

Compiler Optimizations

HW1

Shambhavi Kuthe
PID: shambhavianil

2. Analysis Passes

2.3.1 Function information

Output:

Implementation of FunctionInfo pass on loop.c

```
user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/LocalOpts
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/FunctionInfo$ make
g++ -rdynamic -I/usr/local/include -std=c++14 -fno-exceptions -fno-rtti -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -I/usr/local/include/ -fPIC -g -O0 -c -o FunctionInfo.o FunctionInfo.cpp
g++ -dlib -shared FunctionInfo.o -o FunctionInfo.so
rm FunctionInfo.o
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/FunctionInfo$ opt -enable-new-pm=0 -load ./FunctionInfo.so -function-info loop.bc -o out
5544 Compiler Optimization Pass
Name:  change_g  Args:  1      Calls:  0      Blocks: 1      Insts: 4      Add/Sub: 1      Mul/Div: 0      Cond. Br: 0      Uncond. Br: 0
Name:  sample  Args:  2      Calls:  0      Blocks: 3      Insts: 9      Add/Sub: 1      Mul/Div: 1      Cond. Br: 1      Uncond. Br: 1
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/FunctionInfo$
```

2.3.2 Local Optimizations Implementation

Algebraic identities: The LocalOpts pass includes a class “Algebraic” which consists of the following 3 functions:

eliminateZero() : It manages the expressions where one operand is 0.

Operand types: Integers, Floating point

Operators: Add, FAdd, Sub, FSub, Mul, FMul

Expressions handled:

$x = y + 0;$	$x = 0 + y;$	Replaces instruction with $x = y;$
$x = y - 0;$		Replaces instruction with $x = y;$
$x = y * 0;$	$x = y * 0;$	Replaces instruction with $x = 0;$

eliminateOne() : It manages the expressions where one operand is 1.

Operand types: Integers, Floating point

Operators: Mul, FMul, SDiv, UDiv, FDiv

Expressions handled:

$x = y * 1;$	$x = 1 * y;$	Replaces instruction with $x = y;$
--------------	--------------	------------------------------------

$x = y / 1;$ *Replaces instruction with $x = y;$*

eliminateSameOperands() : It manages the expressions where both operands are equal.

Operand types: Integers, Floating point

Operators: Sub, FSub, SDiv, UDiv, FDiv

Expressions handled:

$x = y - y;$ *Replaces instruction with $x = 0;$*

$x = y / y;$ *Replaces instruction with $x = 1;$*

Constant folding: The LocalOpts pass includes a class "ConstantFolding " which replaces expressions containing both operands as constants with a constant value.

Operand types: Integers

Operators: Add, Sub, Mul, Div

Expressions handled: If $x = 6;$ $y = 3;$ //Both are constants

$a = x + y;$ *Replaces instruction with $a = 9;$*

$a = x - y;$ *Replaces instruction with $a = 3;$*

$a = x * y;$ *Replaces instruction with $a = 18;$*

$a = x / y;$ *Replaces instruction with $a = 2;$*

Strength reduction: The LocalOpts pass includes a class "StrengthReduction " which replaces expressions containing multiplication of multiples of 2 or division by multiples of 2 both with shift left and shift right operations respectively.

Operand types: Integers

Operators: Mul, Div

Expressions handled:

$a = x * 2;$ *Replaces instruction with $a = x << 1;$*

$a = x / 4;$ *Replaces instruction with $a = x >> 2;$*

Works for all multiples of 2.

Outputs :

Implementation of LocalOpts pass on algebraic.c

```
user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/LocalOpts
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ make
g++ -rdynamic -I/usr/local/include -std=c++14 -fno-exceptions -fno-rtti -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -I/usr/local/include/ -fPIC -g -O0 -c -o LocalOpts.o LocalOpts.cpp
g++ -dlib -shared LocalOpts.o -o LocalOpts.so
rm LocalOpts.o
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ clang -Xclang -disable-O0-optnone -O0 -emit-llvm -c algebraic.c -o algebraic.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -mem2reg algebraic.bc -o algebraic-m2r.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -enable-new-pm=0 -load ./LocalOpts.so -local-opts algebraic-m2r.bc -o out
Transformations:
Algebraic transformations count: 6
Constant folding transformations count: 1
Strength reduction transformations count: 0
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$
```

[2] 0: bash* "user-VirtualBox" 20:09 14-Feb-23

Implementation of LocalOpts pass on constfold.c

```
user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/LocalOpts
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ make
g++ -rdynamic -I/usr/local/include -std=c++14 -fno-exceptions -fno-rtti -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -I/usr/local/include/ -fPIC -g -O0 -c -o LocalOpts.o LocalOpts.cpp
g++ -dlib -shared LocalOpts.o -o LocalOpts.so
rm LocalOpts.o
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ clang -Xclang -disable-O0-optnone -O0 -emit-llvm -c constfold.c -o constfold.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -mem2reg constfold.bc -o constfold-m2r.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -enable-new-pm=0 -load ./LocalOpts.so -local-opts constfold-m2r.bc -o out
Transformations:
Algebraic transformations count: 1
Constant folding transformations count: 5
Strength reduction transformations count: 0
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$
```

[2] 0: bash* "user-VirtualBox" 20:12 14-Feb-23

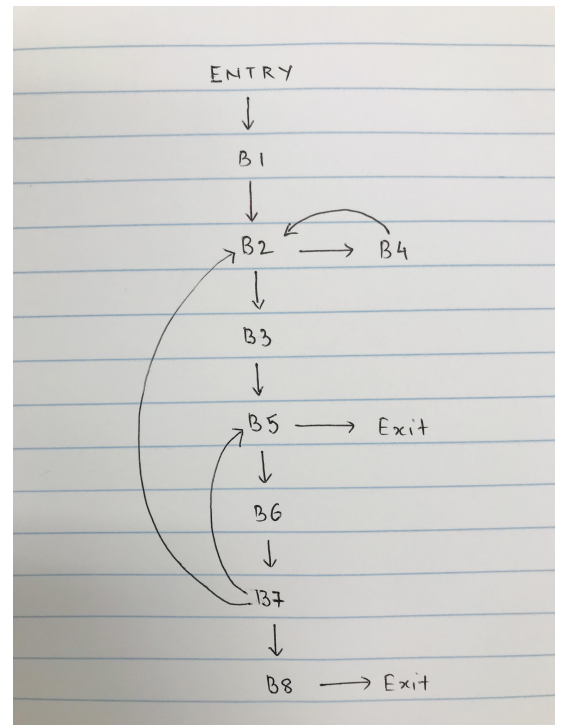
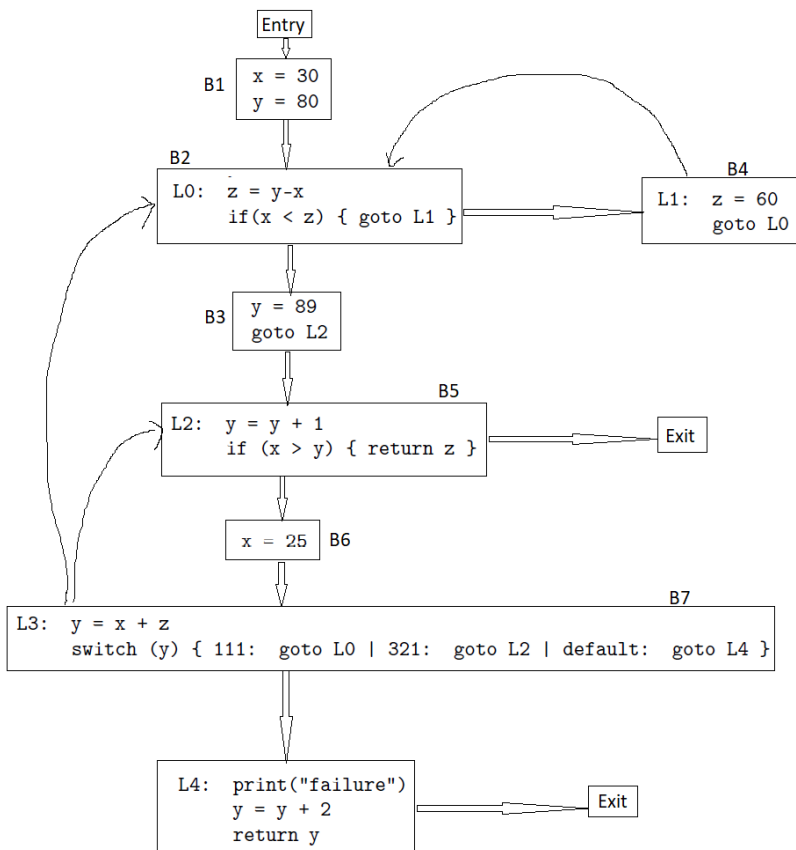
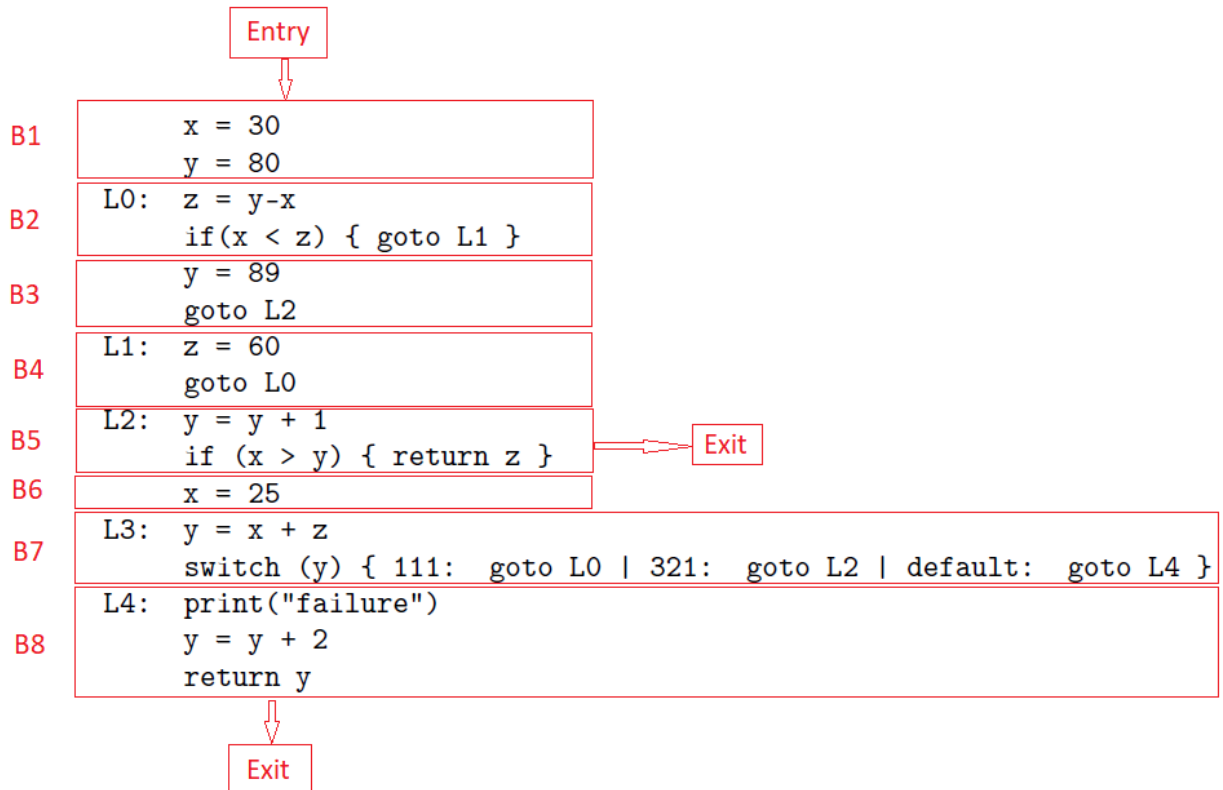
Implementation of LocalOpts pass on strength.c

```
user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/LocalOpts
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ make
g++ -rdynamic -I/usr/local/include -std=c++14 -fno-exceptions -fno-rtti -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -I/usr/local/include/ -fPIC -g -O0 -c -o LocalOpts.o LocalOpts.cpp
g++ -dlib -shared LocalOpts.o -o LocalOpts.so
rm LocalOpts.o
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ clang -Xclang -disable-O0-optnone -O0 -emit-llvm -c strength.c -o strength.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -mem2reg strength.bc -o strength-m2r.bc
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$ opt -enable-new-pm=0 -load ./LocalOpts.so -local-opts strength-m2r.bc -o out
Transformations:
Algebraic transformations count: 1
Constant folding transformations count: 0
Strength reduction transformations count: 5
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/LocalOpts$
```

[2] 0: bash* "user-VirtualBox" 20:10 14-Feb-23

3. Homework Questions

3.1 CFG Basics



3.2 Available Expressions

Expressions are represented by “BlockNumber+DestinationVariable”. For example: expression $a = b + c$ from Block 1 is represented by 1a. Expression $a = a + d$ from block 3 is represented by 3a and so on.

Iteration 1:

BB	e_gen	e_kill	IN[BB]	OUT[BB]
1	1a, 1c, 1e, 1i	2c, 3a, 3e, 6i	Φ	1a, 1c, 1e, 1i
2	2b, 2c, 2f	1c, 4f, 5b	Φ	2b, 2c, 2f
3	3e, 3a	1a, 1e	Φ	3e, 3a
4	4g, 4f	2f	Φ	4g, 4f
5	5b	2b	Φ	5b
6	6i	1i	Φ	6i

Iteration 2:

BB	e_gen	e_kill	IN[BB]	OUT[BB]
1	1a, 1c, 1e, 1i	2c, 3a, 3e, 6i	Φ	1a, 1c, 1e, 1i
2	2b, 2c, 2f	1c, 4f, 5b	1a, 1c, 1e, 1i, 6i	1a, 1c, 1e, 1i, 2b, 2c, 2f, 6i
3	3e, 3a	1a, 1e	2b, 2c, 2f	2b, 2c, 2f, 3e, 3a
4	4g, 4f	2f	2b, 2c, 2f	2b, 2c, 4g, 4f
5	5b	2b	2b, 2c, 2f	2b, 2c, 5b
6	6i	1i	3e, 3a, 4g, 4f, 5b	3e, 3a, 4g, 4f, 5b, 6i

Iteration 3:

BB	e_gen	e_kill	IN[BB]	OUT[BB]
1	1a, 1c, 1e, 1i	2c, 3a, 3e, 6i	Φ	1a, 1c, 1e, 1i
2	2b, 2c, 2f	1c, 4f, 5b	1a, 1c, 1e, 1i, 3e, 3a, 4g, 4f, 5b, 6i	1a, 1c, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i
3	3e, 3a	1a, 1e	1a, 1e, 1i, 2b, 2c, 2f, 6i	1i, 2b, 2c, 2f, 3e, 3a, 6i
4	4g, 4f	2f	1a, 1e, 1i, 2b, 2c, 2f,	1a, 1e, 1i, 2b, 2c, 4g, 4f, 6i

			6i	
5	5b	2b	1a, 1e, 1i, 2b, 2c, 2f, 6i	1a, 1e, 1i, 2c, 2f, 5b, 6i
6	6i	1i	2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b	2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i

Iteration 4:

BB	e_gen	e_kill	IN[BB]	OUT[BB]
1	1a, 1c, 1e, 1i	2c, 3a, 3e, 6i	Φ	1a, 1c, 1e, 1i
2	2b, 2c, 2f	1c, 4f, 5b	1a, 1c, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i
3	3e, 3a	1a, 1e	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i
4	4g, 4f	2f	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1a, 1e, 1i, 2b, 2c, 3e, 3a, 4g, 4f, 6i
5	5b	2b	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1a, 1e, 1i, 2c, 2f, 3e, 3a, 4g, 5b, 6i
6	6i	1i	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i	1a, 1e, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i

Iteration 5: Final

BB	e_gen	e_kill	IN[BB] Final	OUT[BB] Final
1	1a, 1c, 1e, 1i	2c, 3a, 3e, 6i	Φ	1a, 1c, 1e, 1i
2	2b, 2c, 2f	1c, 4f, 5b	1a, 1c, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i
3	3e, 3a	1a, 1e	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i
4	4g, 4f	2f	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1a, 1e, 1i, 2b, 2c, 3e, 3a, 4g, 4f, 6i
5	5b	2b	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 6i	1a, 1e, 1i, 2c, 2f, 3e, 3a, 4g, 5b, 6i
6	6i	1i	1a, 1e, 1i, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i	1a, 1e, 2b, 2c, 2f, 3e, 3a, 4g, 4f, 5b, 6i

Iteration 5 matches with iteration 4. Thus, Iteration 5 above is the final iteration for the available expressions for each basic block.

3.3 Faint Analysis

1. Define the set of elements that your analysis operates on.

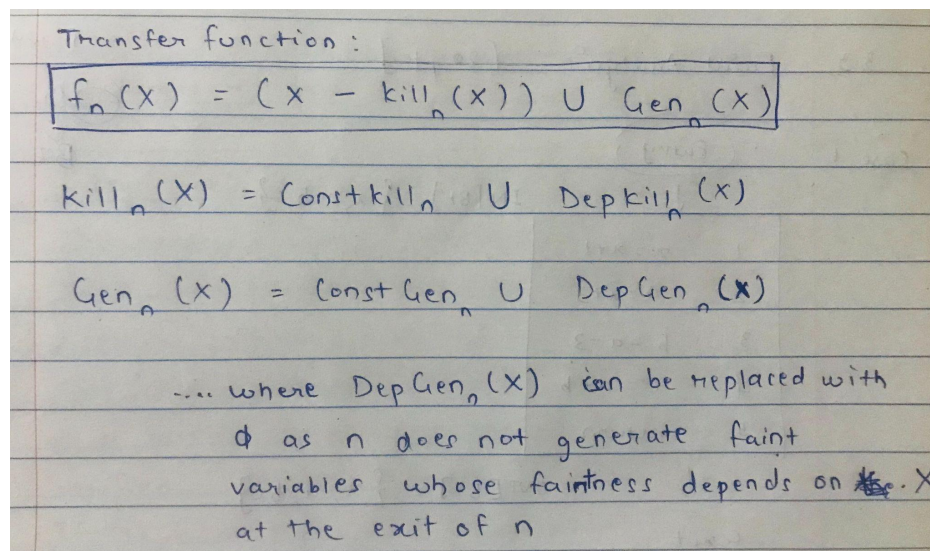
Faint analysis operates on the variables that are faint i.e. if along every path following the assignment of the variable, the variable is either dead or is used in an expression whose LHS is also faint. Thus, the analysis operates on all the faint variables X . All expressions where variables are not used on the RHS, used in return statements, used in conditional statement checks and are a part of any arguments to the functions will be included in the set of elements that the analysis will be operated on.

2. Define the direction of your analysis.

The analysis is performed in the backward direction which is necessary to identify the faint variables by checking if a variable is dead or being used in an expression where the LHS is also a faint variable in the downstream.

3. Define the transfer function. Make sure to define any sets that your transfer function uses.

The transfer function is as follows:



Transfer function:

$$f_n(X) = (X - \text{kill}_n(X)) \cup \text{Gen}_n(X)$$
$$\text{kill}_n(X) = \text{Constkill}_n \cup \text{Depkill}_n(X)$$
$$\text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X)$$

... where $\text{DepGen}_n(X)$ can be replaced with \emptyset as n does not generate faint variables whose faintness depends on ~~the~~ X at the exit of n

For a given set of faint variables X at the exit of a block, the transfer function $f_n(X)$ for each basic block n is determined by eliminating the kill set $\text{kill}_n(X)$ from the faint variable X and then applying union with the generator set $\text{Gen}_n(X)$.

The kill set $\text{kill}_n(X)$ consists of the faint variables killed by the block and $\text{Gen}_n(X)$ consists of the faint variables generated by the block. The kill set is a union of the $\text{Constkill}_n(X)$ and

Depkill_n(X) where **Constkill_n(X)** does not require faint variables at the input while **Depkill_n(x)** takes faint variables at the input.

$$\text{Kill}_n = \begin{cases} \{x\} & \text{if } n \text{ is use}(x) \\ \phi & \text{otherwise} \end{cases}$$

$$\text{Depkill}_n(X) = \begin{cases} \text{operand}(e) & \text{if } n \text{ is } n=e, x \notin X \\ \phi & \text{otherwise} \end{cases}$$

The gen set **Genn(X)** is a union of **ConstGenn(X)** faint variables generated by the block and the **DepGenn(X)** faint variables. However, the **DepGenn(x)** can be replaced with void ϕ as 'n' does not generate any faint variables which are dependent on the faintness of the variables at the exit of the block.

$$\text{Gen}_n = \begin{cases} \{x\} & n \text{ is an assignment of the form } x=e, x \notin \text{operand}(e) \\ \{x\} & n \text{ is read}(x) \\ \phi & \text{otherwise} \end{cases}$$

4. Define the meet operator, and give the equation that uses this operator.

The meet operator for the analysis is "**Intersection \cap** ". Intersection operator is used to define the relationship between the input and the output set of faint variables of a basic block. Since it is a Backwards flow analysis, we compute the OUT set first followed by the IN set. The sets using the meet operator are defined as follows:

$$\text{In}_n = \text{fn}(\text{out}_n)$$

$$\text{Out}_n = \bigcap_{s \in \text{successors}(n)} \text{In}_s$$

IN set is obtained by applying the transfer function $\text{fn}(x)$ to the OUT set. OUT set is obtained by considering the intersection of the inputs to all the successors of the block.

Intersection operator is used for the inputs of the successor because any variable which is faint in one successor block can be used in the downstream of some other successor block i.e. a faint variable in one successor block might not be faint in another successor block.

5. What are the value(s) that ENTRY and/or EXIT is initialized to?

The initial faint variables at the entry of the EXIT block need to be determined as our analysis is in the backward direction. The EXIT block is initialized to all the local variables in the function i.e. $IN[EXIT] = Var$ where Var is a set of all the local variables.

6. What are the value(s) that the IN and/or OUT sets are initialized to?

Initial values of IN set of a basic block is the set of all the variables used in the function i.e.

$IN[B_i] = Var$ where Var represents all the expressions of a function such that when the intersection operator is applied, only the faint variables will remain. The $OUT[B_i]$ set can be determined by applying intersection to the IN sets of all the successors of the block. Thus, due to intersection, the OUT set will result in all the common faint variables from all the successor IN sets which were initialized to all local variables.

7. What impact, if any, does the order of basic block traversal have on the correctness of your analysis? What order would you implement and why?

In the case of faint variable analysis, the order of the basic block traversal in the backward direction won't affect the correctness of the analysis. This is because, the meet operator being an intersection operator will eventually eliminate all the unique variables and the common faint variables will be extracted at the confluence of the basic blocks.

8. Will your analysis converge? Please explain in a few sentences; no proof is necessary.

Yes, the analysis will converge with the output as all the common faint variables which are either dead or are only used in expressions which have a faint variable in the LHS. For each iteration, more information will be gathered i.e. as the faint variables will be discovered or killed through each path, the intersection operator at the confluence will eventually eliminate all the non-faint variables by detecting their use in the basic blocks and converge on the faint variables.

9. In pseudo-code, give an algorithm that performs the data flow analysis and removes faint expressions, using your definitions from the previous sub-problems.

1. Initialize the $IN[EXIT]$ set and $IN[B_i]$ set of all the basic blocks to all the local variables in the function.
2. Traversing in the backward direction:
For each Basic block:
 - a. Determine the faint variables X , $killIn(X)$ and $Genn(X)$ sets from the above expressions.
 - b. Determine the $OUT[B_i]$ set using: $OUTn[B_i] = \cap Ins$ where s consists of all the successors of the basic block.
 - c. Determine the $IN[B_i]$ set using the transfer function $fn(X)$ and $OUT[B_i]$ set.
3. Repeat step 2 by considering initial values from the output of the previous iterations.
Repeat till 2 consecutive iterations match and we obtain the faint variables in $IN[B_1]$ set i.e. at the entry of the 1st basic block.

4. Erase the faint expressions.

10. Give a real-world example where faint analysis can be used to optimize IR.(explanation is enough, IR/code is not required)

Faint analysis can be used to optimize search engine complex algorithms as search engines return relevant results to queries asked by the user. For example, if a user searches “restaurants near me”, it is possible that the keywords “restaurants” and “near” are extracted and the keyword “me”, because of its less relevance might be skipped. However, if using faint variable analysis, the subtle keywords are also detected, it will improve the results given by the search engine as the results will now include subtle relevant data for the user with restaurants near the location of the user. Thus, faint variable analysis can be used to optimize IR by detecting weaker and subtle patterns in complex codes.