

# Compiler Optimization Notes

November 16, 2022

## Contents

<b>1</b>	<b>Local Optimizations</b>	<b>1</b>
1.1	Basic Blocks/Flow graphs	1
1.1.1	Basic Blocks	1
1.1.2	Flow graphs	1
1.1.3	Partitioning into Basic Blocks	2
1.1.4	Reachability of Basic Blocks	2
1.2	Local optimizations	3
1.2.1	common subexpression elimination	3
1.3	Abstraction 1:DAG	4
1.3.1	How well do DAGs hold up across statements?	4
1.4	Abstraction 2:Value numbering	5
1.4.1	Algorithm	5
1.4.2	Example	6
<b>2</b>	<b>Introduction to Data Flow Analysis</b>	<b>7</b>
2.1	Motivation for Dataflow Analysis	7
2.1.1	What is Data Flow Analysis?	7
2.1.2	Static Program vs. Dynamic Execution	8
2.1.3	Data Flow Analysis Schema	8
<b>3</b>	<b>Live Variabl Analysis</b>	<b>10</b>
3.1	Motivation	10
3.2	Problem formulation	10
3.3	Semantic vs. syntactic	10
<b>4</b>	<b>Reaching Definitions</b>	<b>12</b>
4.1	Iterative Algorithm	12
4.2	Worklist Algorithm	12
4.3	Example	12

<b>5</b>	<b>Available Expressions Analysis</b>	<b>13</b>
5.1	Motivation . . . . .	13
5.2	Backgroud Knowledge . . . . .	13
5.3	Problem Formulation . . . . .	13
5.4	Semantic vs. Syntactic . . . . .	14
<b>6</b>	<b>Foundations of Data Flow Analysis</b>	<b>16</b>
6.1	A Unified Framework . . . . .	16
6.2	Partial Order . . . . .	16
6.3	Lattice . . . . .	16
6.4	Complete Lattice . . . . .	17
6.5	Semi-Lattice . . . . .	17
6.6	Meet Operator . . . . .	17
6.7	Descending Chain . . . . .	18
6.8	Transfer Functions . . . . .	18
6.9	Monotonicity . . . . .	19
6.10	Distributivity . . . . .	19
<b>7</b>	<b>Introduction to Static Single Assignment</b>	<b>20</b>
7.1	Definition-Use and Use-Definition Chains . . . . .	20
7.2	Static Single Assignment(SSA) . . . . .	21
7.2.1	Why SSA is useful? . . . . .	21
7.3	How to represent SSA? . . . . .	21
7.3.1	How does the $\phi$ -function know which edge was taken? . . . . .	22
7.4	Converting to SSA form . . . . .	22
7.4.1	Trivial SSA . . . . .	22
7.4.2	Minimal SSA . . . . .	23
7.4.3	Path-convergence criterion . . . . .	24
7.4.4	Dominance property of SSA form . . . . .	24
7.5	Computing the dominance frontier . . . . .	27
7.6	Inserting $\Phi$ -functions . . . . .	28
7.7	Renaming the variables . . . . .	29
7.8	Edge Splitting . . . . .	29
<b>8</b>	<b>SSA-Style optimizations</b>	<b>31</b>
8.1	Constant Propagation . . . . .	31
8.2	Conditional Constant Propagation . . . . .	31
8.2.1	Example . . . . .	32
8.3	Copy Propogation . . . . .	34

## 1 Local Optimizations

Local Optimizations never goes away because this is always a piece of what happens even when we talk about even more sophiscated types of optimizations.

First we will talk about how to represent the code within a function or procedure, that's using something called a flow graph which is made of basic blocks. Next we will contrast two different abstractions for doing local optimizations.

## 1.1 Basic Blocks/Flow graphs

### 1.1.1 Basic Blocks

A basic block is a sequence of instructions(3-address statements). There are some requirements for basic block:

- **Only the first instruction can be reached from outside the block.** The reason why this property is useful is that within a basic block, we just march instruction by instruction through the block, this simplifies things at least within a basic block.
- **All the statements are executed consecutively if the first one is.**
- **The basic block must be maximal.** i.e., they cannot be made larger without violating conditions.

### 1.1.2 Flow graphs

Flow graph is a graph representation of the procedure. In flow graph, basic blocks are the nodes, and the edge for  $B_i \rightarrow B_j$  stands for a path from node  $B_i$  to node  $B_j$ . So how will  $B_i \rightarrow B_j$  happen? There are two possibilities:

- Either first instruction of  $B_j$  is the target of a goto at end of  $B_i$ .
- $B_j$  physically follows  $B_i$  which doesn't end in an unconditional goto.

### 1.1.3 Partitioning into Basic Blocks

- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader and ends at instruction immediately before a leader(or the last instruction).

An example of flow graph is shown below:

### 1.1.4 Reachability of Basic Blocks

There is one thing interesting need to mention here. So the source code is below:

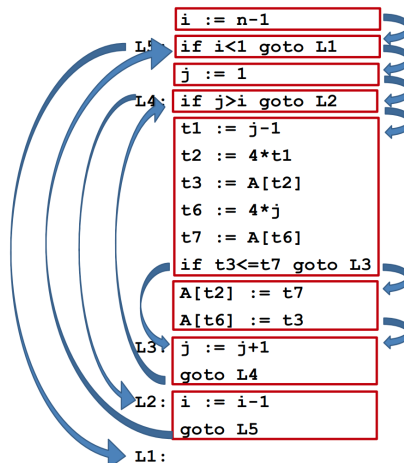


Figure 1: Example of a flow graph

```

1  if x {
2      ...
3      return;
4  } else {
5      ...
6  }

```

Listing 1: An example

The corresponding flow graph is shown in 2:

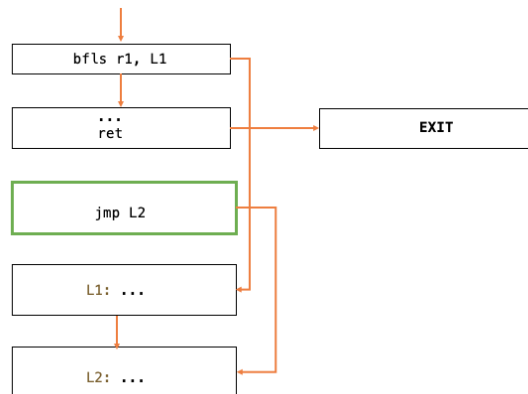


Figure 2: Example of a flow graph

We can see that the box in green is unreachable from the entry. So why is that interesting? Typically, after compilers construct the control flow graph, they will go through and remove any

unreachable nodes. Just do depth first traversal of the graph from the entry node and mark all those visited nodes. So unmarked nodes will be deleted. This will help the compiler get a better optimization result.

So why do these unreachable nodes appear? The answer is it is not the job of the front-end of the compiler to clean up the unreachable nodes.

## 1.2 Local optimizations

Local optimizations are those occur **within the basic blocks**.

### 1.2.1 common subexpression elimination

There're some types of local optimizations. One is called **common subexpression elimination**. Subexpressions are some arithmetic expressions that occur on the right hand of the instructions. The goal of this common subexpression elimination is to identify expressions that are guaranteed to produce identical values at runtime and arrange to only perform the associated computation once (when the first instance of the expression is encountered).

```
2 a = b + c ;  
  d = b + c ;
```

Listing 2: Subexpression example

In the example 2,  $b + c$  is so called common subexpression, we could replace the instruction containing common subexpression with an assign expression.

```
2 a = b + c ;  
  d = a
```

Listing 3: code snippet applied common subexpression elimination to 2

You may wonder why this kind of redundancy can occur in code? Are we programmers stupid to do so? In fact, the redundancy most comes from the stage when compilers turn your source code. For example, **when you use arrays**, you need to do some arithmetic to generate the address of the array element you are accessing. So every time you reference the same array element, compiler will calculate the same address again. Similarly, if you **access offsets within fields**. Last example is **access to parameters** in the stack.

## 1.3 Abstraction 1:DAG

DAG is the acronym for Directed Acyclic Graph. The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. DAG is an efficient method for identifying common sub-expressions.<sup>1</sup>

The parse tree and DAG of the expression  $a + a * (b + c) + (b + c) * d$  is shown in 3.

In DAG, some of the computation are reused. So we can generate optimized code based on DAG.

The optimized code for the DAG3 is:

---

<sup>1</sup>copied from <https://wildpartyofficial.com/what-is-dag-in-compiler-construction>

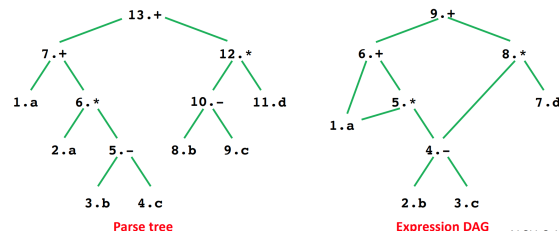


Figure 3: Example of a DAG

```

1      t1 = b - c;
2      t2 = a * t1;
3      t3 = a + t2;
4      t4 = t1 * d;
5      t5 = t3 + t4;

```

Listing 4: code

### 1.3.1 How well do DAGs hold up across statements?

We have seen that DAGs can be useful in a long arithmetic expression. So how well do DAGs perform in sequence of instructions?

```

1      a = b + c;
2      b = a - d;
3      c = b + c;
4      d = a - d;

```

Listing 5: code

The corresponding DAG is shown in 4.

Based on the DAG4, one optimizaed code is 6

```

1      a = b+c;
2      d = a-d;
3      c = d+c;

```

Listing 6: code

6 is not correct. B need to be overwritten but not yet. So if using DAGs, you need to be very careful.

DAGs make sense if you just have one long expression, but once you have sequence of instructions overwriting variables , DAGs are less appealing because this abstraction doesn't really include the concept of time.

## 1.4 Abtraction 2:Value numbering

We have seen drawbacks of DAGs. One way to fix the problem is to attach variable name to latest value. Value numbering is such abstraction.

The idea behind value numbering is there is a mapping between variables(static) to values(dynamic). So common subexpression means same value number.

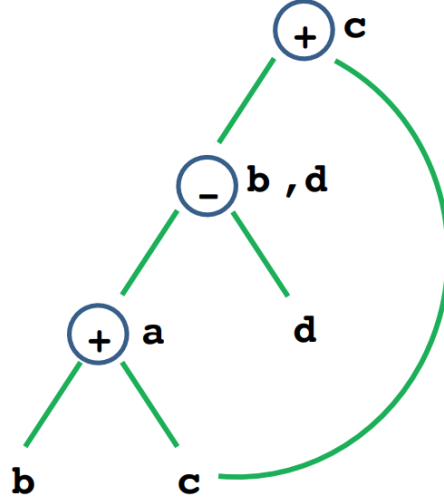


Figure 4: Example of a DAG

#### 1.4.1 Algorithm

```

1 Data structure:
  VALUES = Table of
3   expression /* [OP, valnum1, valnum2] */
   var /* name of variable currently holding expr */
5 For each instruction (dst = src1 OP src2) in execution order
  valnum1=var2value(src1); valnum2=var2value(src2)
7
  IF [OP, valnum1, valnum2] is in VALUES
9   v = the index of expression
   Replace instruction with: dst = VALUES[v].var
11 ELSE
   Add
13   expression = [OP, valnum1, valnum2]
   var = tv
15   to VALUES
   v = index of new entry; tv is new temporary for v
17   Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
   dst = tv
19 set_var2value (dst, v)

```

Listing 7: code

#### 1.4.2 Example

Figure 5 shows a concrete example of how VN identifies computation redundancies within a basic block. The VN processes each instruction statically. It obtains the previously computed symbolic value of each operand on the RHS, assigning a unique number on encountering a new operand. Then, it hashes the symbolic values assigned to operands together with the operator to obtain a

1. $w = a^1 * b^2$	$\langle *, 1, 2 \rangle = 3$ ; $VN(w) = 3$
2. $x = w^3 + c^4$	$\langle +, 3, 4 \rangle = 5$ ; $VN(x) = 5$
3. $d = a^1$	$VN(d) = VN(a) = 1$
4. $e = b^2$	$VN(e) = VN(b) = 2$
5. $y = d^1 * e^2$	$\langle *, 1, 2 \rangle$ <b>redundant!</b> $VN(y) = 3$
6. $z = y^3 + c^4$	$\langle +, 3, 4 \rangle$ <b>redundant!</b> $VN(z) = 5$

Figure 5: An example of value numbering.

symbolic value for the computation. If the computed symbolic value for a computation is already present in the table of previously computed values, then the current computation is redundant. In this basic block, computations on Line 5 and 6 are redundant since the computations are already computed by instruction on Line 1 and 2. <sup>2</sup>

---

<sup>2</sup>copied from [https://www.researchgate.net/publication/283214075\\_Runtime\\_Value\\_Numbering\\_A\\_Profiling\\_Technique\\_to\\_Pinpoint\\_Redundant\\_Computations](https://www.researchgate.net/publication/283214075_Runtime_Value_Numbering_A_Profiling_Technique_to_Pinpoint_Redundant_Computations)



## 2 Introduction to Data Flow Analysis

### 2.1 Motivation for Dataflow Analysis

Some optimizations<sup>3</sup>, however, require more "global" information. For example, consider the code 8

```
1  a = 1;  
   b = 2;  
3  c = 3;  
   if (...) x = a + 5;  
5  else x = b + 4;  
   c = x + 1;
```

Listing 8: An

In this example, the initial assignment to  $c$  (at line 3) is useless, and the expression  $x + 1$  can be simplified to 7, but it is less obvious how a compiler can discover these facts since they cannot be discovered by looking only at one or two consecutive statements. A more global analysis is needed so that the compiler knows at each point in the program:

- which variables are guaranteed to have constant values, and
- which variables will be used before being redefined.

To discover these kinds of properties, we use dataflow analysis.

#### 2.1.1 What is Data Flow Analysis?

Local Optimizations only consider optimizations within a node in CFG. Data flow analysis will take edges into account, which means composing effects of basic blocks to derive information at basic block boundaries. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program.

Typically, we will do local optimization for the first step to know what happens in a basic block, step 2 is to do data flow analysis. In the third step, we will go back and revisit the individual instructions inside of the blocks.

Data flow analysis is **flow-sensitive**, which means we take into account the effect of control flow. It is also a **intraprocedural analysis** which means the analysis is within a procedure. Data-flow analysis computes its solutions over the paths in a control-flow graph. The well-known, meet-over-all-paths formulation produces safe, precise solutions for general dataflow problems. All paths-whether feasible or infeasible, heavily or rarely executed-contribute equally to a solution.

Here are some examples of intraprocedural optimizations:

- **constant propagation.** Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program

---

<sup>3</sup>based on <https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html>

as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further.

- **common subexpression elimination**
- **dead code elimination.** Actually, source code written by programmers doesn't contain a lot of dead code, dead code happens to occur partly because of how the front end translates code into the IR. Doing optimizations will also turn code into dead.

### 2.1.2 Static Program vs. Dynamic Execution

Program is statically finite, but there can be infinite many dynamic execution paths. On one hand, analysis need to be precise, so we will take into account as much dynamic execution as possible. On the other hand, analysis need to do the analysis quickly. For a compromise, the analysis result is **conservative** and what it does is for each point in the program, combines information of all the instances of the same program point.

### 2.1.3 Data Flow Analysis Schema

Before thinking about how to define a dataflow problem, note that there are two kinds of problems:

- Forward problems (like constant propagation) where the information at a node  $n$  summarizes what can happen on paths from "enter" to  $n$ . So if we care about what happened in the past, it's a forward problem.
- Backward problems (like live-variable analysis), where the information at a node  $n$  summarizes what can happen on paths from  $n$  to "exit". So if we care about what will happen in the future, it's a backward problem.

In what follows, we will assume that we're thinking about a forward problem unless otherwise specified.

Another way that many common dataflow problems can be categorized is as may problems or must problems. The solution to a "may" problem provides information about what may be true at each program point (e.g., for live-variables analysis, a variable is considered live after node  $n$  if its value may be used before being overwritten, while for constant propagation, the pair  $(x, v)$  holds before node  $n$  if  $x$  must have the value  $v$  at that point).

Now let's think about how to define a dataflow problem so that it's clear what the (best) solution should be. When we do dataflow analysis "by hand", we look at the CFG and think about:

- What information holds at the start of the program.
- When a node  $n$  has more than one incoming edge in the CFG, how to combine the incoming information (i.e., given the information that holds after each predecessor of  $n$ , how to combine that information to determine what holds before  $n$ ).
- How the execution of each node changes the information.

This intuition leads to the following definition. An instance of a dataflow problem includes:

- a *CFG*,
- a domain  $D$  of "dataflow facts",
- a dataflow fact "init" (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- an operator  $\wedge$  (used to combine incoming information from multiple predecessors),
- for each *CFG* node  $n$ , a dataflow function  $f_n : D \rightarrow D$  (that defines the effect of executing  $n$ ).

For constant propagation, an individual dataflow fact is a set of pairs of the form (var, val), so the domain of dataflow facts is the set of all such sets of pairs (the power set). For live-variable analysis, it is the power set of the set of variables in the program.

For both constant propagation and live-variable analysis, the "init" fact is the empty set (no variable starts with a constant value, and no variables are live at the end of the program).

For constant propagation, the combining operation  $\wedge$  is set intersection. This is because if a node  $n$  has two predecessors,  $p1$  and  $p2$ , then variable  $x$  has value  $v$  before node  $n$  iff it has value  $v$  after both  $p1$  and  $p2$ . For live-variable analysis,  $\wedge$  is set union: if a node  $n$  has two successors,  $s1$  and  $s2$ , then the value of  $x$  after  $n$  may be used before being overwritten iff that holds either before  $s1$  or before  $s2$ . In general, for "may" dataflow problems,  $\wedge$  will be some union-like operator, while it will be an intersection-like operator for "must" problems.

For constant propagation, the dataflow function associated with a *CFG* node that does not assign to any variable (e.g., a predicate) is the identity function. For a node  $n$  that assigns to a variable  $x$ , there are two possibilities:

- 1. The right-hand side has a variable that is not constant. In this case, the function result is the same as its input except that if variable  $x$  was constant the before  $n$ , it is not constant after  $n$ .
- 2. All right-hand-side variables have constant values. In this case, the right-hand side of the assignment is evaluated producing constant-value  $c$ , and the dataflow-function result is the same as its input except that it includes the pair  $(x, c)$  for variable  $x$  (and excludes the pair for  $x$ , if any, that was in the input).

For live-variable analysis, the dataflow function for each node  $n$  has the form:  $f_n(S) = Gen_n \cup (S - Kill_n)$ , where  $Kill_n$  is the set of variables defined at node  $n$ , and  $Gen_n$  is the set of variables used at node  $n$ . In other words, for a node that does not assign to any variable, the variables that are live before  $n$  are those that are live after  $n$  plus those that are used at  $n$ ; for a node that assigns to variable  $x$ , the variables that are live before  $n$  are those that are live after  $n$  except  $x$ , plus those that are used at  $n$  (including  $x$  if it is used at  $n$  as well as being defined there).

An equivalent way of formulating the dataflow functions for live-variable analysis is:  $f_n(S) = (S \cap NOT - Kill_n) \cup Gen_n$ , where  $NOT - Kill_n$  is the set of variables not defined at node  $n$ . The advantage of this formulation is that it permits the dataflow facts to be represented using bit vectors, and the dataflow functions to be implemented using simple bit-vector operations (and or).

It turns out that a number of interesting dataflow problems have dataflow functions of this same form, where  $Gen_n$  and  $Kill_n$  are sets whose definition depends only on  $n$ , and the combining operator  $\wedge$  is either union or intersection. These problems are called GEN/KILL problems, or bit-vector problems.

## 3 Live Variable Analysis

In compilers, live variable analysis (or simply liveness analysis) is a classic data-flow analysis to calculate the variables that are live at each point in the program. A variable is live at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to. <sup>4</sup>

### 3.1 Motivation

Programs may contain

- code which gets executed but which has no useful effect on the program's overall result;
- occurrences of variables being used before they are defined;
- many variables which need to be allocated registers and/or memory locations for compilation.

The concept of variable liveness is useful in dealing with all three of these situations.

### 3.2 Problem formulation

Liveness is a data-flow property of variables: "Is the value of this variable needed?" We therefore usually consider liveness from an instruction's perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

### 3.3 Semantic vs. syntactic

<sup>5</sup>

There are two kinds of variable liveness : Semantic liveness and Syntactic liveness.

A variable  $x$  is **semantically** live at a node  $n$  if there is some execution sequence starting at  $n$  whose (externally observable) behaviour can be affected by changing the value of  $x$ . Semantic liveness is concerned with the execution behaviour of the program.

A variable is **syntactically** live at a node if there is a path to the exit of the flow graph along which its value may be used before it is redefined. Syntactic liveness is concerned with properties of the syntactic structure of the program.

So what is the difference between Semantic liveness and Syntactic liveness? syntactic liveness is a computable approximation of semantic liveness.

Consider the example

```
2  int t = x * y;  
   if ((x+1)*(x+1) == y) {  
4     t = 1;  
   }  
   if (x*x + 2*x + 1 != y) {  
6     t = 2;  
   }  
8  return t;
```

Listing 9: An

---

<sup>4</sup>based on Wikipedia

<sup>5</sup>based on slides from Cambridge University

In fact, `t` is dead in node `int t = x;` because one of the conditions will be true, so on every execution path `t` is redefined before it is returned. The value assigned by the first instruction is never used.

But on read path from 6 through the flowgraph, `t` is not redefined before it's used, so `t` is syntactically live at the first instruction. Note that this path never actually occurs during execution.

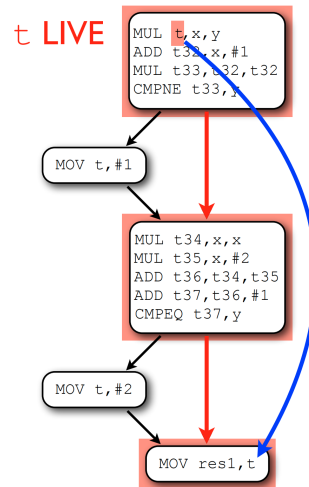


Figure 6: CFG

## 4 Reaching Definitions

The Reaching Definitions Problem is a data-flow problem used to answer the following questions: Which definitions of a variable  $X$  reach a given use of  $X$  in an expression? Is  $X$  used anywhere before it is defined? A definition  $d$  reaches a point  $p$  if there exists path from the point immediately following  $d$  to  $p$  such that  $d$  is not killed(overwritten) along that path.

### 4.1 Iterative Algorithm

Here is the iterative algorithm.

---

#### Algorithm 1 Reaching Definitions: Iterative Algorithm

---

**Input:** control flow graph  $CFG = (N, E, \text{Entry}, \text{Exit})$

---

```

out[Entry] =  $\emptyset$  ▷ Boundary condition
for each basic block B other than Entry do
    out[B] =  $\emptyset$  ▷ Initialization for iterative algorithm
end for
while Changes to any out[] occur do
    for each basic block B other than Entry do
        in[B] =  $\cup(out[p])$ , for all predecessors p of B
        out[B] =  $f_B(in[B])$  ▷  $out[B] = gen[B] \cup (in[B] - kill[B])$ 
    end for
end while

```

---

### 4.2 Worklist Algorithm

### 4.3 Example

Here comes an example of reaching definition.

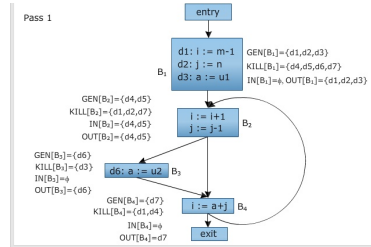


Figure 7: Pass 1

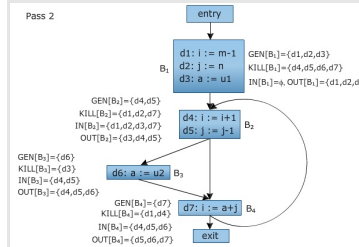


Figure 8: Pass 2

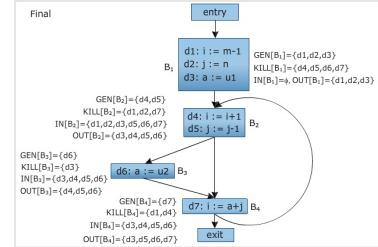


Figure 9: Pass 3

---

**Algorithm 2** Reaching Defintions:Worklist Algorithm

---

**Input:** control flow graph  $CFG = (N, E, \text{Entry}, \text{Exit})$

```
out[Entry] =  $\emptyset$  ▷ Boundary condition
ChangedNodes = N
for each basic block B other than Entry do
    out[B] =  $\emptyset$  ▷ Initialization for iterative algorithm
end for
while ChangedNodes  $\neq \emptyset$  do
    Remove i from ChangedNodes
     $in[B] = \cup(out[p])$ , for all predecessors p of B
    oldout = out[i]
    out[i] =  $f_i(in[i])$  ▷  $out[i] = gen[i] \cup (in[i] - kill[i])$ 
    if oldout  $\neq out[i]$  then
        for all successors s of i do
            add s to ChangedNodes
        end for
    end if
end while
```

---

## 5 Available Expressions Analysis

### 5.1 Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program. The concept of expression availability is useful in dealing with this situation.

### 5.2 Backgroud Knowledge

Any given program contains a finite number of expressions (i.e. computations which potentially produce values),so we may talk about the set of all expressions of a program. Consider the program in

```
int z = x * y;
print s + t;
int w = u / v;
```

Listing 10: An

This program contian expression  $x*y, s+t, u/v$ .

### 5.3 Problem Formulation

Availability is a data-flow property of expressions: “Has the value of this expression already been computed?” At each instruction, each expression in the program is either available or unavailable. So each instruction(or node of the flowgraph) has an associated set of available expression.

## 5.4 Semantic vs. Syntactic

An expression is *semantically* available at a node  $n$  if its value gets computed (and not subsequently invalidated) along every execution sequence ending at  $n$ .

```
int x = y * z;
⋮
return y * z; y*z AVAILABLE
```

Figure 10: Available expression example

```
int x = y * z;
⋮
y = a + b;
⋮
return y * z; y*z UNAVAILABLE
```

Figure 11: unavailable expression example

An expression is *syntactically* available at a node  $n$  if its value gets computed (and not subsequently invalidated) along every path from the entry of the flowgraph to  $n$ .

```
if ((x+1) * (x+1) == y) {
    s = x + y;
}
if (x*x + 2*x + 1 != y) {
    t = x + y;
}
return x + y; x+y AVAILABLE
```

Figure 12:  $x+y$  is semantically available

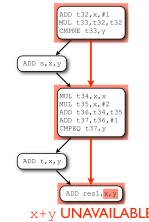


Figure 13:  $x+y$  is syntactically unavailable

On the path in red from Figure 13 through the flowgraph,  $x + y$  is only computed once, so  $x + y$  is syntactically unavailable at the last instruction.

Whereas with live variable analysis we found safety in assuming that more variables were live, here we find safety in assuming that fewer expressions are available. Because if an expression is deemed to be available, we may do something dangerous (e.g. remove an instruction which recomputes its value). So sometimes safe means more, but sometimes means less.



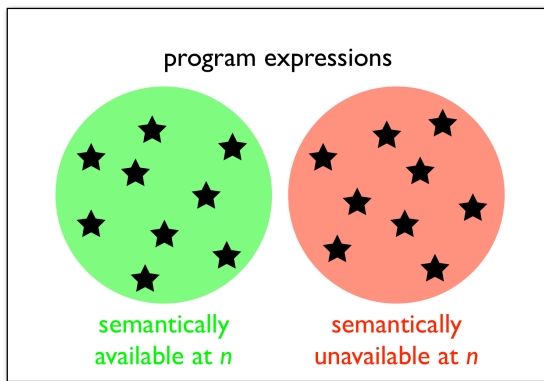


Figure 14: Semantic vs. syntactic

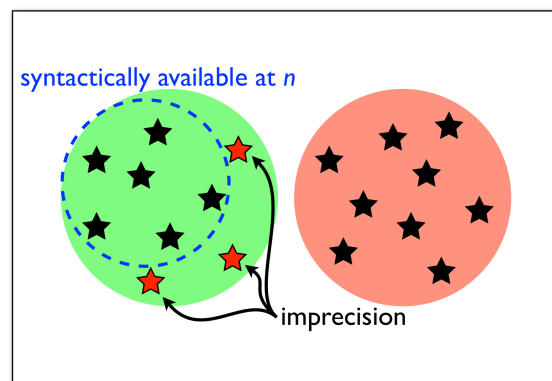


Figure 15: Semantic vs. syntactic

## 6 Foundations of Data Flow Analysis

We saw a lot of examples of data flow analysis, eg. reaching definitions etc. Although there were differences between different types of data flow analysis, they did share number of things in common. Our goal is to develop a general purpose data flow analysis framework.

There are some questions that we want to answer about a framework that performs data flow analysis.

- Correctness: Do we get a correct answer?
- Precision: How good is the answer? <sup>6</sup>
- Coverage: Will the analysis terminate?
- Speed: How fast is the convergence?

### 6.1 A Unified Framework

Data flow problems are defined by

- Domain of values  $V$  (eg, variable names for liveness, the instruction numbers for reaching definitions)
- Meet operator  $V \wedge V \rightarrow V$  to deal with the join nodes.
- Initial value. Once we have defined the meet operator, it will tell us how to initialize all of the non-entry or exits nodes and the boundary conditions for entry and exit nodes.
- A set of transfer functions  $V \rightarrow V$  to define how information flows across basic blocks.

Why we bother to define such a framework?

- First, if meet operator, transfer function and the domains of values are specified in proper way, we will know about correctness, precision and so on.
- From practical engineering perspective, it allows us to reuse code.

### 6.2 Partial Order

A relation  $R$  on a set  $S$  is called a **partial order** if it is

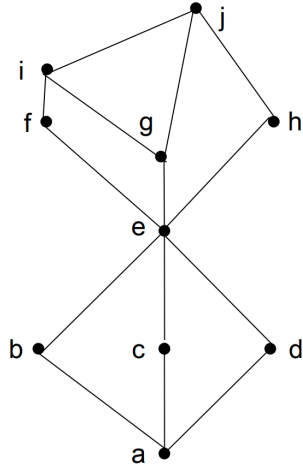
- **Transitivity** if  $x \preceq y$  and  $y \preceq z$  then  $x \preceq z$
- **Antisymmetry** if  $x \preceq y$  and  $y \preceq x$  then  $x = y$
- **Reflexivity**  $x \preceq x$

### 6.3 Lattice

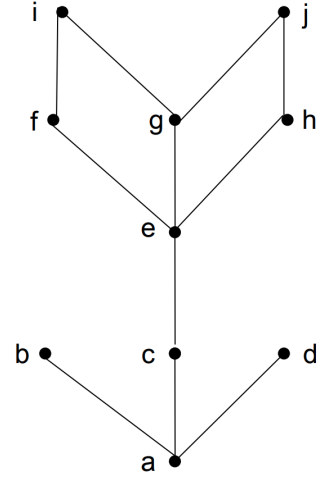
A lattice is a partially ordered set in which every pair of elements has both a least upper bound (lub) and a greatest lower bound (glb).

---

<sup>6</sup>We want a safe solution but as precise as possible.



(a) This is a lattice example.



(b) This is not a lattice example because the pair  $b, c$  does not have a lub.

Figure 16: Two examples

## 6.4 Complete Lattice

A lattice  $A$  is called a complete lattice if every subset  $S$  of  $A$  admits a glb and a lub in  $A$ .

## 6.5 Semi-Lattice

A semilattice (or upper semilattice) is a partially ordered set that has a least upper bound for any nonempty finite subset.

## 6.6 Meet Operator

Meet operator must hold the following properties:

- **commutative:**  $x \wedge y = y \wedge x$ . No ordering in the incoming edges.
- **idempotent:**  $x \wedge x = x$
- **associative :**  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a Top element  $T$  such that  $x \wedge T = x$ . Partly due to the way we initialize everything we need.

Meet Operator defines a partial ordering on values. This is important in ensuring the analysis converges. So what does it mean ?  $x \preceq y$  if and only if  $x \wedge y = x$ . The  $\preceq$  not means less or equal to or subset, but it really means lattice inclusion. So if  $x \preceq y$ , this means  $x$  is more conservative

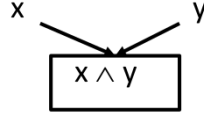


Figure 17: Meet Operator

or constrained. In another word,  $x$  is lattice included in  $y$ . Partial ordering will also lead to some other properties

- **Transitivity** if  $x \preceq y$  and  $y \preceq z$  then  $x \preceq z$
- **Antisymmetry** if  $x \preceq y$  and  $y \preceq x$  then  $x = y$
- **Reflexivity**  $x \preceq x$

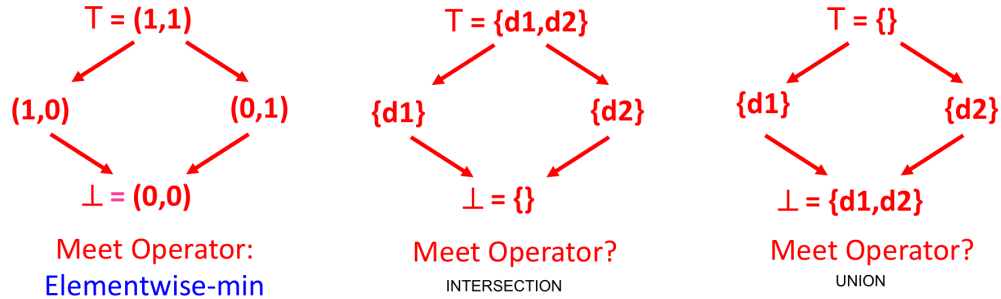


Figure 18: Different meet operator defines different lattice

For our data flow analysis, values and meet operator define a semi-lattice, which means  $\top$  exists, but not necessarily  $\perp$ .

## 6.7 Descending Chain

The height of a lattice is the largest number of  $\succ$  relations that will fit in a descending chain. eg.  $x_0 \succ x_1 \succ x_2 \succ \dots$

So, for reaching definitions, the height is the number of definitions.

Finite descending chain will ensure the convergence. If we don't have a finite descending chain, there is a possibility that the analysis will never terminate. But an infinite lattice still can have a finite descending chain. I want to note that infinite lattice doesn't always mean non-convergence.

So consider the constant propagation, the infinite lattice has a finite descending chain, so this can converge.

## 6.8 Transfer Functions

Transfer function dictates how information propagates across a basic block. So what we need for our transfer function? **First**, it must have an identity function which means there exists an  $f$  such

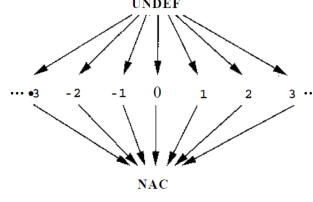


Figure 19: The lattice of constant propogation

that  $f(x) = x$  for all  $x$ . For example, in Reaching Definitions and Liveness, when  $Gen, KILL = \Phi$ , this transfer function satisfies  $f(x) = x$ . **Second**, when we compose transfer functions, it must be consitent with the transfer function. So if  $f_1, f_2 \in F$ , the  $f_1 \cdot f_2 \in F$ .

For example,

$$\begin{aligned}
 f_1(x) &= G_1 \cup (x - K_1) \\
 f_2(x) &= G_2 \cup (x - K_2) \\
 f_2(f_1(x)) &= G_2 \cup [(G_1 \cup (x - K_1)) - K_2] \\
 &= [G_2 \cup (G_1 - K_2)] \cup [x - (K_1 \cup K_2)] \\
 G &= G_2 \cup (G_1 - K_2) \\
 K &= K_1 \cup K_2
 \end{aligned}$$

## 6.9 Monotonicity

A framework  $(F, V, \wedge)$  is monotone if and only if  $x \preceq y$  implies  $f(x) \preceq f(y)$ . This means that a "smaller(more conservative) or equal" input to the same function will always give a "smaller(more conservative) or equal" output.

Alternatively,  $(F, V, \wedge)$  is monotone if and only if  $f(x \wedge y) \preceq f(x) \wedge f(y)$ . So merge input, then apply  $f$  is small(more conservative) or equal to apply the transfer function individually and then merge the result. Values are defined by semi-lattice, the meet operator only ever moves down the lattice from top towards the bottom. So we need to constrain the transfer function.

I will show you a unmonotone example.

Let top be 1 and bottom be 0 and the meet operator is  $\cap$ .  $f(0) = 1, f(1) = 0$

Let's check whether reaching definitions is monotone.

Note that monotone framework does not mean  $f(x) \preceq x$ .

## 6.10 Distributivity

Reaching definitions is distributive but constant propogation is not.

## 7 Introduction to Static Single Assignment

Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the use sites of variables defined there, and a list of pointers to all definition sites of the variables used there. An improvement on the idea of def-use chains is static single-assignment form, or SSA form, an intermediate representation in which each variable has only one definition in the program text. SSA is very useful for many optimizations such as Loop-Invariant Code Motion and Copy Propagation.

### 7.1 Definition-Use and Use-Definition Chains

#### Use-Definition (UD) Chains

For a given definition of a variable X, what are all of its uses?

#### Definition-Use (DU) Chains

For a given use of a variable X, what are all of the reaching definitions of X?

Unfortunately, it is expensive to use UD and DU chains, because if we have N defs, and M uses, the space complexity is  $O(NM)$ . An example is in 20

```
foo(int i, int j) {  
    ...  
    switch (i) {  
    case 0: x=3; break;  
    case 1: x=1; break;  
    case 2: x=6; break;  
    case 3: x=7; break;  
    default: x = 11;  
    }  
    switch (j) {  
    case 0: y=x*7; break;  
    case 1: y=x+4; break;  
    case 2: y=x-2; break;  
    case 3: y=x+1; break;  
    default: y=x+9;  
    }  
}
```

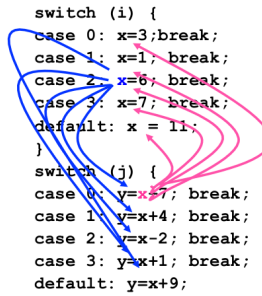


Figure 20: If a variable has N uses and M definitions (which occupy about  $N + M$  instructions in a program), it takes space (and time) proportional to  $N \cdot M$  to represent def-use chains – a quadratic blowup.

## 7.2 Static Single Assignment(SSA)

### Static Single Assignment

Static Single Assignment is an IR where every variable is assigned a value at most once in the program text.

### the $\Phi$ function

$\Phi$  merges multiple definitions along multiple control paths into a single definition. At a basic block with  $p$  predecessors, there are  $p$  arguments to the  $\Phi$  functions.

$$x_{\text{new}} \leftarrow \Phi(x_1, x_1, x_1, \dots, x_p)$$

### 7.2.1 Why SSA is useful?

**Useful for Dataflow Analysis** Dataflow analysis and optimization algorithms can be made simpler when each variable has only one definition.

**Less space and time complexity** If a variable has  $N$  uses and  $M$  definitions (which occupy about  $N + M$  instructions in a program), it takes space (and time) proportional to  $N \cdot M$  to represent def-use chains – a quadratic blowup. For almost all realistic programs, the size of the SSA form is linear in the size of the original program.

**Simplify some algorithms** Uses and defs of variables in SSA form relate in a useful way to the dominator structure of the control-flow graph, which simplifies algorithms such as interference-graph construction.

**Eliminate needless relationship** Unrelated uses of the same variable in the source program become different variables in SSA form, eliminating needless relationships shown in 11.

```
1 for i <- 1 to N do A[i] <- 0
3 for i <- 1 to M do s <- s + B[i]
```

Listing 11: An example

## 7.3 How to represent SSA?

In straight-line code, such as within a basic block, it is easy to see that each instruction can define a fresh new variable instead of redefining an old one shown in 21

But when two control-flow paths merge together, it is not obvious how to have only one assignment for each variable. To solve this problem we introduce a notational fiction, called a  $\Phi$  function. Figure 22 shows that we can combine  $a1$  (defined in block 1) and  $a2$  (defined in block 3) using the function  $a3 \leftarrow \Phi(a1, a2)$ .

unlike ordinary mathematical functions,  $\Phi(a1, a2)$  yields  $a1$  if control reaches block 4 along the edge  $2 \rightarrow 4$ , and yields  $a2$  if control comes in on edge  $3 \rightarrow 4$ .

$a \leftarrow x + y$	$a_1 \leftarrow x + y$
$b \leftarrow a - 1$	$b_1 \leftarrow a_1 - 1$
$a \leftarrow y + b$	$a_2 \leftarrow y + b_1$
$b \leftarrow x \cdot 4$	$b_2 \leftarrow x \cdot 4$
$a \leftarrow a + b$	$a_3 \leftarrow a_2 + b_2$

(a) A straight-line program. (b) The program in single-assignment form.

Figure 21: SSA for straight-line code

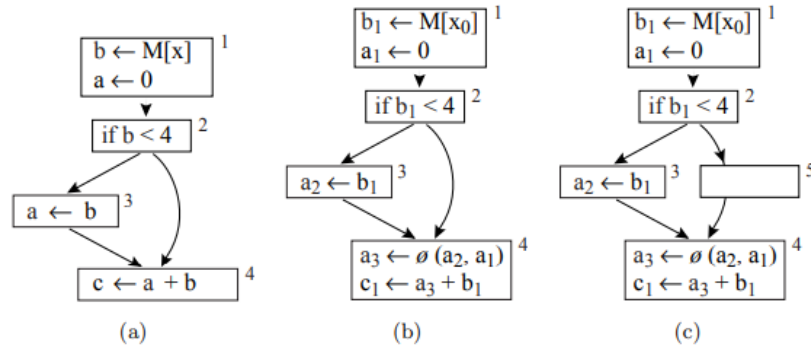


Figure 22: (a) A program with a control-flow join; (b) the program transformed to single-assignment form; (c) edge-split SSA form.

### 7.3.1 How does the $\phi$ -function know which edge was taken?

If we must execute the program, or translate it to executable form, we can “implement” the  $\Phi$ -function using a move instruction on each incoming edge as shown in Figure 23. However, in many cases, we simply need the connection of uses to definitions and don’t need to “execute” the  $\Phi$ -functions during optimization. In these cases, we can ignore the question of which value to produce.

## 7.4 Converting to SSA form

The algorithm for converting a program to SSA form is roughly as follows:

- 1. adds  $\Phi$  functions for the variables, and then
- 2. renames all the definitions and uses of variables using subscripts.

### 7.4.1 Trivial SSA

Trivial SSA form is based on a simple observation:  $\Phi$  functions are only needed for variables that are “live” after the  $\Phi$  function.



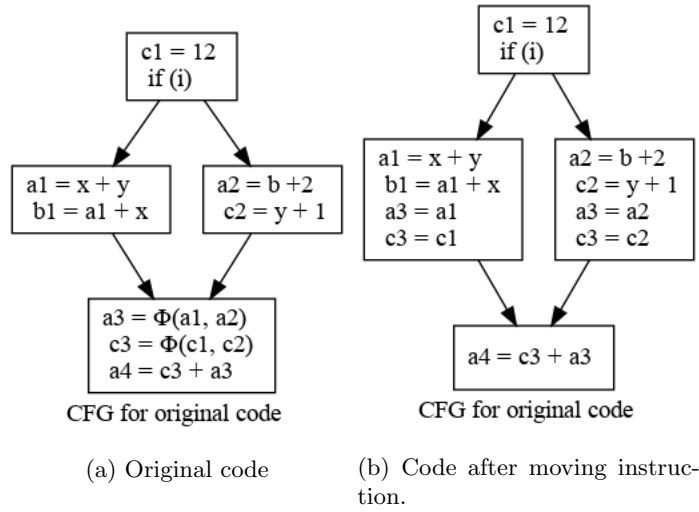


Figure 23: Implementing  $\Phi$ -function

- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  for all live variables.

Trivial SSA will generate some useless  $\Phi$  functions. An example is shown in Figure 24 So a  $\Phi$ -function is not needed for every variable at each point.

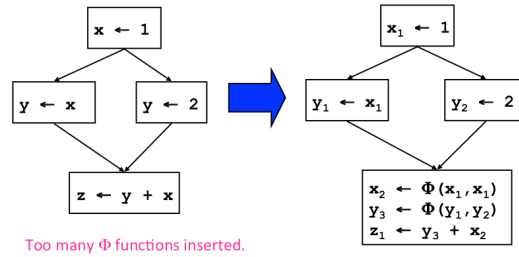


Figure 24:  $x_2 \leftarrow \Phi(x_1, x_1)$  is useless because  $x_2$  is equal to  $x_1$ .

### 7.4.2 Minimal SSA

Minimal SSA is an updated version compared to trivial SSA.

- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  for all live variables with multiple outstanding defs.

### 7.4.3 Path-convergence criterion

There should be a  $\Phi$ -function for variable  $a$  at node  $z$  of the flow graph exactly when all of the following are true:

- 1. There is a block  $x$  containing a definition of  $a$ ,
- 2. There is a block  $y$  (with  $y \neq x$ ) containing a definition of  $a$ ,
- 3. There is a nonempty path  $P_{xz}$  of edges from  $x$  to  $z$ ,
- 4. There is a nonempty path  $P_{yz}$  of edges from  $y$  to  $z$ ,
- 5. Paths  $P_{xz}$  and  $P_{yz}$  do not have any node in common other than  $z$ , and
- 6. The node  $z$  does not appear within both  $P_{xz}$  and  $P_{yz}$  prior to the end, though it may appear in one or the other.

We consider the start node to contain an implicit definition of every variable, either because the variable may be a formal parameter or to represent the notion of  $a \leftarrow$  uninitialized without special cases. A  $\Phi$ -function itself counts as a definition of  $a$ , so the path-convergence criterion must be considered as a set of equations to be satisfied. As usual, we can solve them by iteration as shown in 3.

---

**Algorithm 3** Iterated path-convergence criterion

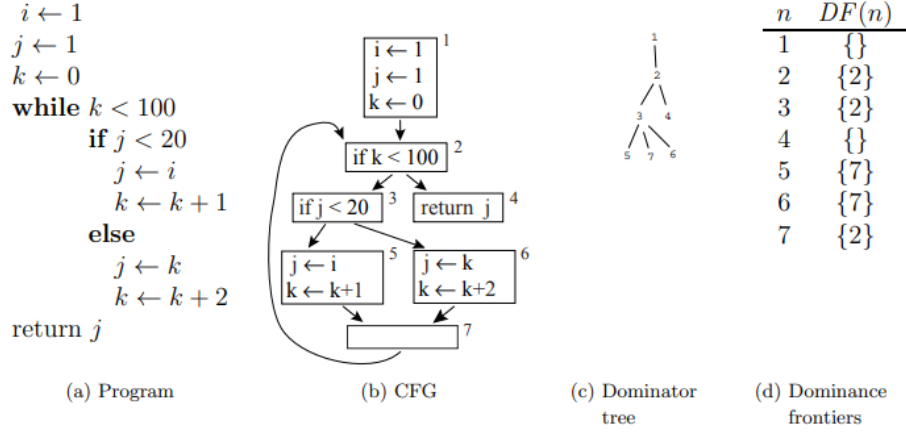
---

```
while there are nodes  $x, y, z$  satisfying conditions 1–5 and  
   $z$  does not contain a  $\Phi$ -function for  $a$  do  
    insert  $a \leftarrow \Phi(a, a, \dots, a)$  at node  $z$   
end while
```

---

### 7.4.4 Dominance property of SSA form

The iterated path-convergence algorithm for placing  $\Phi$ -functions is not practical, since it would be very costly to examine every triple of nodes  $x, y, z$ , and every path leading from  $x$  and  $y$ . A much more efficient algorithm using the dominator tree of the flow graph as shown in Figure 25.



Variable  $j$  defined in node 1, but  $DF(1)$  is empty. Variable  $j$  defined in node 5,  $DF(5)$  contains 7, so node 7 needs  $\phi(j, j)$ . Now  $j$  is defined in 7 (by a  $\phi$ -function),  $DF(7)$  contains 2, so node 2 needs  $\phi(j, j)$ .  $DF(6)$  contains 7, so node 7 needs  $\phi(j, j)$  (but already has it).  $DF(2)$  contains 2, so node 2 needs  $\phi(j, j)$  (but already has it). Similar calculation for  $k$ . Variable  $i$  defined in node 1,  $DF(1)$  is empty, so no  $\phi$ -functions necessary for  $i$ .

(e) Insertion criteria for  $\phi$ -functions

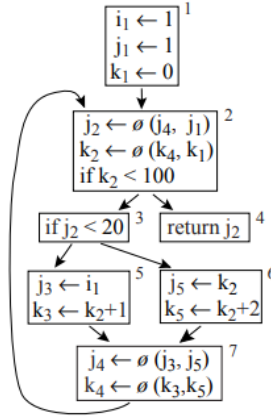
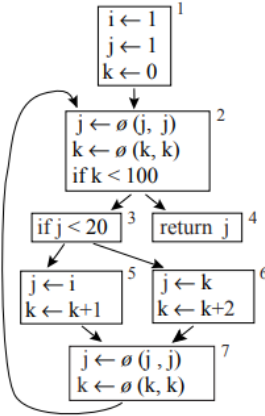


Figure 25: Conversion of a program to static single-assignment form. Node 7 is a postbody node, inserted to make sure there is only one loop edge; such nodes are not strictly necessary but are sometimes helpful.

### Strictly dominance

$x$  strictly dominates  $w$  ( $x$  sdom  $w$ ) iff impossible to reach  $w$  without passing through  $x$  first.

### Dominance

$x$  dominates  $w$  ( $x \text{ dom } w$ ) iff  $x \text{ sdom } w$  or  $x = w$ .

$$\text{Dom}(n) = \begin{cases} \{n\} & \text{if } n = n_0 \\ \{n\} \cup \left( \bigcap_{p \in \text{preds}(n)} \text{Dom}(p) \right) & \text{if } n \neq n_0 \end{cases}$$

### Dominance tree

$x \text{ sdom } w$  iff  $x$  is a proper ancestor of  $w$ .

### Dominance Frontier

The dominance frontier of a node  $x$  is the set of all nodes  $w$  such that  $x$  dominates a predecessor of  $w$ , but does not strictly dominate  $w$ .

$$F(x) = \{w \mid x \text{ dom pred}(w) \text{ AND } !(x \text{ sdom } w) \}$$

An essential property of static single assignment form is that definitions dominate uses; more specifically,

- If  $x$  is the  $i$ th argument of a  $\Phi$ -function in block  $n$ , then the definition of  $x$  dominates the  $i$ th predecessor of  $n$ .
- If  $x$  is used in a non- $\Phi$  statement in block  $n$ , then the definition of  $x$  dominates  $n$

### Dominance Property of SSA

In SSA,

- If  $x$  is used in  $x \leftarrow \Phi(\dots, x_i, \dots)$ , then  $BB(x_i)$  dominates  $i$ th predecessor of  $BB(\Phi)$
- If  $x$  is used in  $y \leftarrow \dots x \dots$ , then  $BB(x)$  dominates  $BB(y)$

***Dominance frontier criterion.*** Whenever node  $x$  contains a definition of some variable  $a$ , then any node  $z$  in the dominance frontier of  $x$  needs a  $\Phi$ -function for  $a$ .

***Iterated dominance frontier.*** Since a  $\Phi$ -function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need  $\Phi$ -functions.

***Theorem.*** The iterated dominance frontier criterion and the iterated path convergence criterion specify exactly the same set of nodes at which to put  $\Phi$ -functions

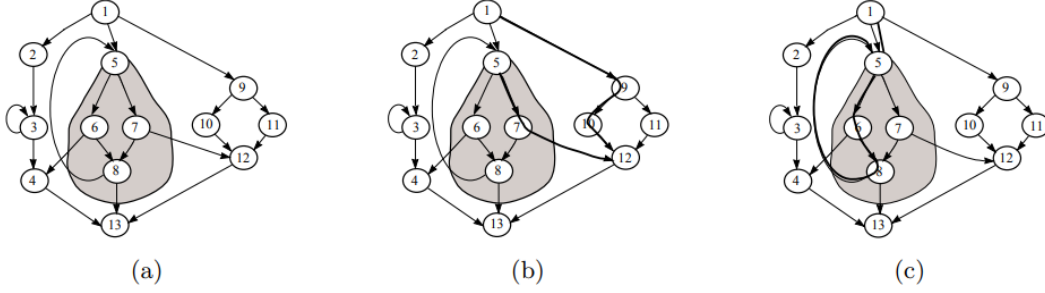


Figure 26: Node 5 dominates all the nodes in the grey area. (a) Dominance frontier of node 5 includes the nodes (4, 5, 12, 13) that are targets of edges crossing from the region dominated by 5 (grey area including node 5) to the region not strictly dominated by 5 (white area including node 5). (b) Any node in the dominance frontier of  $n$  is also a point of convergence of nonintersecting paths, one from  $n$  and one from the root node. (c) Another example of converging paths  $P_{1,5}$  and  $P_{5,5}$ .

#### Proof

The sketch of a proof that shows if  $w$  is in the dominance frontier of a definition, then it must be a point of convergence.

Suppose there is a definition of variable  $a$  at some node  $n$  (such as node 5 in Figure 26b), and node  $w$  (such as node 12 in Figure 26b) is in the dominance frontier of  $n$ . The root node implicitly contains a definition of every variable, including  $a$ . There is a path  $P_{rw}$  from the root node (node 1 in Figure 26) to  $w$  that does not go through  $n$  or through any node that  $n$  dominates; and there is a path  $P_{nw}$  from  $n$  to  $w$  that goes only through dominated nodes. These paths have  $w$  as their first point of convergence.

## 7.5 Computing the dominance frontier

To insert all the necessary  $\Phi$ -functions, for every node  $n$  in the flow graph we need  $DF[n]$ , its dominance frontier. Given the dominator tree, we can efficiently compute the dominance frontiers of all the nodes of the flow graph in one pass. We define two auxiliary sets

- $DF_{local}[n]$  The successors of  $n$  that are not strictly dominated by  $n$ ;
- $DF_{up}[n]$  Nodes in the dominance frontier of  $n$  that are not dominated by  $n$ 's immediate dominator.

The dominance frontier of  $n$  can be computed from  $DF_{local}[n]$  and  $DF_{up}[n]$

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in \text{children}[n]} DF_{up}[c]$$

where  $\text{children}[n]$  are the nodes whose immediate dominator ( $\text{idom}$ ) is  $n$ .

To compute  $DF_{\text{local}}[n]$  4 more easily (using immediate dominators instead of dominators), we use the following theorem:  $DF_{\text{local}}[n]$  = the set of those successors of  $n$  whose immediate dominator is not  $n$ . The following `computeDF` function should be called on the root of the dominator tree (the start node of the flow graph). It walks the tree computing  $DF[n]$  for every node  $n$ : it computes  $DF_{\text{local}}[n]$  by examining the successors of  $n$ , then combines  $DF_{\text{local}}[n]$  and (for each child  $c$ )  $DF_{\text{up}}[n].a$

---

**Algorithm 4** `computeDF`

---

```

 $S \leftarrow \{\}$ 
for each node  $y$  in  $\text{succ}[n]$  do                                      $\triangleright$  This loop computes  $DF_{\text{local}}[n]$ 
    if  $\text{idom}(y) \neq n$  then
         $S \leftarrow S \cup \{y\}$ 
    end if
end for
for each child  $c$  of  $n$  in the dominator tree do
    computeDF $[c]$ 
    for each element  $w$  of  $DF[c]$  do                                      $\triangleright$  This loop computes  $DF_{\text{up}}[n]$ 
        if  $n$  does not dominate  $w$  then
             $S \leftarrow S \cup \{w\}$ 
        end if
    end for
end for

```

---

This algorithm is quite efficient. It does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. Although there are pathological graphs in which most of the nodes have very large dominance frontiers, in most cases the total size of all the DFs is approximately linear in the size of the graph, so this algorithm runs in “practically” linear time.

## 7.6 Inserting $\Phi$ -functions

Starting with a program not in SSA form, we need to insert just enough  $\Phi$ -functions to satisfy the iterated dominance frontier criterion. To avoid re-examining nodes where no  $\Phi$ -function has been inserted, we use a work-list algorithm.

Algorithm 5 starts with a set  $V$  of variables, a graph  $G$  of controlflow nodes – each node is a basic block of statements – and for each node  $n$  a set  $A_{\text{orig}}[n]$  of variables defined in node  $n$ . The algorithm computes  $A_{\Phi}[a]$ , the set of nodes that must have  $\Phi$ -functions for variable  $a$ . Sometimes a node may contain both an ordinary definition and a  $\Phi$ -function for the same variable; for example, in Figure 26b,  $a \in A_{\text{orig}}[2]$  and  $2 \in A_{\Phi}[a]$ .

This algorithm does a constant amount of work (a) for each node and edge in the control-flow graph, (b) for each statement in the program, (c) for each element of every dominance frontier, and (d) for each inserted  $\Phi$ -function. For a program of size  $N$ , the amounts  $a$  and  $b$  are proportional to  $N$ ,  $c$  is usually approximately linear in  $N$ . The number of inserted  $\Phi$ -functions (d) could be  $N^2$  in the worst case, but empirical measurement has shown that it is usually proportional to  $N$ . So in practice, Algorithm 5 runs in approximately linear time.

---

**Algorithm 5** Place- $\Phi$ -Functions

---

```
for each node n do
  for each variable a in  $A_{orig}[n]$  do
    defsites[a]  $\leftarrow$  defsites[a]  $\cup \{n\}$ 
  end for
end for
for each variable a do
  W  $\leftarrow$  defsites[a]
  while W not empty do
    remove some node n from W
    for each y in DF[n] do
      if  $y \notin A_{\Phi}[a]$  then
        insert the statement  $a \leftarrow \Phi(a, a, \dots, a)$  at the top of block y, where the  $\Phi$ -function
        has as many arguments as y has predecessors
         $A_{\Phi}[a] \leftarrow A_{\Phi}[a] \cup \{y\}$ 
        if  $a \notin A_{orig}[y]$  then
          W  $\leftarrow$  W  $\cup \{y\}$ 
        end if
      end if
    end for
  end while
end for
```

---

## 7.7 Renaming the variables

After the  $\Phi$ -functions are placed, we can walk the dominator tree, renaming the different definitions (including  $\Phi$ -functions) of variable  $a$  to  $a_1$ ,  $a_2$ ,  $a_3$  and so on. Rename each use of  $a$  to use the closest definition  $d$  of  $a$  that is above  $a$  in the dominator tree. Algorithm renames all uses and definitions of variables, after the  $\Phi$ -functions have been inserted by Algorithm 6. In traversing the dominator tree, the algorithm “remembers” for each variable the most recently defined version of each variable, on a separate stack for each variable. Although the algorithm follows the structure of the dominator tree – not the flow graph – at each node in the tree it examines all outgoing flow edges, to see if there are any  $\Phi$ -functions whose operands need to be properly numbered.

## 7.8 Edge Splitting

Some analyses and transformations including reverse transformation from SSA back into a normal form are simpler if there is never a controlflow edge that leads from a node with multiple successors to a node with multiple predecessors. To give the graph this unique successor or predecessor property, we perform the following transformation: For each control-flow edge  $a \leftarrow b$  such that  $a$  has more than one successor and  $b$  has more than one predecessor, we create a new, empty controlflow node  $z$ , and replace the  $a \leftarrow b$  edge with an  $a \leftarrow z$  edge and a  $z \leftarrow b$  edge.

An SSA graph with this property is in edge-split SSA form. Figure 22 illustrates edge splitting. Edge splitting may be done before or after insertion of  $\Phi$ -functions.

---

**Algorithm 6** Renaming variables.

---

```
Initialization:
for each variable  $a$  do
    Count[ $a$ ]  $\leftarrow$  0
    Stack[ $a$ ]  $\leftarrow$  empty
    push 0 onto Stack[ $a$ ]
end for
Rename( $n$ )
for each statement  $S$  in block  $n$  do
    if  $S$  is not a  $\Phi$ -function then
        for each use of some variable  $x$  in  $S$  do
             $i \leftarrow \text{top}(\text{Stack}[x])$ 
            replace the use of  $x$  with  $x_i$  in  $S$ 
        end for
    end if
    for each definition of some variable  $a$  in  $S$  do
        Count[ $a$ ]  $\leftarrow$  Count[ $a$ ]+1
         $i \leftarrow$  Count[ $a$ ]
        push  $i$  onto Stack[ $a$ ]
        replace definition of  $a$  with definition of  $a_i$  in  $S$ 
    end for
end for
for each successor  $Y$  of block  $n$ , do
    Suppose  $n$  is the  $j$ th predecessor of  $Y$ 
    for each  $\Phi$ -function in  $Y$  do
        suppose the  $j$ th operand of the  $\Phi$ -function is  $a$ 
         $i \leftarrow \text{top}(\text{Stack}[a])$ 
        replace the  $j$ th operand with  $a_i$ 
    end for
end for
for each child  $X$  of  $n$  do
    Rename( $X$ )
end for
for each definition of some variable  $a$  in the original  $S$  do
    pop Stack[ $a$ ]
end for
```

---



## 8 SSA-Style optimizations

### 8.1 Constant Propagation

#### notes

- If  $v \leftarrow c$ , replace all uses of  $v$  with  $c$
- If  $v \leftarrow \Phi(c, c, c)$  (each input is the same constant), replace all uses of  $v$  with  $c$

---

#### Algorithm 7 SSA-CP

---

```
W ← list of all defs
while !W.isEmpty do
  Stmt S ← W.removeOne
  if (S has form  $v \leftarrow c$ ) or (S has form  $v \leftarrow \Phi(c, \dots, c)$ ) then
    delete S
    for each stmt U that uses  $v$  do
      replace  $v$  with  $c$  in U
      W.add(U)
    end for
  end if
end while
```

---

### 8.2 Conditional Constant Propagation

Wegman and Zadeck's Sparse Conditional Constant (SCC) algorithm was used to find constant expressions, constant conditions, and unreachable code [WZ91]. The output of the SCC algorithm is an association of variables to one of  $\{\perp, c, \top\}$ , where  $\perp$  marks a variable that can hold different values at different times, and  $\top$  means the variable is not executed. In addition, every flow-graph node (corresponding to a quadruple) is marked as executable or non-executable. We then walk the flow-graph, eliminating dead-code (quadruples marked non-executable), replacing constant variables with their values, and changing constant conditional branches to goto statements.

#### notes

- Assume all blocks unexecuted until proven otherwise
- Assume all variables are not executed (only with proof of assignment of a non-constant value do we assume not constant)

### 8.2.1 Example

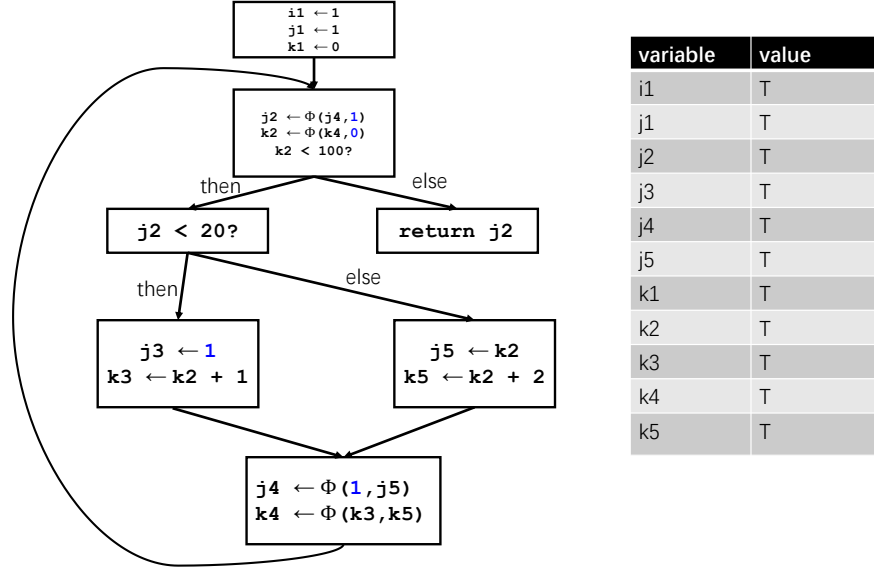


Figure 27: Original code. The black block is marked as unexecuted

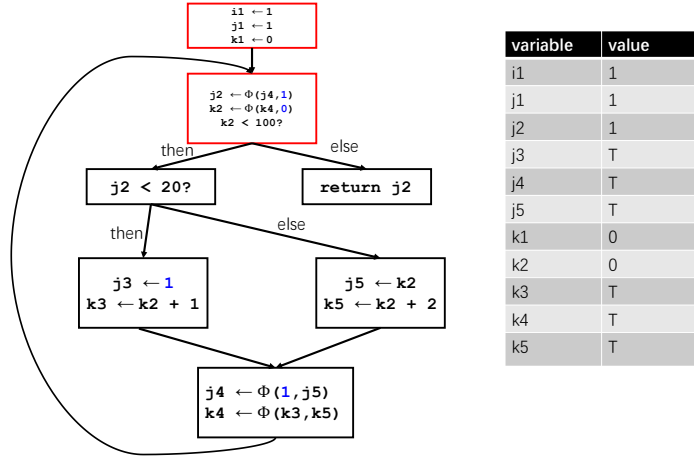


Figure 28: The read block is marked as executed. After walking the first two blocks, the value is shown above.

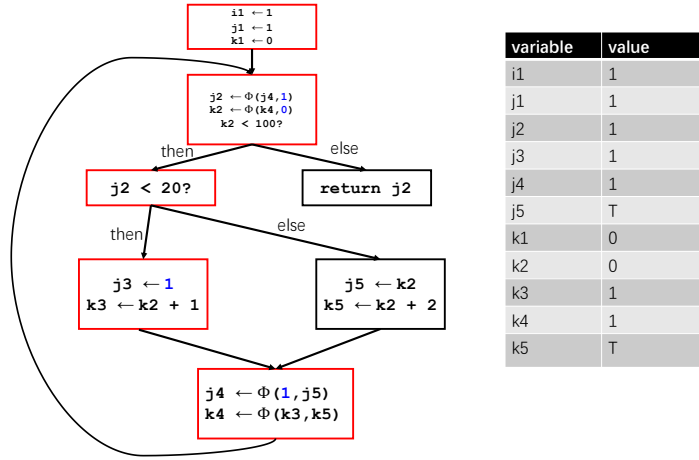


Figure 29: After walking 5 blocks.

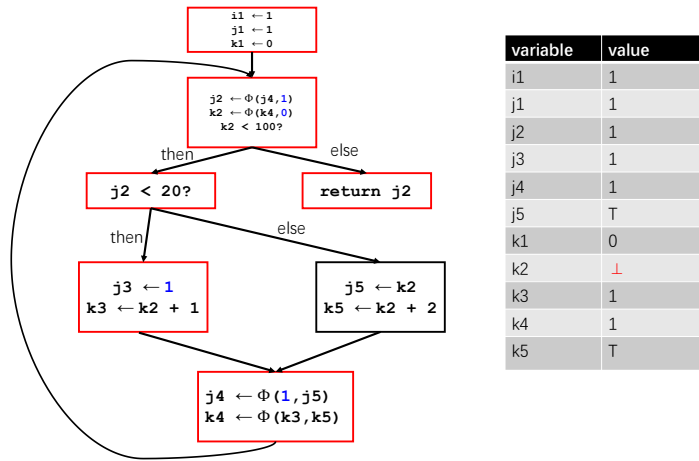


Figure 30: Now  $k2$  is  $\perp$ , so the `return j2` is reachable.

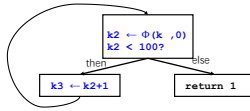


Figure 31: Code after applied SCC.

### 8.3 Copy Propagation

#### notes

- delete  $x \leftarrow \Phi(y, y, y)$  and replace all  $x$  with  $y$
- delete  $x \leftarrow y$  and replace all  $x$  with  $y$

### References

- [1] Easwaran Raman and Xinliang David Li. “Learning Branch Probabilities in Compiler from Datacenter Workloads”. In: *arXiv preprint arXiv:2202.06728* (2022).