$$

All 8 rules are satisfied by this solution but this solution is useless to us because it doesn't allow us to perform any transformations/optimizations. We're not interested in just any solution. We are interested in the "most precise" solution (for some notion of precision). For eg. Saying that $x = 4$ at some program point is more precise than just saying $x = \perp$. Similarly, saying that $x = \top$ is more precise than saying $x = \perp$ because the fact that this program point is unreachable is still useful information because now we can consider doing dead code elimination in this part of the program.

To solve this system of equations in a more precise way, we use a Fixed Point Iteration Algorithm.

Algorithm:

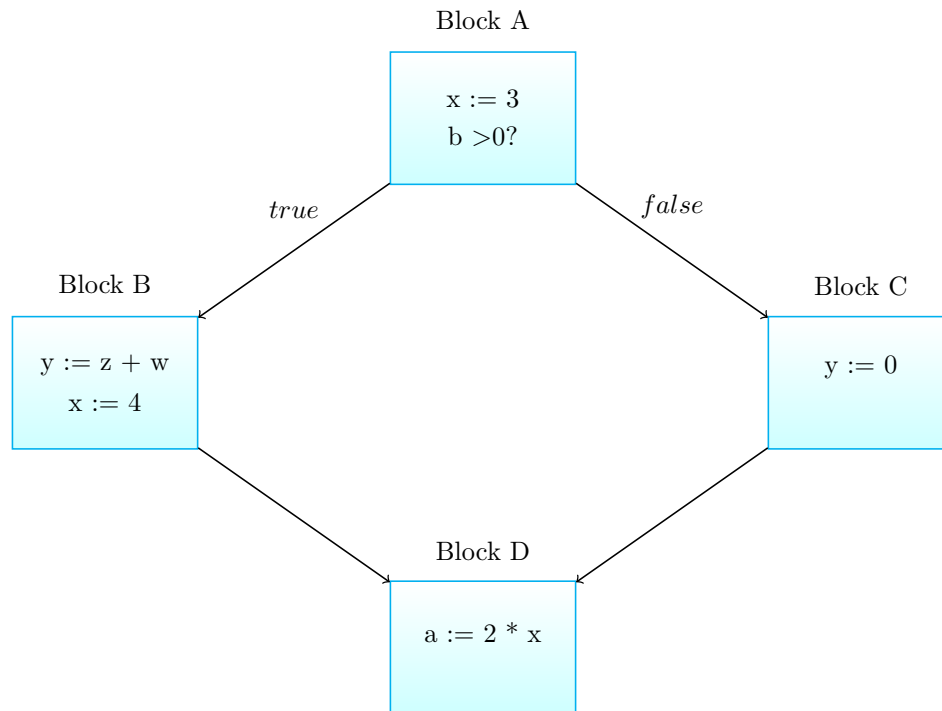
- At every entry point s to the program, set $C(s, x, in) = \perp$
- At every other statement s (which is non-entry), set $C(s, x, in) = \top$
- For every statement s , $C(s, x, out) = \top$
- Repeat the following until all program points satisfy rules 1-8:
 - Pick a statement s not satisfying one or more rules in rules 1-8 and update the corresponding $C()$ function value using the appropriate rule

There is a guarantee that this algorithm will converge. It will also give us the most precise answer. In the worst case, the solution it gives will be \perp at all program points.

11 DFA Example

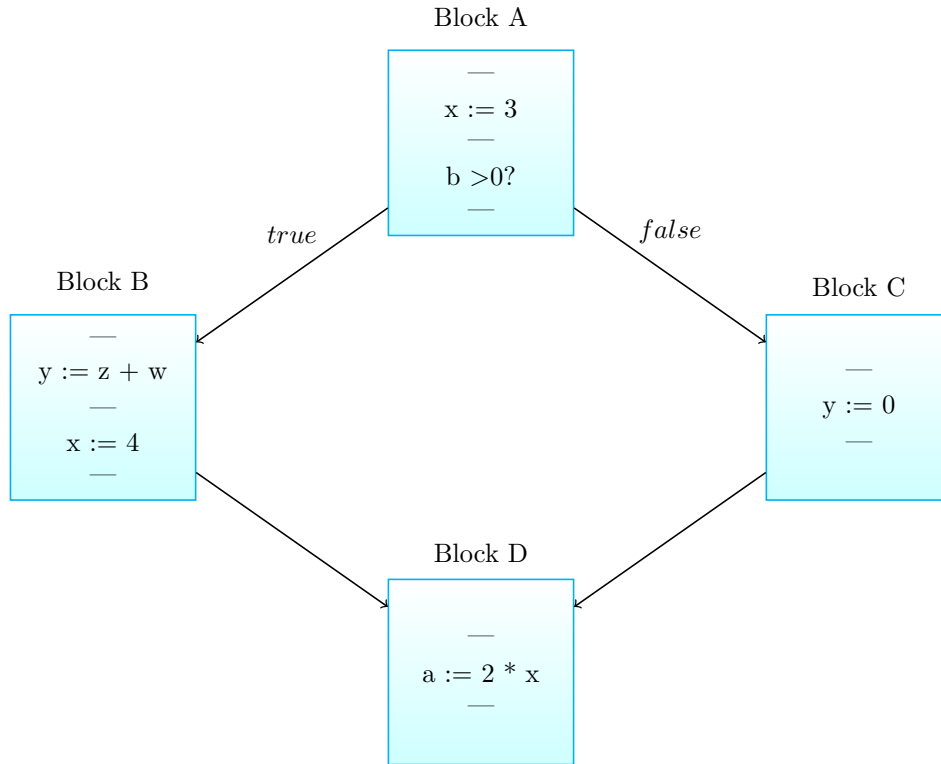
We will apply the fixed point algorithm to the variable “x” in the following piece of code.

Figure 10: Example



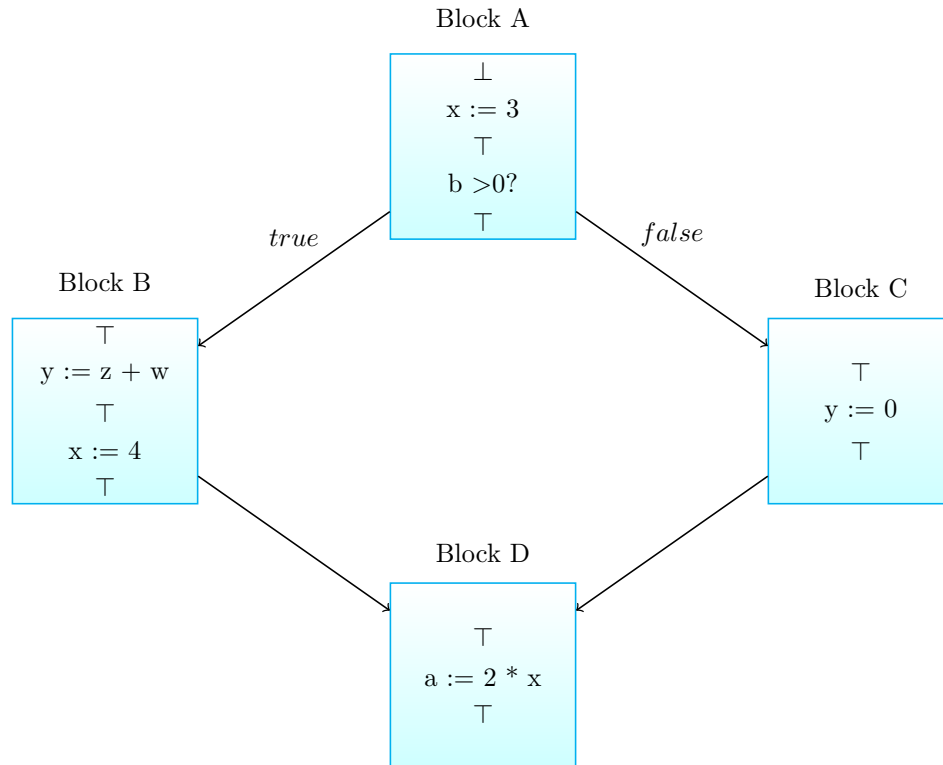
Next figure shows all the program points in our example. Program points are all the points that appear before and after every statement. In case of only one predecessor, the “out of the predecessor” is merged with the “in of the successor” while working on this piece of code. “—” marks the program points in the example we are working on.

Figure 11: Program Points



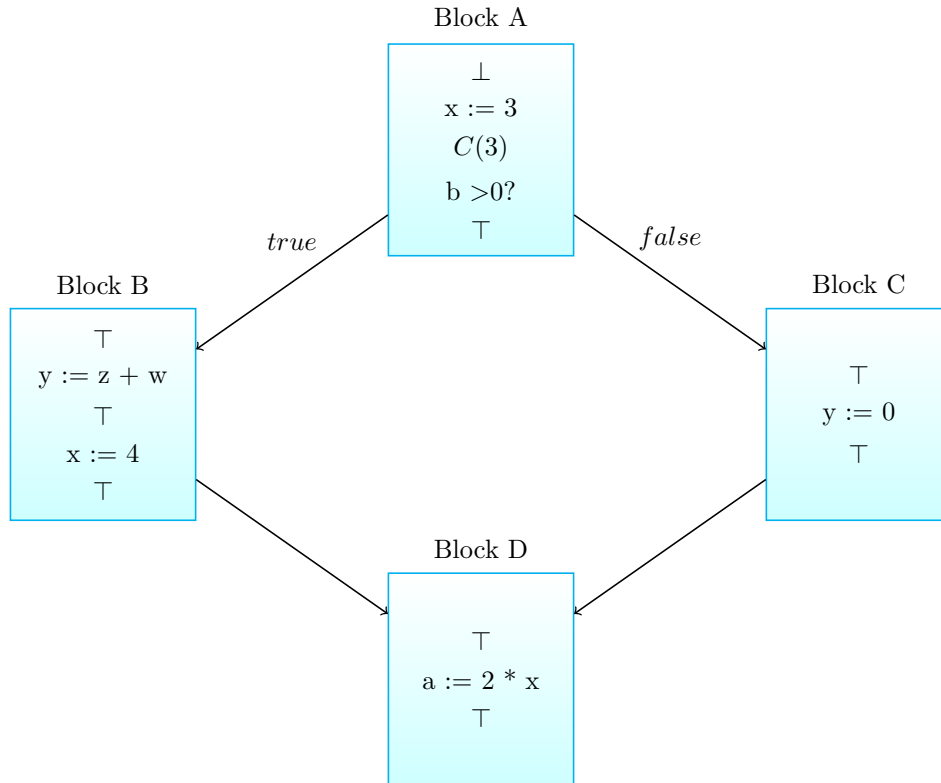
Algorithm starts by initializing every value to \top (top) except the “in of entry statement” which is initialized to \perp (bottom).

Figure 12: Initialization



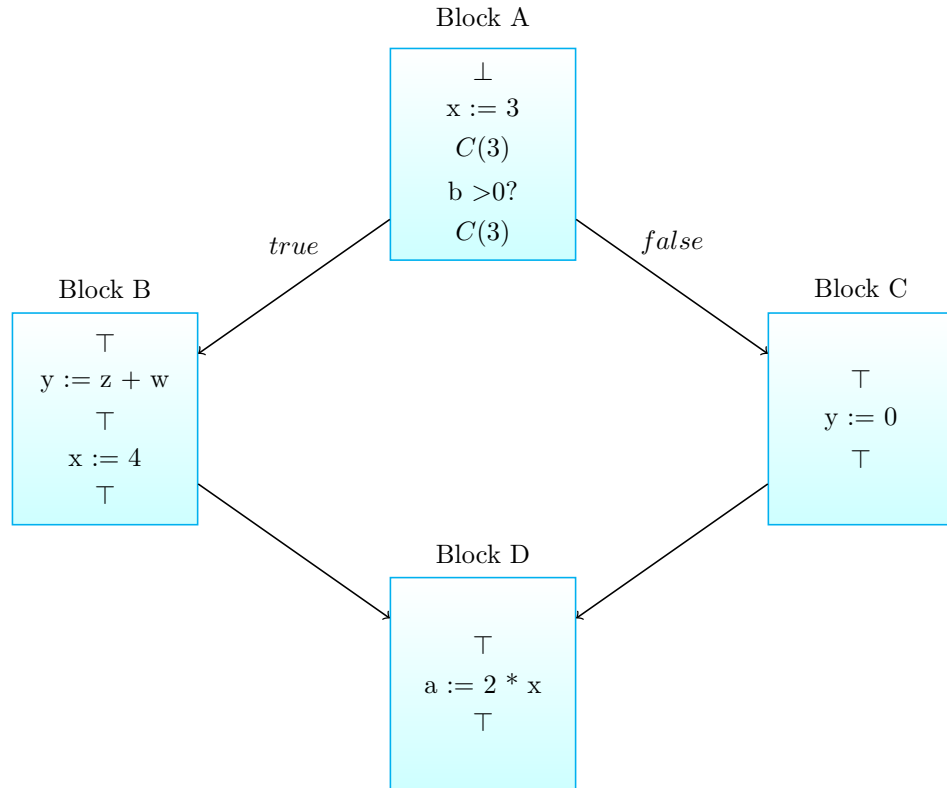
Now the algorithm will pick the statement which does not follow the transfer function rules, which is the “ $x := 3$ ” statement in this example, and update it. The “out of the statement” should be Constant 3 according to the transfer function rule. After updating the “out of the statement”, we get this graph.

Figure 13: “ $x := 3$ ” statement update



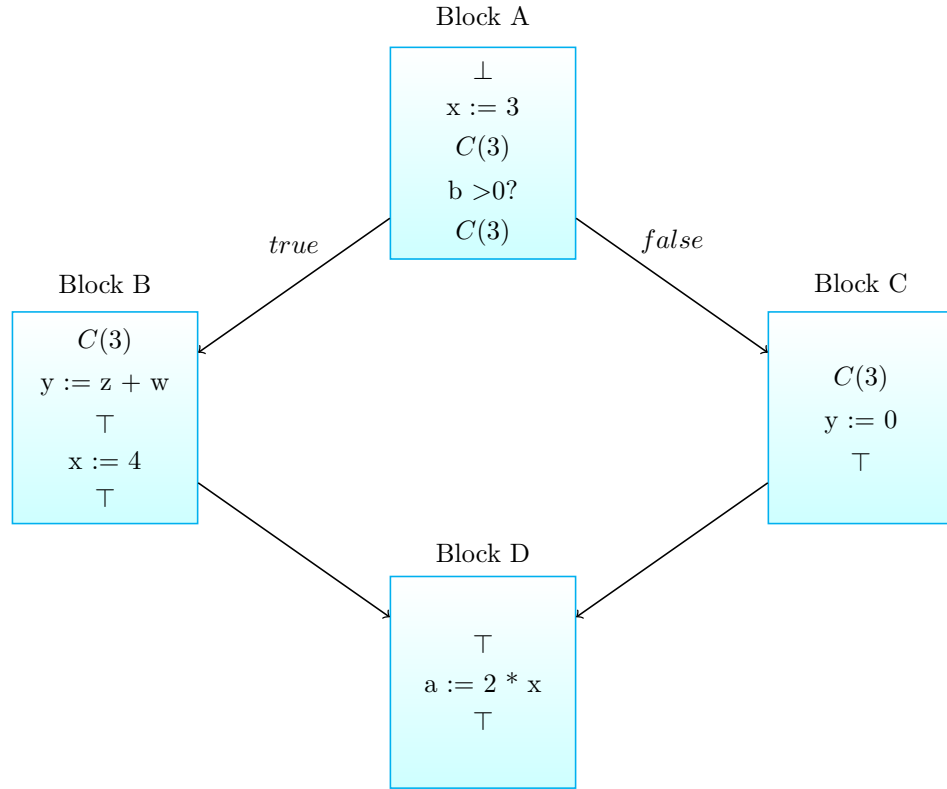
The algorithm repeats this process until all program points satisfy the transfer function rules.

Figure 14: “b > 0?” statement update



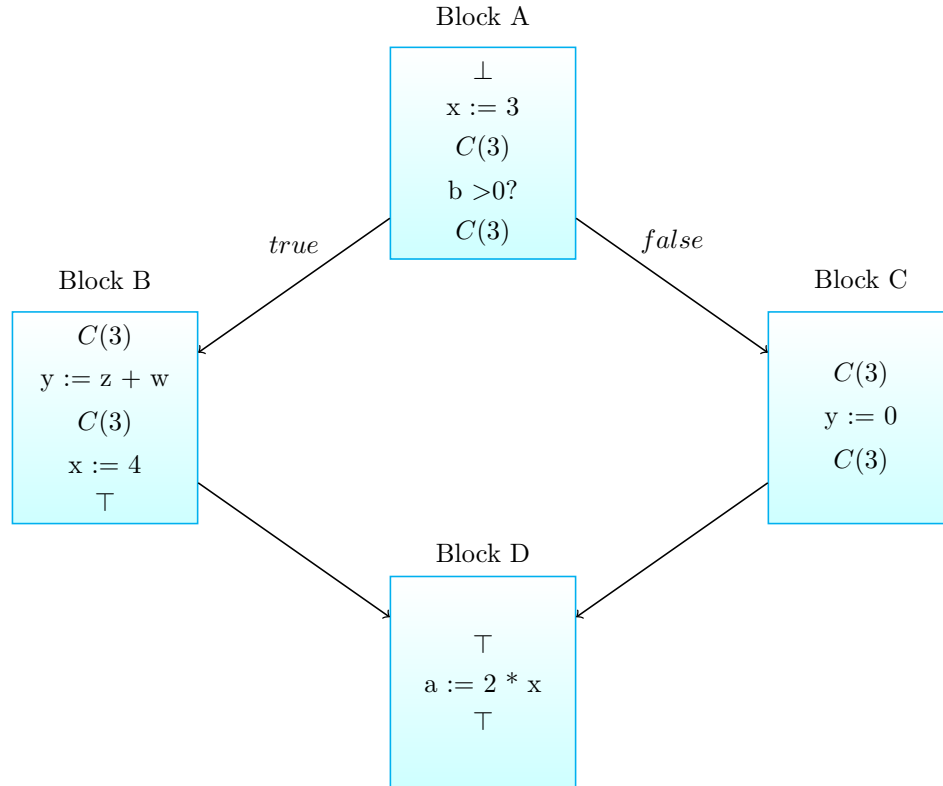
At this stage, there are two program points which do not satisfy the transfer function rules. The entry point of Block B and Block C do not match with their predecessors. We can update them.

Figure 15: “Successors of Block B and Block C” update



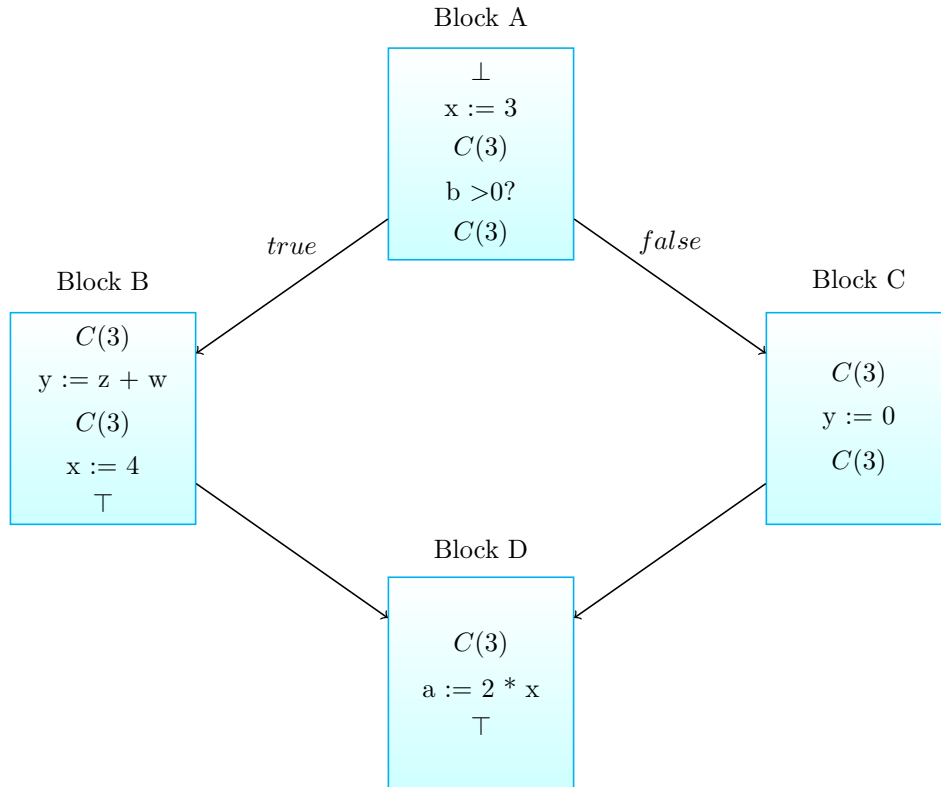
Again there are two different statements that do not satisfy the transfer function rules, we can update them.

Figure 16: “ $y := z + w$ ” and “ $y := 0$ ” statements update



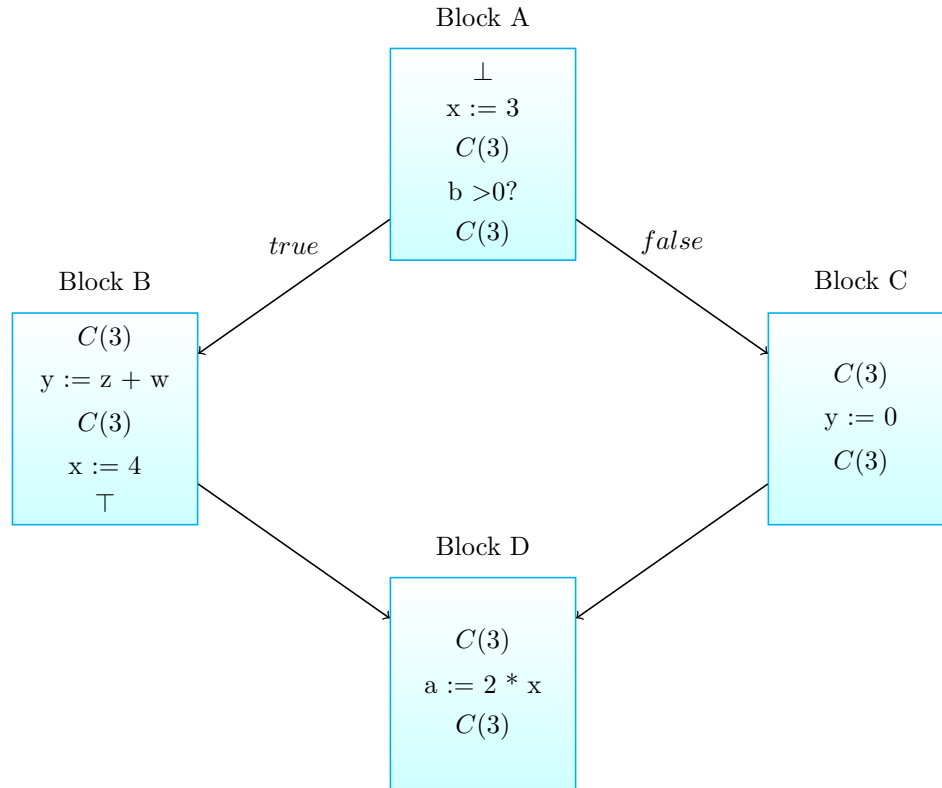
At this stage, we can either update “ $x := 4$ ” statement, or we can update the successor of Block D. Irrespective of which one we pick, algorithm will return the same answer. In this example, we will update the successor of Block D. One predecessor is a constant, and other predecessor of Block D is \top . Using the 3rd transfer function rule, we can update the value of successor to the same constant.

Figure 17: “Successor of Block D” update



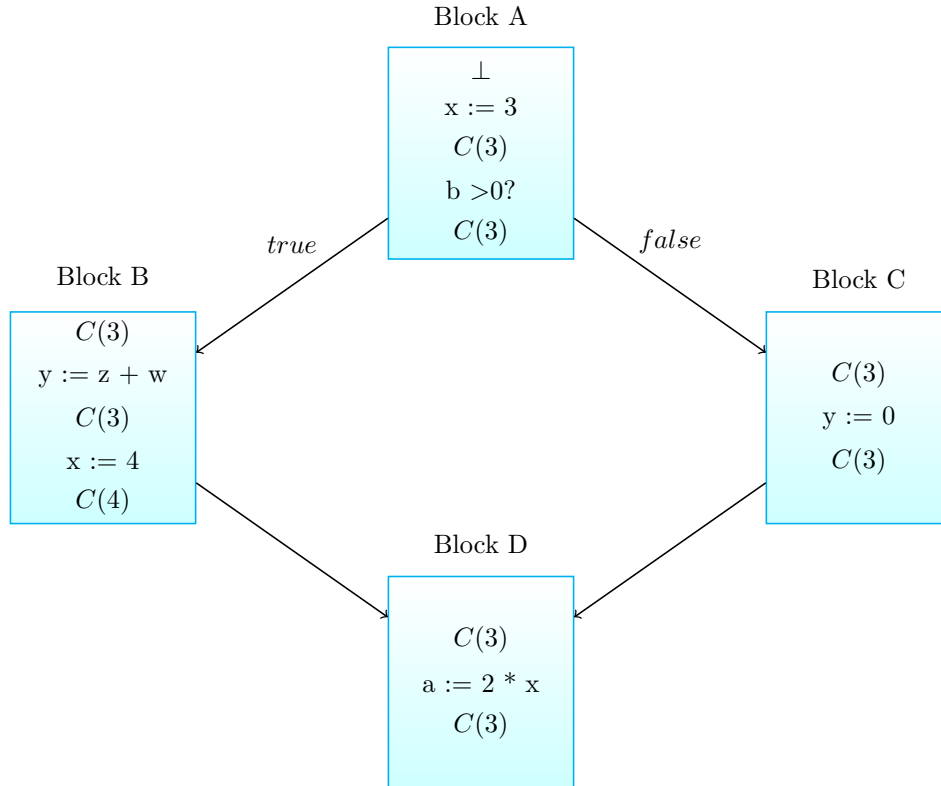
Updating “a := 2 * x” statement in Block D.

Figure 18: “a := 2 * x” update



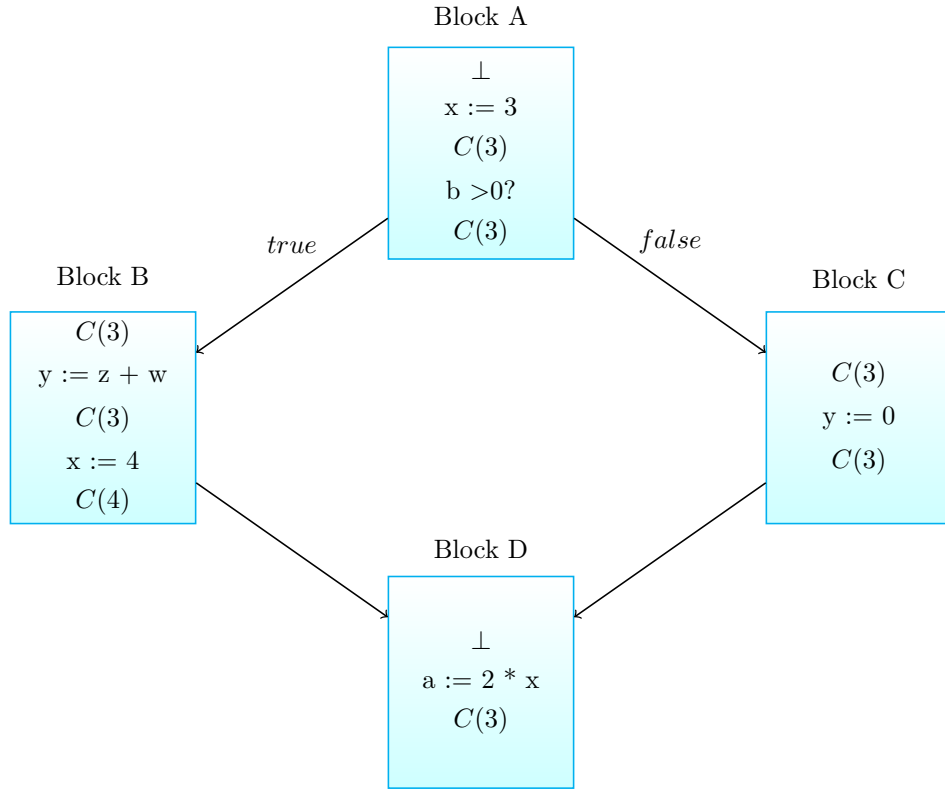
There is only one statement, “x:=4” in Block B, that does not satisfy the transfer function rules. After updating it, we get this graph.

Figure 19: “x := 4” statement update



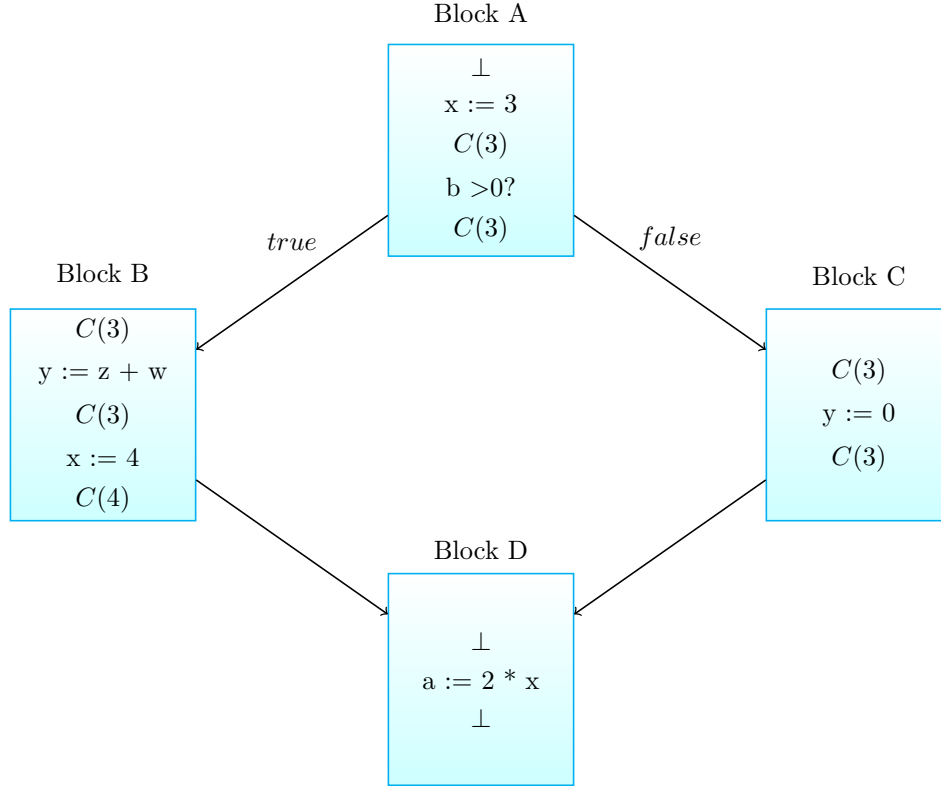
Because of the previous update, the successor of the Block D does not satisfy the transfer function rules. According to the 2nd transfer function rule, it should have the value \perp .

Figure 20: “Successor of Block D” update



Updating the “ $a = 2 * x$ ” statement in Block D, we get this graph.

Figure 21: “ $a := 2 * x$ ” statement update



At this stage, there is no program point that violates any of the 8 transfer function rules, so the algorithm terminates by returning this **Fixed point solution** to us.

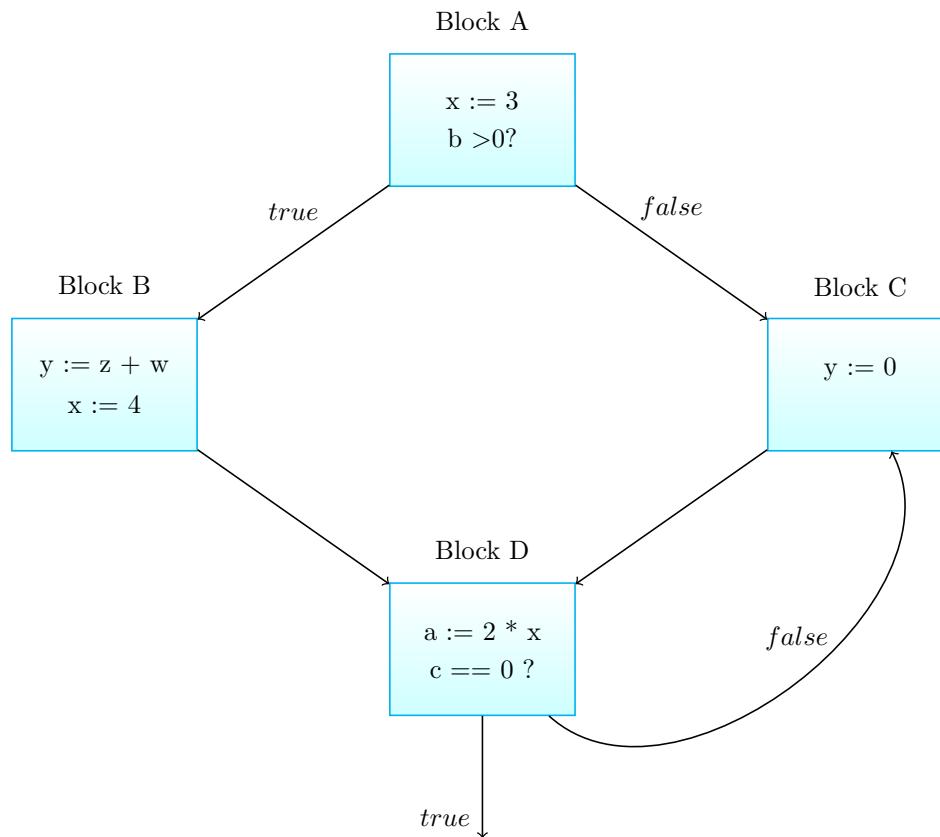
11.1 Some Guarantees

- The fixed point solution will satisfy all the transfer function rules at each program point.
- The algorithm is guaranteed to converge, irrespective of whether the program has loops or not.
- Whenever there are multiple options to choose for a statement S to be updated, irrespective of which one we choose, we always get the same answer.
- The fixed point solution is going to have some precision guarantee, by some definition of precision, for the system of equations we are trying to solve.

12 DFA Loop Example

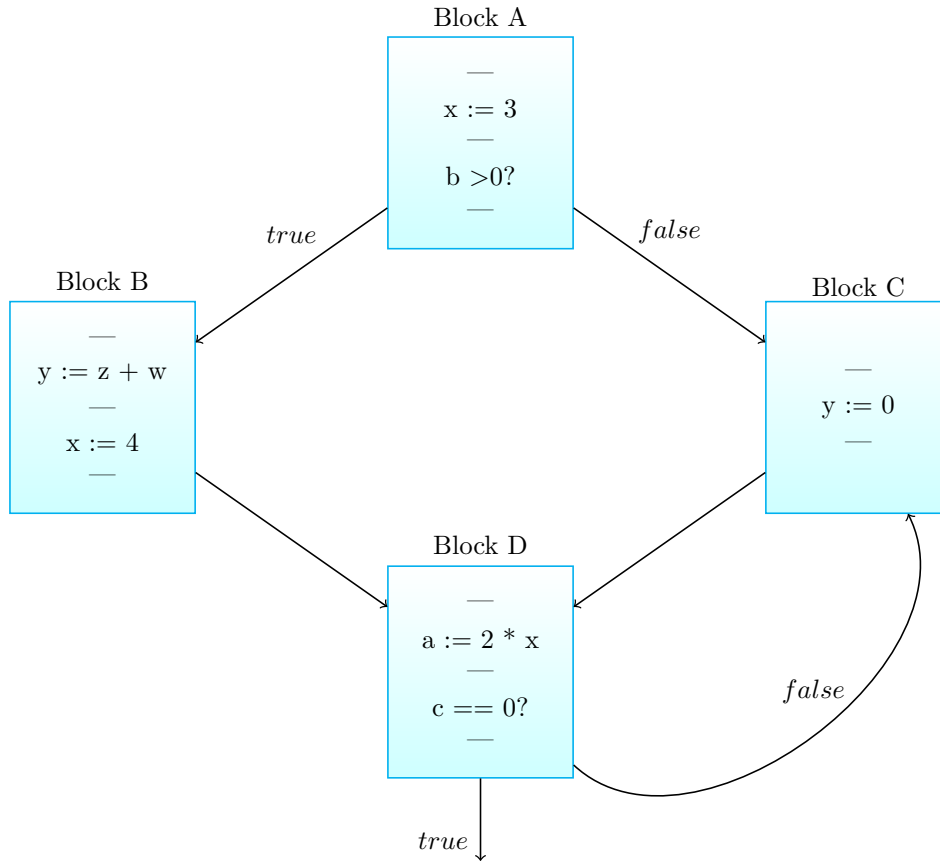
In this section, we will look at an example of DFA for a program with loops. We will apply the DFA algorithm on the variable “x” in the following piece of code.

Figure 22: Example



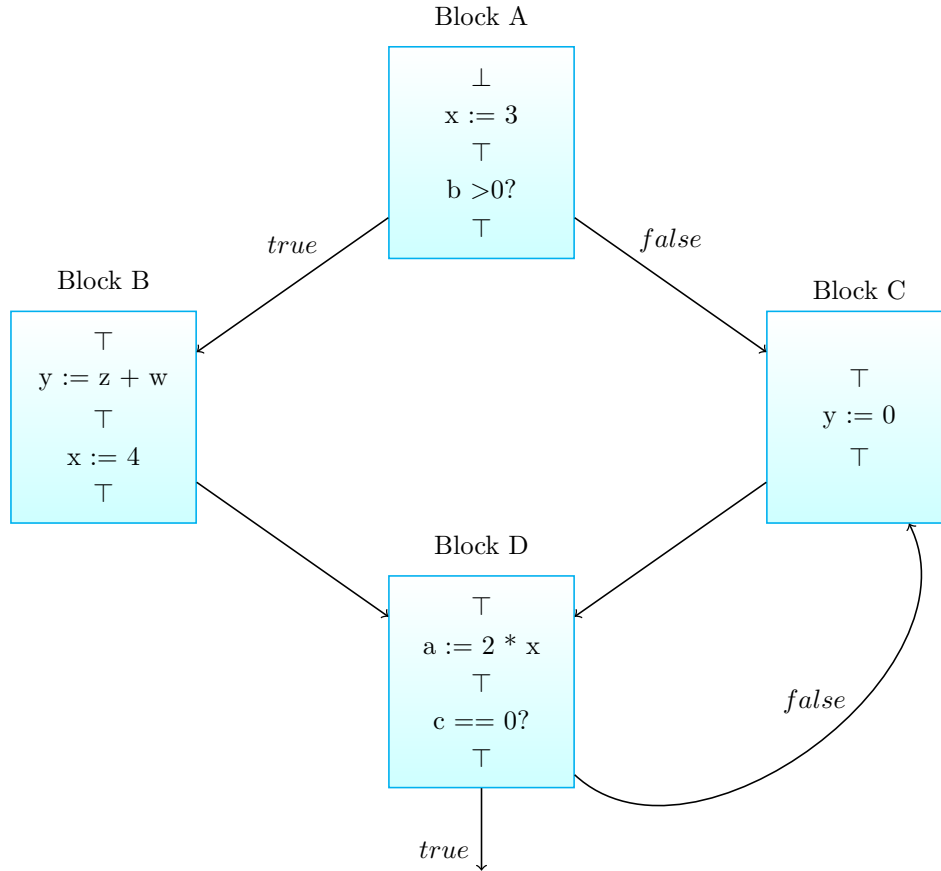
Next figure shows all the program points in our example. Program points are all the points that appear before and after every statement. In case of only one predecessor, the “out of the predecessor” is merged with the “in of the successor” while working on this piece of code. “—” marks the program points in the example we are working on.

Figure 23: Program Points



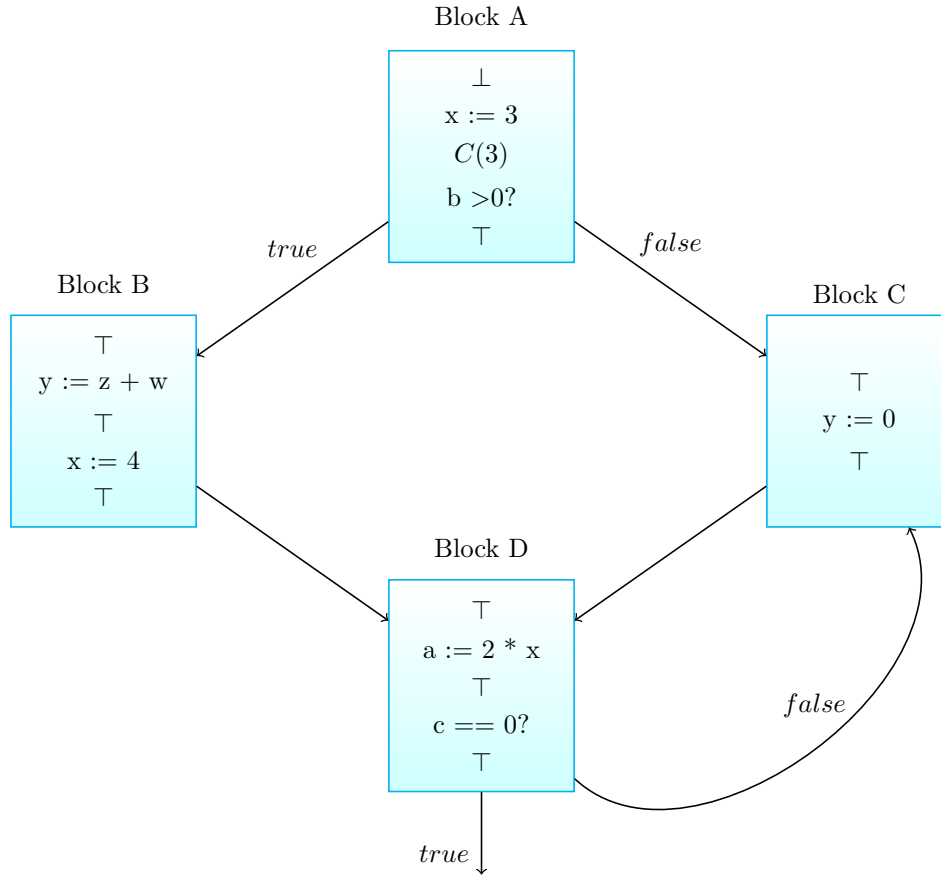
Next step is to initialize all the program points. To do that, we initialize all the program points to value \top except the entry points of the program which are initialized to \perp

Figure 24: Initialization



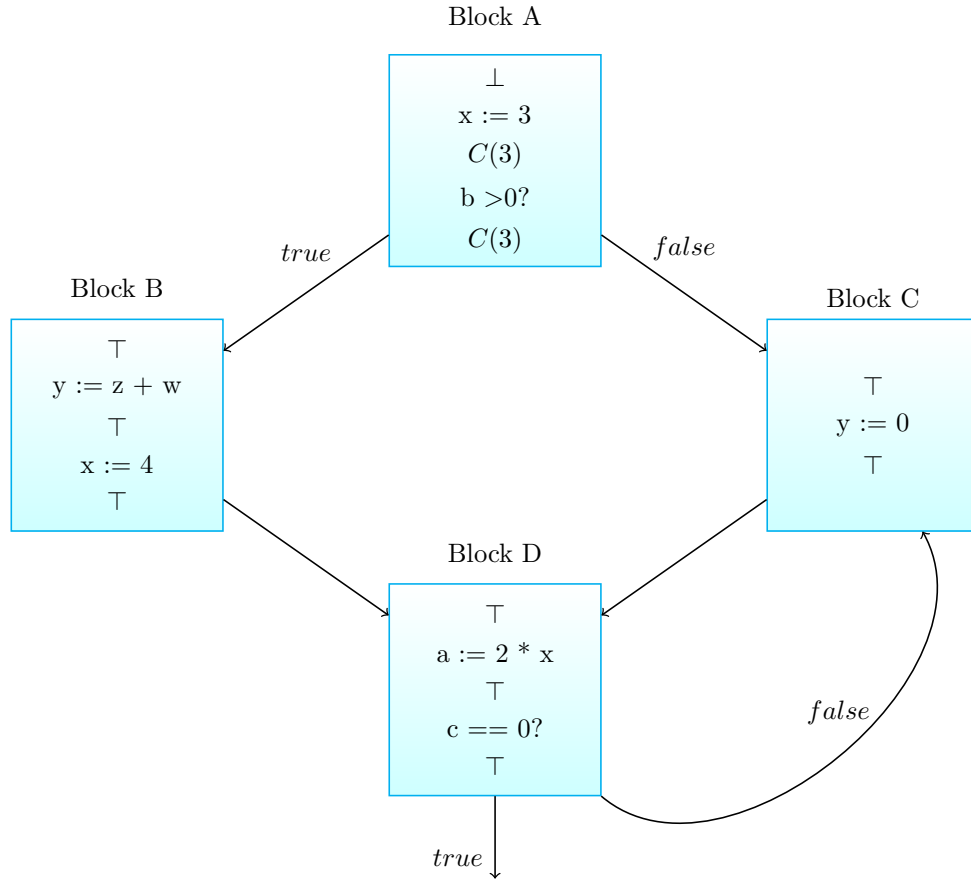
In the next step, we identify the statements which do not follow the transfer function rules and start updating them. At this stage, only the statement “ $x := 3$ ” does not follow the transfer function rule, so we update it.

Figure 25: “ $x := 3$ ” statement update



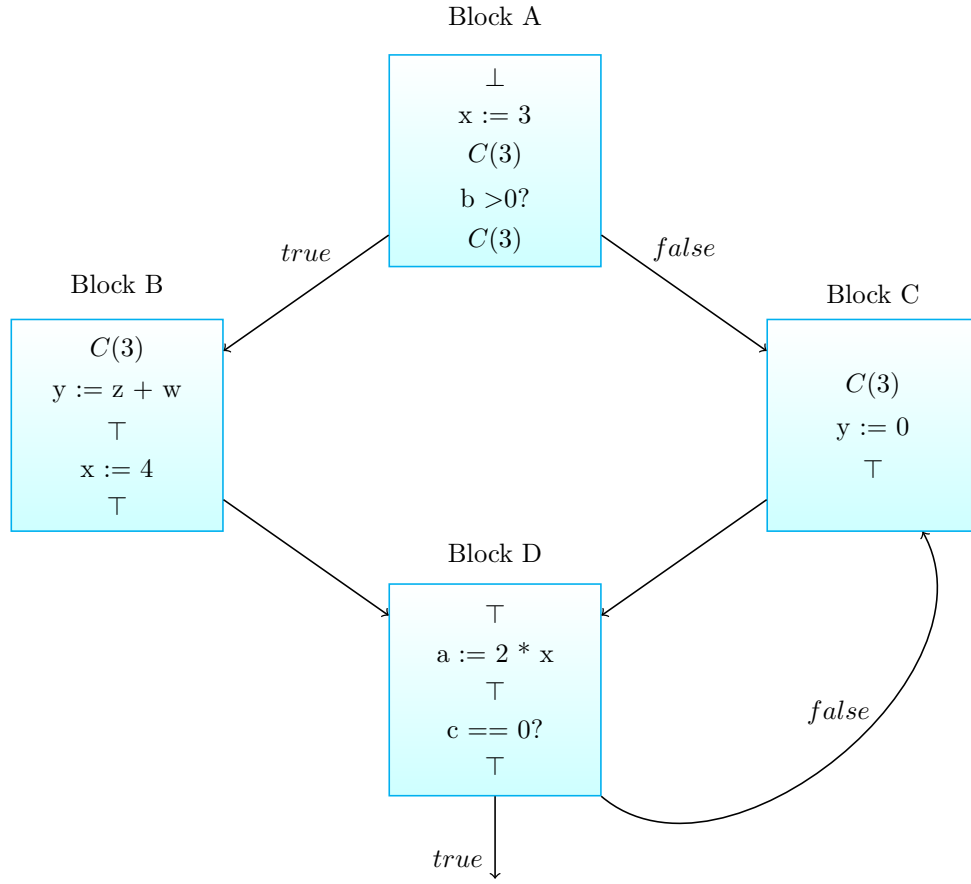
The process of identifying the statement that violates the transfer function rules and updating it, is repeated until every program point satisfies them. At this stage, the “b > 0?” statement violates the transfer function rules, so we update it.

Figure 26: “b > 0?” statement update



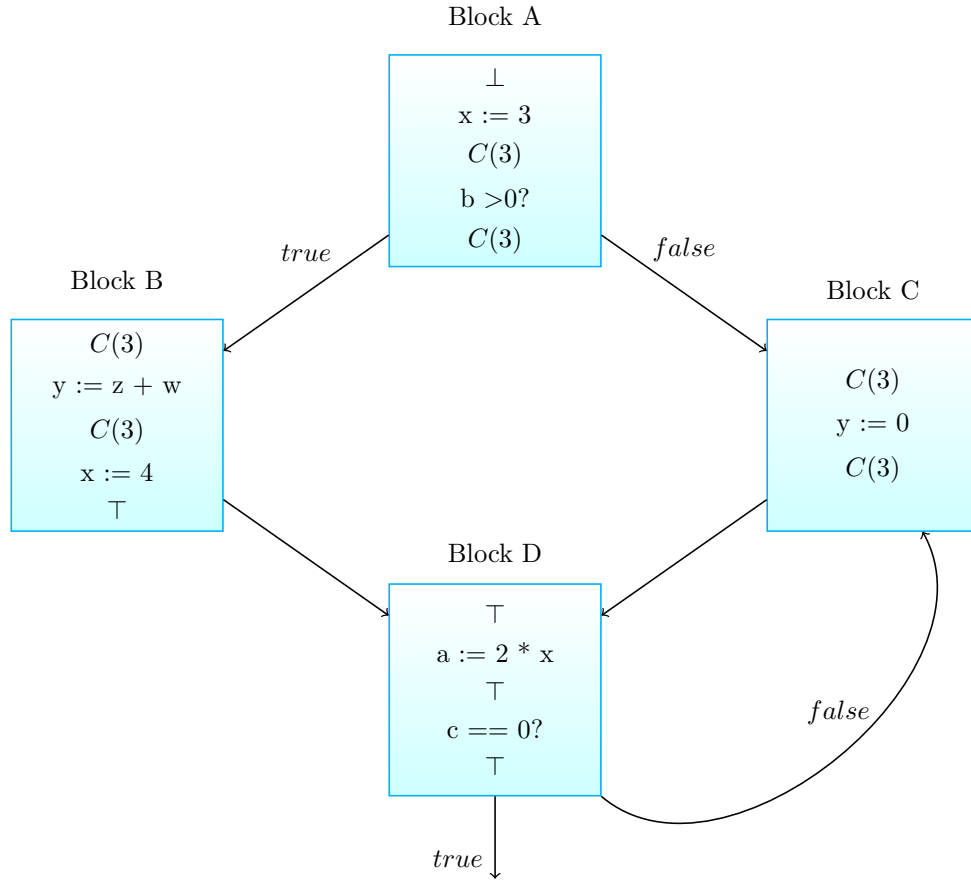
At this stage, the successors of Block B and Block C do not satisfy the transfer function rules, so we can update them.

Figure 27: “Successors of Block B and Block C” update



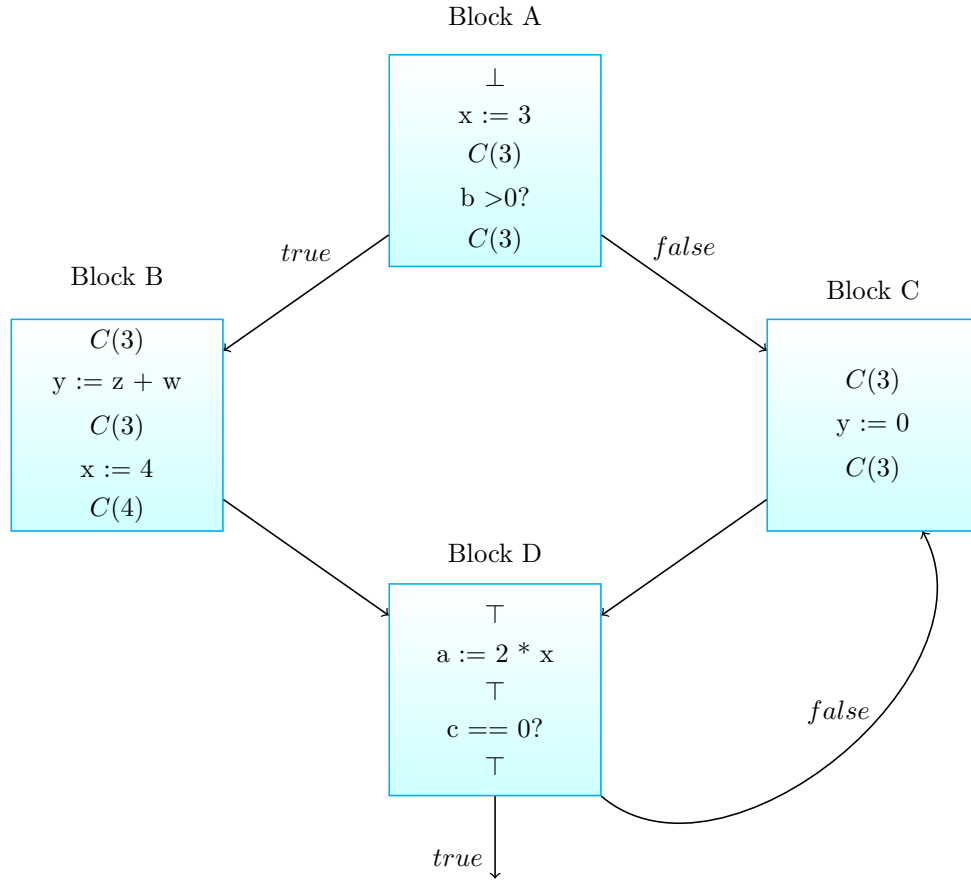
There are again two different statements “ $y := z + w$ ” and “ $y:=0$ ” that violate the transfer function rules, and we will update them.

Figure 28: “ $y := z + w$ ” and “ $y := 0$ ” statements update



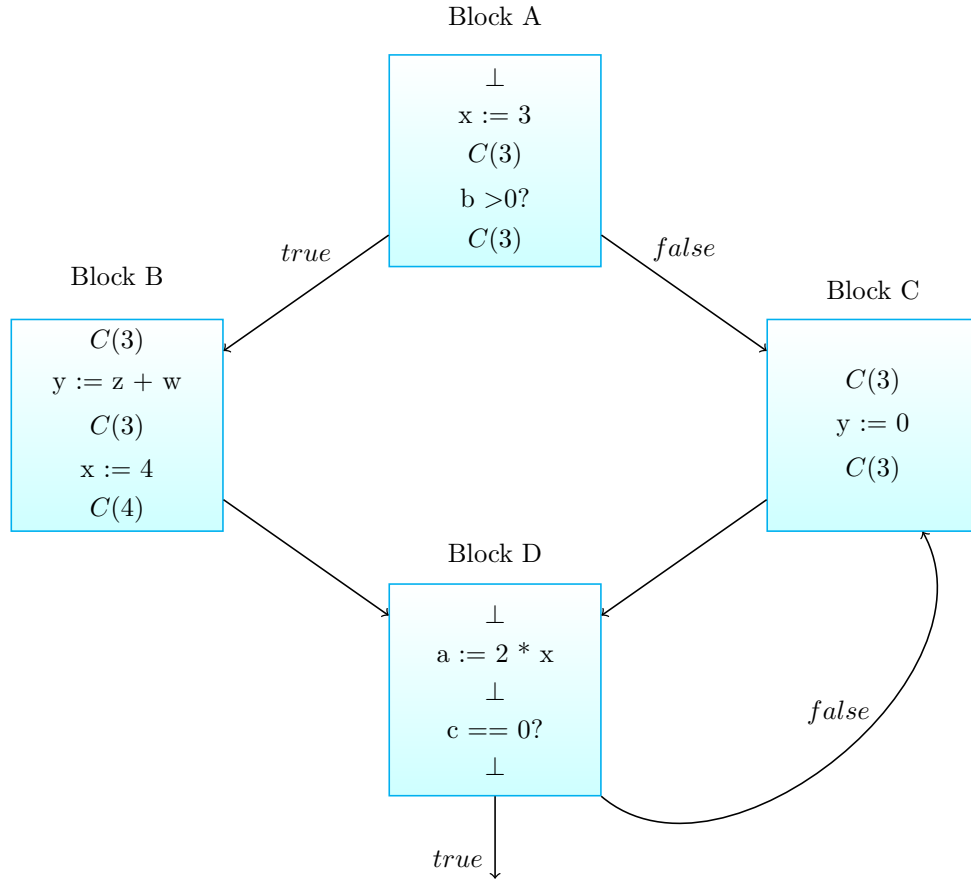
We have again two different statements that we can update at this stage, and we can choose to update anyone that we like, since the algorithm is guaranteed to return the same solution. In this example, we will update the “ $x := 4$ ” statement.

Figure 29: “ $x := 4$ ” statement update



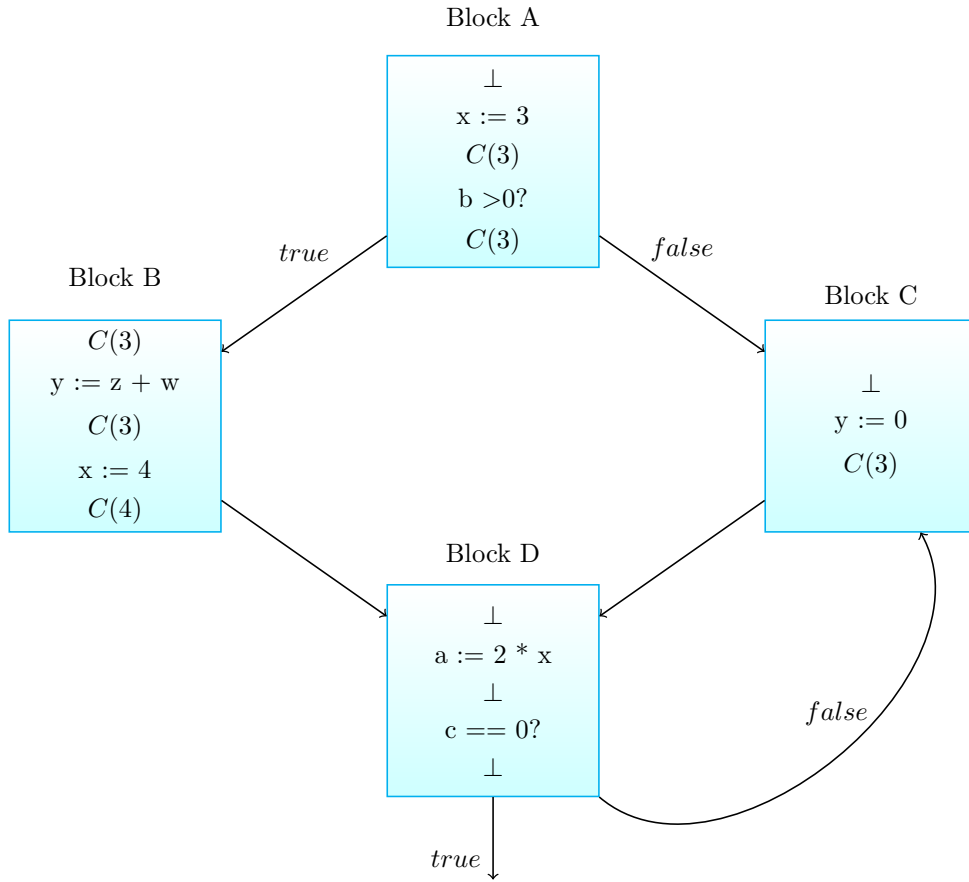
Now we will do the same thing for Block D, update the successor for Block D and then update its statements one by one. At the end, we will reach this stage.

Figure 30: “Block D” update



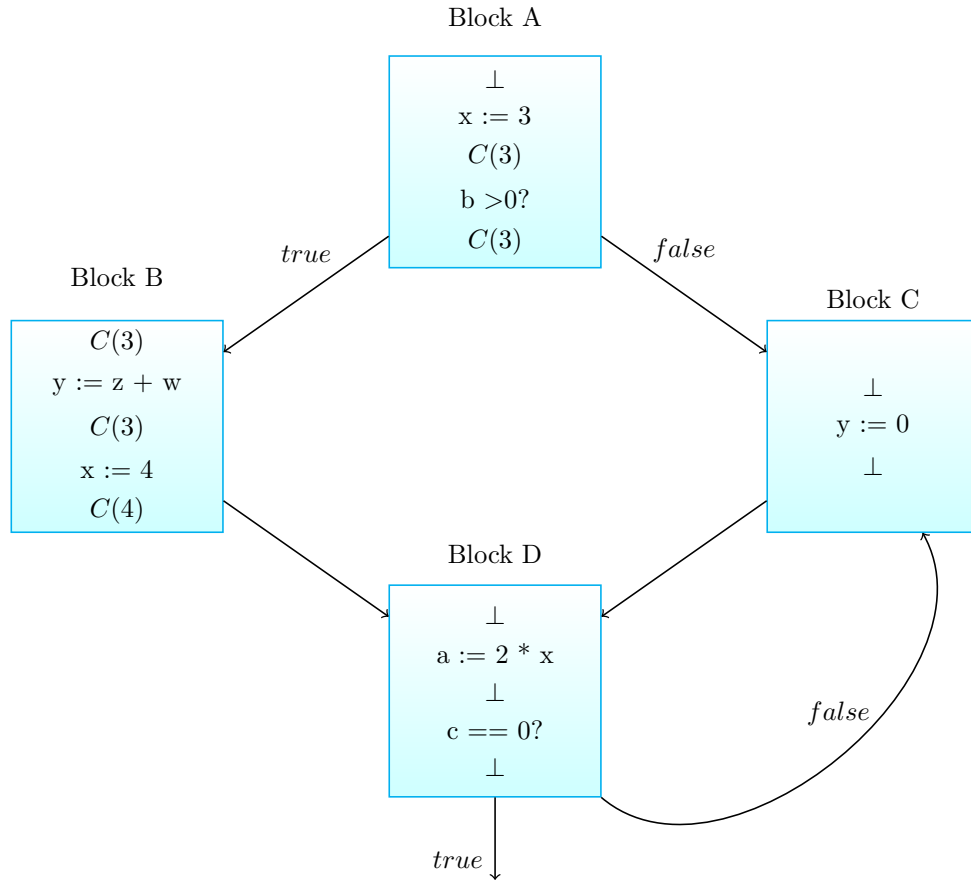
At this stage, we haven't reached the fixed point solution. The successor of Block C violates the transfer function rules. Earlier, it was acceptable for the algorithm to update the value to $C(3)$ because the predecessor entering from Block D was \top , which is now updated to \perp . So, we will have to update this program point.

Figure 31: "Successor of Block C" update



Updating the “ $y := 0$ ” statement, we reach this stage.

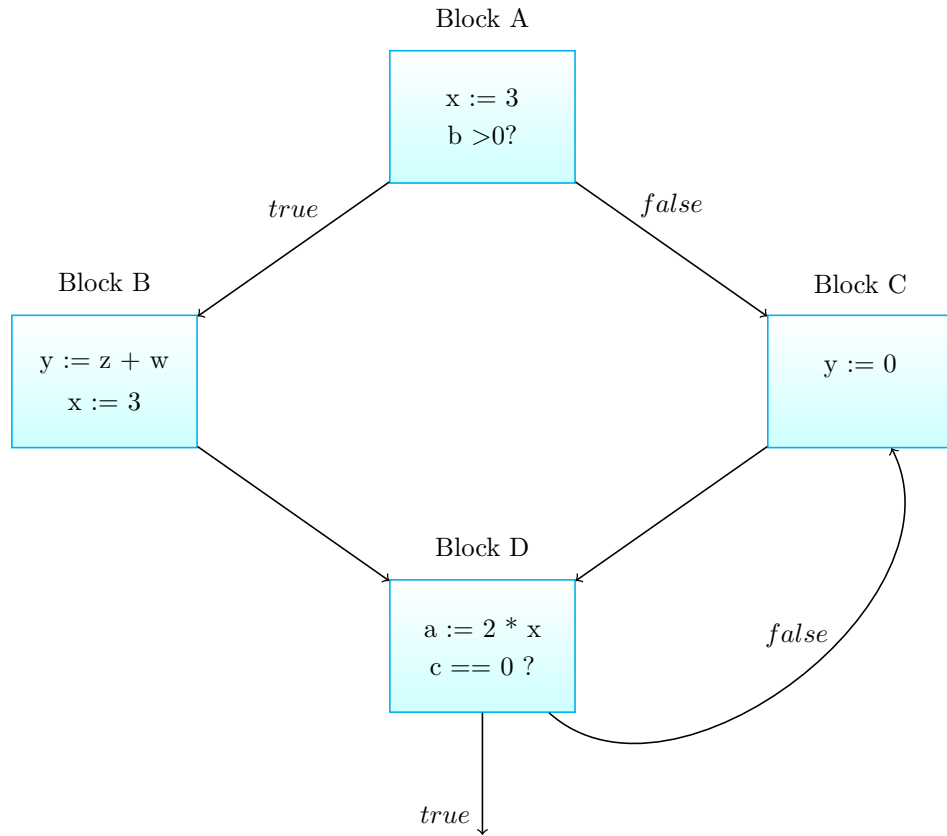
Figure 32: “ $y := 0$ ” statement update



All the program points at this stage satisfy the transfer function rules, so this is a **Fixed Point Solution**.

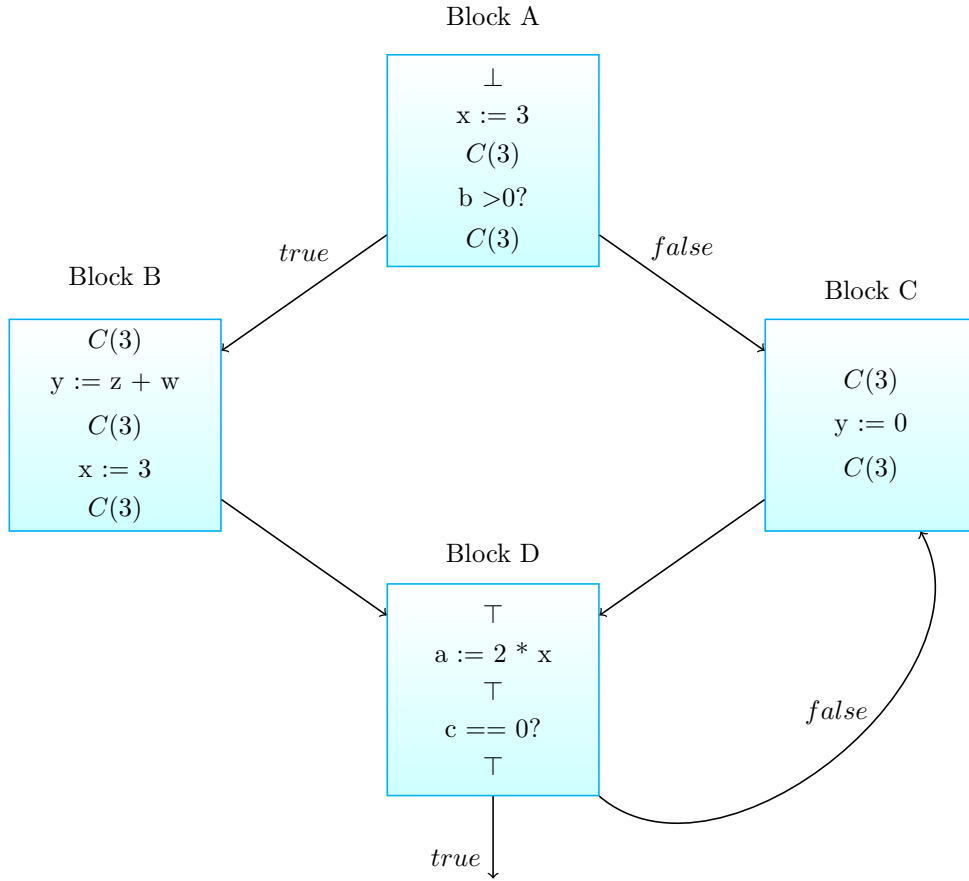
To show how the algorithm could have worked differently, we will change the “ $x := 4$ ” statement to “ $x := 3$ ” in Block C of our same example. This is our new example now.

Figure 33: Example



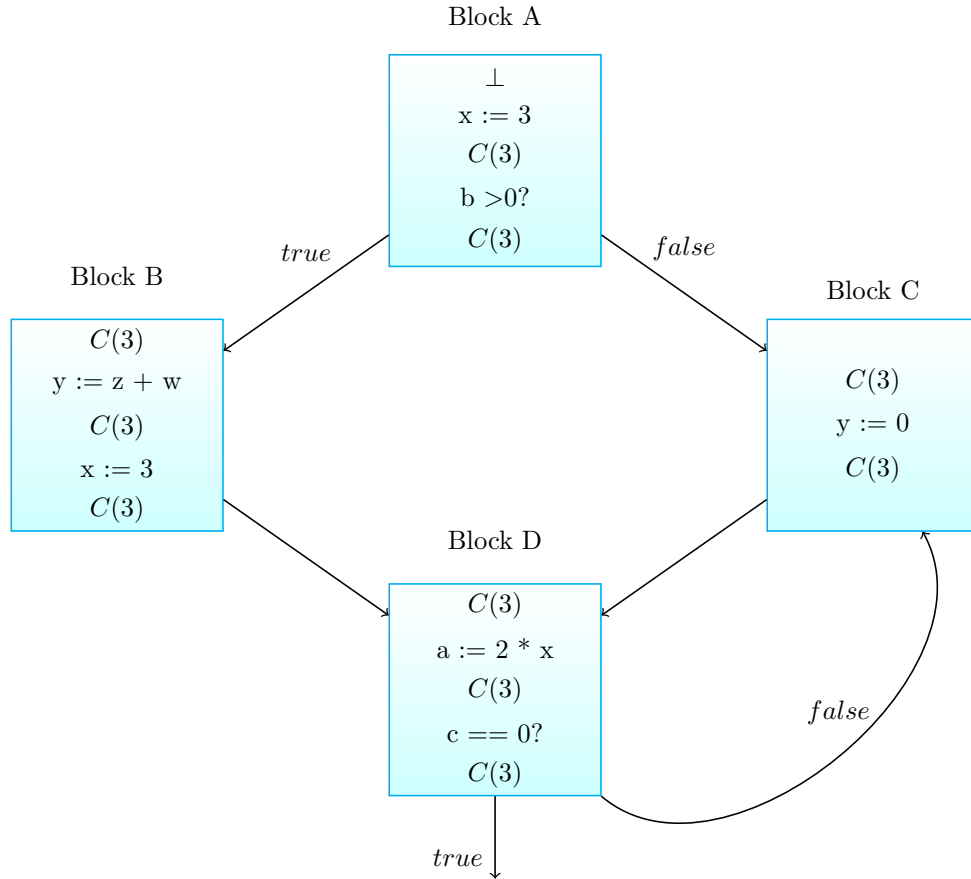
Algorithm will again start by initializing everything to \top , except the entry statement which is initialized to \perp . We will follow the same procedure and eventually reach this point after updating Blocks A, B and C.

Figure 34: “Blocks A, B and C” update



This time the successor of Block D will be updated to constant 3 instead of \perp , because both the predecessors have the same value of constant 3. Updating the successor, and the following instructions in Block D, we reach this stage.

Figure 35: “Block D” update



Question : Have we reached the fixed point now?

Answer : It turns out that, in this case, we have reached the **Fixed Point Solution**. Both the predecessors of Block C, coming from Block A and Block D, have the value of Constant 3, so there is no need to update it.

13 DFA Value Orderings

In this section, we are going to discuss the convergence property of the DFA algorithm. And we will start by defining some arguments for it.

13.1 Orderings

Idea: Simplify the presentation of analysis by ordering the abstract values. We will create an arbitrary operator, “<” (less than), which orders them in the following manner.

$$\perp(\text{bottom}) < C(\text{constant}) < \top(\text{top})$$

This operator is a transitive which means that $\perp(\text{bottom}) < \top(\text{top})$ is also true.

We are going to use this operator to reason about the convergence properties of the DFA algorithm.

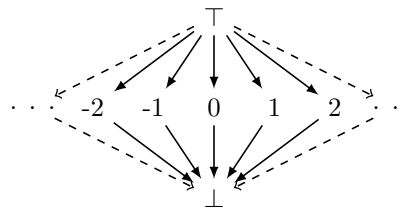
13.2 Partial Ordering

This ordering that we defined is a partial ordering, and not a total ordering. It means that among all the possible values, it is not necessary that all values are comparable. There are some values that are comparable, and some values that are not.

Another way to represent the partial ordering is by using a vertical representation instead of a horizontal representation, and using the directed arrow to represent the “<” (less than) operator. The arrow from \top to C indicates that $C < \top$.



C is not a value, but a place holder for any constant. A better way to represent the ordering would be in the following manner.



Notice that the figure does not have any relation between 0 and 1, or 1 and 2, or between any two constants, in general. That is why it is called a partial ordering, because only some values are related by “<” operator.

Greatest Value: The value which is not less than any other value. In this case, it is \top (top).

Least Value: The value which is not greater than any other value. In this case, it is \perp (bottom).

13.3 Greatest Lower Bound

To build the concept of greatest lower bound, we introduce “ \leq ” (less than equal to) operator which in addition to all the properties of “ $<$ ” (less than) operator, is also reflexive, *i.e.*, $x \leq x \forall x$.

The **greatest lower bound(glb)** of x_1, x_2, \dots, x_n is the greatest value that is lower than (by “ \leq ” operator) $x_i, \forall i$ s.t. $1 \leq i \leq n$.

Here are some examples of glb:

1. $\text{glb}(\top, 1) = 1$
2. $\text{glb}(\top, \perp) = \perp$
3. $\text{glb}(2, \perp) = \perp$
4. $\text{glb}(1, 2, \top) = \perp$
5. $\text{glb}(1, 2) = \perp$

Observation: Glb can be used to replace the first four transfer function rules which defined the relation between the “out of the predecessors” and “in of the successor”.

$$C(s, x, in) = \text{glb}\{C(p, x, out) \mid p \text{ is a predecessor of } s\}$$

13.4 Convergence Argument

The DFA algorithm repeats itself until nothing changes. The only reason for it to not converge is, if it keeps changing forever. There are two reasons for it to keep changing forever.

1. Algorithm can keep taking a different value from the infinite set of values, for the same variable and at the same program point.
2. Algorithm can keep oscillating between the finite number of same values, for the same variable at the same program point.

Using glb, we will prove that the fix point iteration always converges. Here are the convergence arguments:

1. Values start at \top and can only decrease.
2. \top can change to C , and C to \perp .

It cannot happen that \top changes to some constant C , let's say 1 and in a different iteration it changes to a different constant, let's say 0. This is because one of the predecessors would still have the value 1, and if any other predecessor changes its value to a different constant, we are going to take the glb of predecessors which only allows the value to decrease in the order mentioned in this argument. This intuitive argument can be proved more formally by using induction on the Control Flow Graph of the program.

Since this argument does not allow the value to go upwards, *i.e.*, from C to \top , or from \perp to C , or from \perp to \top , oscillation is not possible among these values.

3. Thus, $C(s, x, in/out)$ can change at most twice, $\forall s$ and x .

13.5 Worst Case Execution Time

13.5.1 For One variable

At least one value changes in each iteration, and at every program point a value can change twice therefore:

Number of steps of Fixed Point Iteration Algorithm \leq (Number of C(s, x, in/out) statements) * 2

There can be at most two C(s, x, in/out) statements per each statement therefore:

Number of steps of Fixed Point Iteration Algorithm \leq (Number of program statements) * 4

13.5.2 For all variables

There are two options to deal with all the variables.

1. Run the algorithm separately for each variable.
To calculate the worst execution time, we multiply the number of variables to worst execution time for one variable. The number of variables would be less than the number of program statements, assuming Three Address Code. Therefore, the worst case would be quadratic in size of the program.
2. Keep track of all variables simultaneously.
This can be done by modifying the transfer function such that it looks at every variable simultaneously and updates multiple values in one step. It improves the performance because looking at a set of variables is usually cheaper than looking at each of them separately. It also provides more information while updating the values. For example, if we have a statement like “ $x := y + z$ ”, and we have the information that y and z are constants at this point, then we can also infer that x is a constant. Had we dealt with the variables separately, this would have not been possible. In another words, the analysis would have been weaker. If we do things separately, we may end up with less precise solution in some situations, for some type of analysis.

14 Feb 19 Discussion

- **Jai Javeria:** When discussing global constant propagation, it was said that to apply this transformation, we need to know that property X applies at a particular point and to know that this property X applies at this point we need knowledge of the entire program. However, this is not always true and is not a necessary condition and so it is more accurate to say that to know that this property X applies at this point we *may* need knowledge of the entire program.

- **Sonu Mehta:** How do we apply optimizations? Do we apply local optimizations first and then global?

A: Global optimizations subsume local optimizations. So, typically we'll be more concerned with global optimizations than local optimizations because of the more general nature of global optimizations. Most of the initial optimizations will all be global. However, there are specialized local scenarios where local optimizations come in handy and are helpful. So at the end of the optimization pipeline we apply the local optimizations. These local optimizations are typically difficult to do globally.

- **Shubham Sondhi:** In module 77, it was said that assigning bottom to all program points would be a valid solution but it is actually not a valid solution because it would violate the rule which says that if we do not see a value \top in the "in" of a statement, and the statement is $x:=c$, then we should assign a constant to the out of the statement. Everything \top is also not an answer because it doesn't satisfy the boundary condition that the in of the beginning condition must be \perp . Here, what might work is slightly modifying the rule being violated to say that if "in" of a statement is not \top and the statement is $x:=c$, then the out can be c or lower (which is \perp). With this rule, all \perp will be a valid solution.

- **Anirudh Panigrahi:** In rule 7, it was said that if a statement is $x:=f(\dots)$ and in of this statement is not top, then the out of this statement will be bottom. Is the RHS of this statement necessarily a function or can it be anything which is not a constant?

A: It can be anything which is not a constant. The $f(\dots)$ is just a placeholder for anything not a constant.

- **Jai Javeria:** It was said in Module 78 that the DFA values have a partial order but it would be more accurate to say that its a strict partial order because partial orders can be reflexive.

- **Jai Javeria:** Dataflow was discussed in the contex of tracking one variable's values across different program points. It was mentioned that handling multiple variables might be potentially faster than handling just single variables?

A: There is a time vs. space tradeoff. If we have a table/set of variables storing each variables values at different program points, we could maintain that table with efficient data structures. Details will be clearer when you watch later modules.

- **Sonu Mehta:** Dataflow analysis was discussed in relation to global constant propagation. Can similar things be done for other optimizations as well.

A: Yes. Dataflow analysis is a general framework that can be applied to many different frameworks.

- **Sonu Mehta:** The greatest lower bound (glb) was discussed and it was said that it was transitive and reflexive. Is $\text{glb}(1,1)=1$?

A: Yes. It is reflexive.

- **Indrajit Banerjee:** Instead of just having 3 values of \top , \perp and constant, is it possible to extend this analysis by including a new abstract value which tells whether x is definitely greater than 0 or we cannot say anything about it, at some program point?

A: We need to get some new transfer function rules, for the different kind of statements that would affect the variable in consideration.

- **Arpit Saxena:** Wouldn't it be difficult to define the transfer function rules for a variable to be definitely positive at a program point? Even for a simple statement like $x = x + 1$, we cannot surely say that the "out of the statement" would be greater than 0, given that the "in of the statement" was positive, because the value might overflow in some case.

A: It is tied to the language semantics. A language like C considers interger overflow an undefined behaviour. We can ignore the overflow cases while defining the transfer function rules for a language like C. For other languages, We can maintain a set of potential values that x could have a program point, every time we find a new value that x can take, we add it to the set. But this would take a lot of memory and time, in some cases it may not even converge.

- **Jai Javeria:** Once we convert the language to its intermediate representation like LLVM, do we lose information for example, whether an integer can overflow or not?

A: LLVM preserves information in some cases(for example, whether integer overflow is an undefined behaviour or not), and in other cases it choses the most conservative answer.

- **Vijay Bhardwaj:** How is the dataflow technique applied in the multi-threaded programs, or in case of pointers?

A: For multi threaded programs, we make an assumption that there are no data races, even Java and C makes this assumption. Data races in C are undefined behaviour, that's what allows the compilers to do all the optimizations.

Nothing changes in case of Pointers. Pointer analysis and Aliasing analysis(whether two pointers are pointing to the same location or not) is also modelled as dataflow analysis.

- **Aditya Senthilnathan:** If C assumes that data races are undefined behaviour, does the compiler fail silently or detect such data races and warn user about it?

A: Compiler typically fails silently. Detecting the data races is very expensive and it would make the program very slow, so it is not done in production code.

15 Liveness Analysis

Previously, we saw a type of Global Optimisation called Global Constant Propagation, which required the knowledge of the whole program. After all the constants have been propagated globally, there can be a scope for eliminating dead code.

Consider a statement $x := \dots$

Questions to ask: Is x live or dead just after the statement? What does liveness mean?

Ans: A variable x is live after a statement if it can be used in the downflow logic

A variable x is live at a statement s if:

- There exists a statement s' that uses x
Eg:- $s' : y := f(\dots, x, \dots)$
- There is a path from s' to s (directed)
- The path has no intervening assignments to x

Once we have figured if a variable x live or dead at all statements, then we can easily identify dead code.

A statement $x := \dots$ is dead code if x is dead immediately after the assignment. In this case, as the variable is not being used in the downflow logic, which means that we can simply remove the assignment.

15.1 Liveness Analysis as a DFA

In this analysis, the property that we want to know at a particular program point is the liveness of a particular variable. In the example above, we found the liveness value at one point using the other values. This gives us an intuition for a **Transfer Function**, and hence we use a DFA for this problem as follows:

- Express liveness at a program point based on the liveness of the successor program point (Backward Dataflow)
- The Liveness property for a variable x would take a boolean value
`true` \rightarrow The variable may be live (This is a conservative value)
`false` \rightarrow The variable is definitely dead (not live)

15.2 Ordering of Liveness Values

As seen previously, it really helps to construct a partial ordering because it makes the rules concise. So we do would the same here. Define the ordering as:

$$\text{true} \leq \text{false}$$

Once this ordering has been established, it can be easily seen that the greatest lower bound of any 2 values x, y would be the boolean OR of those values.

$$\begin{aligned} glb(x, y) &= x \vee y \\ glb(x_1, x_2, \dots, x_n) &= x_1 \vee x_2 \vee \dots \vee x_n \end{aligned}$$

16 Liveness DFA rules

Now here we will define the transfer function for the Liveness Analysis. Define the function $L(s, x, in/out)$ as follows:

- x - the variable for which we want to compute the liveness values
- s - a program statement s
- in/out - whether it is input to the statement or output to the statement i.e just before the statement or just after the statement.
 - $L(s, x, in)$ = The liveness value of x just before s
 - $L(s, x, out)$ = The liveness value of x just after s i.e just after statement s is executed

As discussed before, $L(s, x, in/out) \in \{\mathbf{true}, \mathbf{false}\} \forall s, x$.

From now on, the discussion is for a particular variable x , but it can be generalized to more than one variables. We are going to define rules for the following 2 cases:

- **Case 1:** A statement p has one or more successor program points. So in this case, the *out* value of the statement p is expressed as a function of *in* values of the successor program points.
- **Case 2:** For a given statement s , the *in* value is a function of the *out* value of that statement.

It can be observed from here that unlike Global Constant Propagation, which is a Forward Dataflow Analysis, this is a Backward Dataflow Analysis (it starts from the exit point).

16.1 Rules for the Transfer Function:

- $L(p, x, out) = glb\{L(s_i, x, in) \mid s_i \text{ is a successor of } p\}$ If x is live before any of the successors, then x is live after that statement as it may get used in the downflow logic.
- If s is of the form $\dots = f(\dots, x, \dots)$, then

$$L(s, x, in) = \mathbf{true}$$

as the variable x is getting used in this statement.

- If s is of the form $x := e$, where e is an expression which does not refer to x , then

$$L(s, x, in) = \mathbf{false}$$

as we don't need the values of x just above this statement due to x being rewritten

Note: If e referred to x , then Rule #2 would apply

- If s does not refer to x at all (neither updating it, nor using it), then

$$L(s, x, in) = L(s, x, out)$$

These rules are exhaustive. These can be thought of as a system of equations, and our solution should satisfy all these rules.

16.2 Liveness DFA Algorithm

- Initialize $L(s, x, in/out) = \text{false}$ for all statements s and variables x (We start from a more aggressive value)
- Repeat until all program points satisfy Rules 1-4
 - Pick a statement s not satisfying one or more rules in rules 1-4 and update the corresponding $L()$ function value using the appropriate rule

There is a guarantee that this algorithm will converge.

17 Liveness DFA example

Consider the following example:

```
x := 0
while(x != 0){
    x := x + 1
}
return
```

17.1 Liveness DFA Observations

- Every $L()$ in this analysis can change only once ($\text{false} \rightarrow \text{true}$). Now this fact guarantees the convergence of the fixed point algorithm.
- Worst Case execution time: For a particular variable, all the $L()$ can change only once, and as the number of such values is $2 \times \# \text{statements}$, so the worst case execution time is $O(2 \times \# \text{statements})$.
- Once this analysis is finished, we can use it to identify dead code.
- Notice that information flowed in the forward direction (in the direction of the program execution) for constant propagation but flowed in the reverse direction (against the direction of the program execution) for liveness analysis. The former types of analyses are called forward dataflow analyses. The latter types of analyses are called backward dataflow analyses

18 More DFA Examples

18.1 Common Subexpression Elimination

In this analysis, the property that we want to know at a program point is all the expressions available at that point.

- The idea is to maintain a set of available expressions and the temporary in which they are stored, for every program point
- If a subexpression is available in the set of available expressions just before the statement that computes that subexpression, then replace it by the corresponding temporary (the precomputed value)

18.2 Available expressions DFA

In this analysis, the DFA value that we will deal with is a set of available expressions, where each element in this set is a tuple of the register and the expression stored in that register.

So $(x, y + z)$ denotes that the value of the subexpression $y + z$ is stored in the temporary x .

Here also we will set a partial ordering in the values as follows:

$$s_2 \leq s_1 \text{ iff } s_2 \subseteq s_1$$

The lowest value in this ordering is $\{\}$, the empty set. This is a conservative value as it denotes that there is no Subexpression available, hence no subexpression available.

Once this ordering has been established, we can easily see that $glb(s_1, s_2) = s_1 \cap s_2$.

18.3 Transfer Function for Available Expressions

This happens to be a forward dataflow analysis, so we will have 2 types of rules:

- **Case 1:** A statement s has one or more predecessor program points. So in this case, the *in* value of the statement s is expressed as a function of *out* values of the predecessor program points.
- **Case 2:** For a given statement s , the *out* value is a function of the *in* value of that statement.

Define $set_{in}(s)$ be the set of available expressions before the statement s , and set_{out} be the set of available expressions after the statement s . The transfer function has the following rules:

- If s is $x := y + z$, then remove all the set elements that refer to x (all the expressions stored in x and using x) from $set_{in}(s)$. Then add $(x, y + z)$ to $set_{out}(s)$.
- For a statement s and its predecessors, $set_{in}(s) = glb\{set_{out}(p_i) \mid p_i \in \text{predecessor}(s)\}$

Also if s is the starting statement, then the boundary condition for this algorithm would be $set_{in}(s) = \{\}$.

18.4 Copy Propagation

Copy Propagation can be easily modelled as a DFA analysis very similar to Available Expressions DFA, except that we will be limiting ourselves to statements of the form $x := y$. The transformation logic will also be similar.

This optimisation creates opportunities for other global optimizations such that Global Constant Propagation and Liveness Analysis, so it works well in tandem with them.

19 Feb 26th discussion

- **Sonu Mehta** : When we eliminate a common subexpression, would we still add a new mapping? For example, if the statement s in consideration is $d := y + z$, and $set_{in}(s)$ has the mapping $(x, y + z)$, then will we add a mapping for $(d, y + z)$ in $set_{out}(s)$?

A: Yes, we will add a mapping for $(d, y + z)$ after the elimination, despite $y + z$ already being stored in x , as the mapping for x may get killed off at some later point in the analysis, and we still want to keep the information that d contains the value of $y + z$.

- **Vaibhav** : Why would starting from a more aggressive value in DFA give a more precise solution to the fixed point iterative algorithm?

A: This is because that the Semi-Lattice organization of the values is organized in such a way that if $x \leq y$, then it is always okay to replace y with x (x is more relaxed than y).

For example, in liveness analysis, we have that **true** \leq **false**, and the **true** value is a more relaxed as it only says that a variable x "may" be live (figuring out if a variable is definitely *live* is an undecidable problem).

Hence starting from a lower value may give a less precise solution.

- **Vaibhav** : In the fixed point iterative algorithm for DFA analysis, if we have more than one program point that do not satisfy the DFA rules, then would picking one over the other make a difference?

A: Picking one program point over the other in the algorithm would not make a difference with respect to the final solution, but it will make a difference with respect to the efficiency.

- **Arpit Saxena** : Consider a statement of the form $x := e$, and x is dead just after the statement. In our discussion, we declared this statement dead code, and then removed it. But what if the expression e has some side-effects as well, like a function call which modifies the memory?

A: So far we haven't considered function calls in our discussion, but such function calls might have to be handled separately. This can be dealt with in 2 ways:

- Do not remove statements with function calls (Less precise and more conservative)
- The other way is to consider Memory as another variable in the liveness analysis. Assume that memory is dead at the end of the **main** function, and then modify the transfer function (A more precise solution).

- **Jai Arora** : Copy propagation is very similar to Common Subexpression elimination as a DFA Analysis, and they both create opportunities for each other. Also, copy propagation focuses on a subset of all the expressions, so it is possible to modify the transfer function and combine both the analyses?

A: Yes, we can combine both of them together, infact, common subexpression elimination subsumes copy propagation. Hence we can say that common subexpression elimination creates more opportunities for itself.

But copy propagation is a much weaker and a cheaper analysis from the compiler's point of view. Also, sometimes it may just be sufficient to use copy propagation, so it makes sense to keep both the optimizations separate.

- **Anirudh Panigrahi** : How are function calls represented in the intermediate representation? Is it modelled as a jump to a separate basic block?

A: LLVM IR has support for function call abstractions. So a function in a language like C will have a corresponding function in the LLVM IR.

- **Jai Arora** : In Global Constant Propagation, if we keep track of more than one variable simultaneously, then we have a chance to get improvements in the time complexity of the algorithm, but we may get a precision advantage in this case. Is it possible to get a similar advantage in Liveness Analysis?

A: Yes, we can have a more precise but complex liveness analysis if we keep track of all the variables simultaneously.

For example: Consider a statement $s : x := y + z$, where x and y are dead just after the statement. Then we can do the analysis in 2 ways:

- We simply use our previous transfer function, and say that y is live just before the statement as s uses y
- We can also use the fact that x and y both are dead just after the statement, which would mean that y would also be dead before this statement as x is not getting used.

Note that the second way makes the analysis more precise but even more complex. We don't need to do it as we can run original Liveness algorithm multiple times and we would get the same effect.

- **Arpit Saxena** : CPU's are known to reorder instructions for efficiency reasons. Do compilers also do the same, or is it left to the CPU?

A: Yes, compilers also reorder instructions (More examples in later modules). We only reorder instructions only when we preserve the meaning of the program. CPUs use this technique to reduce the number of cycles by taking some unrelated computation and executing it in some stall cycle.

- **Sonu Mehta** : How are pass by reference semantics modelled in the IRs such as LLVM?

A: References will get converted to pointers at the LLVM IR level in this case.