Data-Flow Analysis and Second Assignment

Andrea Lavino

March 2025

Contents

1 Data-Flow Analysis

Compiler optimizations rely heavily on the analysis of intermediate representation code. These analysis can be categorized into three levels: (1) local, (2) global and (3) interprocedural.

Local analysis focuses on a single basic block and its instruction, whereas global analysis evaluates the entire control flow graph, usually related to a single procedure. Finally, interprocedural analysis gathers information across multiple procedures to enable broader optimizations.

Data-flow analysis is a collection of techniques, within the global analysis category, used to extract information about basic blocks and their effects, with the purpose of providing valuable information to the compiler optimizer. Without these information, the compiler would not be able to perform complex optimizations efficiently.

Before diving into some examples of the data-flow analysis problems, it is crucial to understand what can be the effects of instructions and basic blocks and to understand the common framework for data-flow analysis problems.

1.1 Instruction Effects

Every instruction of the type $a = b + c$ can have three effects:

- Uses: the instruction uses the variables b and c in order to compute the value of a
- Kills: the instruction kills a previous definition of a
- Defines: the instruction defines the variable a

The effects of every instruction compound creating the effect of the whole basic block

1.2 Basic-Block Effects

As the instructions, even a basic block can have three effects:

- locally exposed use: the basic block uses a variable that it is not preivously defined within the BB, therefore uses a variable that was defined before the basic block entry point
- kills: every variable definition within the BB kills all the other definitions that have reached the basic block
- locally available definition: the basic block uses a variable that was previously defined in the same block

1.2.1 Example

Here there is a small example of the effects that a basic block can have.

```
t1 = r1 + r2
r2 = t1
t2 = r2 + r1
r1 = t2
t3 = r1 * r1
r2 = t3
if r2 > 100 goto L1
```

Locally exposed use: `r1, r2`
Killed definitions: `r1, r2`
Locally available definitions:
`t1, t2, t3, r1, r2`

1.3 Data-Flow analysis problem structure

Each technique that lies in the domain of data-flow analysis aims at solving a specific problem regarding gaining a certain knowledge about the code. Therefore, it is fundamental to clearly state our goal and define what we are trying to find about the program.

After having properly defined the problem, we need to define its transfer functions, i.e. the function that maps basic blocks inputs to its outputs and viceversa, depending on the problem type (forward or backward).

Besides the transfer function, other important elements that helps properly define the problem are the boundary and initial conditions of each basic block, the direction of the data flow (forward or backward) and the meet operator.

All of the problem informations can be summarized using a simple table representation, like the following:

	Dataflow Problem X
Domain	?
Direction	? ? ?
Transfer function	?
Meet Operation ($\wedge$)	?
Boundary Condition	?
Initial interior points	?

Table 1: Generic data-flow problem table

2 Reaching Definitions

The problem of reaching definitions consists in finding all the available definitions in a specific point of the program. The information obtained by solving this problem can be used to check whether a variable is used before its definition.

2.1 Problem formalization

Formally we say that a definition d **reaches** a point p if there is a path from d to p where d is not killed along the path.

	Reaching definitions problem
Domain	Sets of definitions
Direction	Forward $out[b] = f_b(in[b])$ $in[b] = \wedge out[pred[b]]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cup$
Boundary Condition	$out[entry] = \varnothing$
Initial interior points	$out[b] = \varnothing$

Table 2: Reaching Definition table summary

2.2 Example

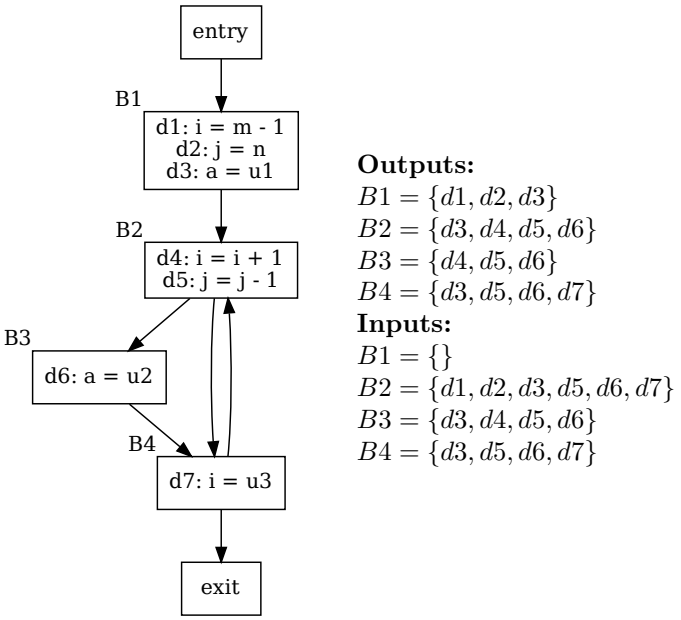


Figure 1: Reaching definitions problem example

3 Liveness Analysis

The problem of liveness analysis aims at finding which variables are *alive* at a specific point of the program. It is primarily used to identify uninitialized variables throughtout the program or to optimally use the registers.

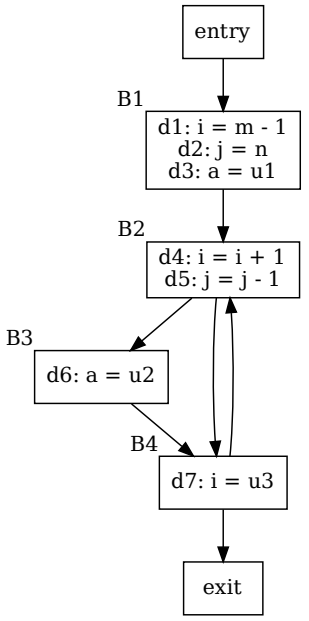
3.1 Problem formalization

Formally we say that a variable v is **alive** at a point p if the value of v is used along some path from p to the exit block, otherwise it is considered **dead**.

	Liveness of variables
Domain	Sets of variables
Direction	Backward $in[b] = f_b(out[b])$ $out[b] = \wedge in[succ(b)]$
Transfer function	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation ($\wedge$)	$\cup$
Boundary Condition	$in[exit] = \varnothing$
Initial interior points	$in[b] = \varnothing$

Table 3: Liveness of variables summary table

3.2 Example



Outputs:
$B1 = \{m, n, u1, u2, u3\}$
$B2 = \{j, u2, u3\}$
$B3 = \{j, u2, u3\}$
$B4 = \{i, j, u2, u3\}$
Inputs:
$B1 = \{m, n, u1, u2, u3\}$
$B2 = \{i, j, u2, u3\}$
$B3 = \{u2, u3\}$
$B4 = \{u3\}$

Figure 2: Your new caption here

4 Available Expressions

The data-flow problem of available expressions tries to find what are the expressions already computed in a certain point of the program. It is used in order to

know if an expression that was computed previously can be reused or not and it is fundamental for Global Common Subexpression Elimination optimization.

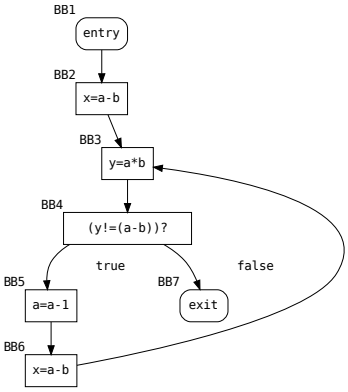
4.1 Problem formalization

Formally, an expression is defined as **available** if it has been computed along every path from the entry block to the point p. Moreover, we say that a block *generates* an expression $x \oplus y$ if it evaluates the expression without subsequently redefining x or y. Otherwise, we say that a block *kills* an expression when it assign a new value either to x or y and do not recomputes the expression $x \oplus y$.

	Dataflow Problem X
Domain	Sets of expressions
Direction	Forward $out[b] = f_b(in[b])$ $in[b] = \wedge out[pred(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cap$
Boundary Condition	$out[entry] = \varnothing$
Initial interior points	$out[b] = U$

Table 4: Dataflow Problem X Properties

4.2 Example



Outputs:
$B1 = \varnothing$
$B2 = \{a - b\}$
$B3 = \{a - b, a * b\}$
$B4 = \{a - b\}$
$B5 = \varnothing$
$B6 = \{a - b\}$
$B7 = \{a - b, a * b\}$
Inputs:
$B1 = \varnothing$
$B2 = \varnothing$
$B3 = \{a - b\}$
$B4 = \{a - b, a * b\}$
$B5 = \{a - b, a * b\}$
$B6 = \varnothing$
$B7 = \{a - b, a * b\}$

Figure 3: Reaching definitions problem example

5 Very Busy Expressions

The search for very busy expressions can be useful for code hoisting, since very busy expressions can be moved from the place they are up to a joint point from which the flow departures.

5.1 Problem formalization

Formally an expression is said to be **very busy** when it is computed along each path that part from the point p without any redefinition of its operands. This information can be used to move the expression to a point of the code in which is computation can be used by all the paths that use the expression.

	Very Busy Expressions
Domain	Sets of expressions
Direction	Backward $in[b] = f_b(out[b])$ $out[b] = \wedge in[succ(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cap$
Boundary Condition	$out[exit] = \varnothing$
Initial interior points	$out[b] = U$

Table 5: Very busy expressions summary table

5.2 Example

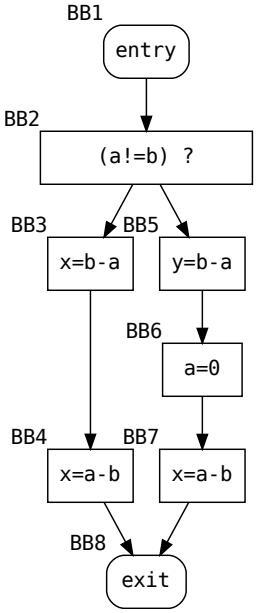


Figure 4: Very Busy Expression Example

	Iteration 1		**Iteration 2**		**Iteration 3**	
	IN[B]	**OUT[B]**	**IN[B]**	**OUT[B]**	**IN[B]**	**OUT[B]**
BB1	$\{b-a\}$	$\{b-a\}$				
BB2	$\{b-a\}$	$\{b-a\}$				
BB3	$\{b-a\}$	$\{a-b\}$				
BB4	$\{a-b\}$	$\varnothing$				
BB5	$\{b-a\}$	$\varnothing$				
BB6	$\varnothing$	$\{a-b\}$				
BB7	$\{a-b\}$	$\varnothing$				
BB8	$\varnothing$	$\varnothing$				

Table 6: Very Busy Expression Algorithm Execution Table

6 Dominator Analysis

Dominator analysis is fundamental to create the single static assignment form.

6.1 Problem formalization

A basic block B_1 **dominates** another block B_2 if it is encountered in every path from entry to B_2.

	Dominator Analysis
Domain	Sets of Basic Blocks
Direction	Forward
Transfer function	$f_b(x) = \{x\} \cup (\bigcap_{m \in preds(x)} f_b(m))$
Meet Operation ($\wedge$)	$\cap$
Boundary Condition	$Dom[entry] = \{entry\}$
Initial interior points	$Dom[b] = N \quad \forall b \neq entry$, with N the number of basic blocks of the CFG

Table 7: Dataflow Problem X Properties

6.2 Example

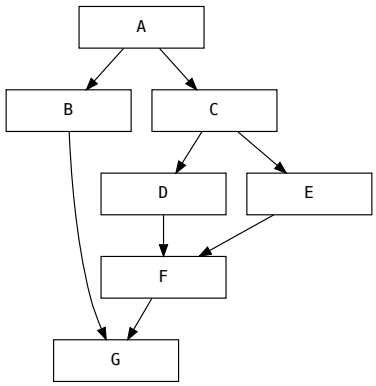


Figure 5: Dominance Analysis example

	DOM[B]
A	$\{A\}$
B	$\{A, B\}$
C	$\{A, C\}$
D	$\{A, C, D\}$
E	$\{A, C, E\}$
F	$\{A, C, F\}$
G	$\{A\}$

Table 8: Dominance analysis execution table

7 Constant Propagation

The constant propagation problem aims at finding what are the couples $< variable, constant\ value >$ that are availables in a certain basic block, so that the variable constant value can be propagated across the blocks.

7.1 Problem formalization

We say that a couple $< variable, constant >$ is valid at block n if it is guaranteed that the variable x gets that constant value every time that the block is reached.

	Constant Propagation
Domain	Sets of variables and their constant values
Direction	Forward $in[b] = \wedge(out[pred(b)])$ $out[b] = f_b(in[b])$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cap$
Boundary Condition	$out[entry] = \varnothing$
Initial interior points	$out[b] = \varnothing$

Table 9: Constant Propagation Problem Summary Table

7.2 Example

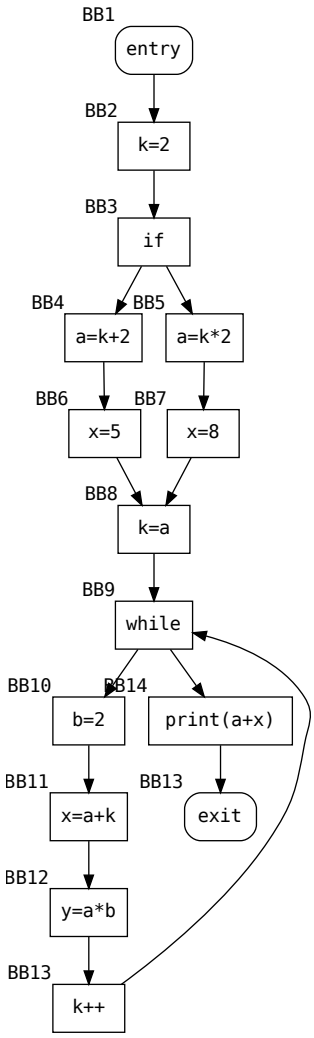


Figure 6: Constant Propagation example

	Iteration 1	
	IN[B]	**OUT[B]**
BB1	$\varnothing$	$< k, 2 >$
BB2	$\varnothing$	$< k, 2 >$
BB3	$< k, 2 >$	$< k, 2 >$
BB4	$< k, 2 >$	$< k, 2 >, < a, 4 >$
BB5	$< k, 2 >$	$< k, 2 >, < a, 4 >$
BB6	$< k, 2 >, < a, 4 >$	$< k, 2 >, < a, 4 >, < x, 5 >$
BB7	$< k, 2 >, < a, 4 >$	$< k, 2 >, < a, 4 >, < x, 8 >$
BB8	$< k, 2 >, < a, 4 >$	$< k, 4 >, < a, 4 >$
BB9	$< k, 2 >, < a, 4 >$	$< k, 2 >, < a, 4 >$
BB10	$< k, 4 >, < a, 4 >$	$< k, 2 >, < a, 4 >, < b, 2 >$
BB11	$< k, 4 >, < a, 4 >, < b, 2 >$	$< k, 4 >, < a, 4 >, < b, 2 >, < x, 8 >$
BB12	$< k, 4 >, < a, 4 >, < b, 2 >, < x, 8 >$	$< k, 2 >, < a, 4 >, < b, 2 >, < x, 8 >, < y, 8 >$
BB13	$< k, 4 >, < a, 4 >, < b, 2 >, < x, 8 >, < y, 8 >$	$< k, 5 >, < a, 4 >, < b, 2 >, < x, 8 >, < y, 8 >$
BB14	$< k, 4 >, < a, 4 >$	$< k, 4 >, < a, 4 >$
BB15	$< k, 4 >, < a, 4 >$	$< k, 4 >, < a, 4 >$

Table 10: Very Busy Expression Algorithm Execution Table

	Iteration 1	
	IN[B]	**OUT[B]**
BB1	$\varnothing$	$< k, 2 >$
BB2	$\varnothing$	$< k, 2 >$
BB3	$< k, 2 >$	$< k, 2 >$
BB4	$< k, 2 >$	$< k, 2 >, < a, 4 >$
BB5	$< k, 2 >$	$< k, 2 >, < a, 4 >$
BB6	$< k, 2 >, < a, 4 >$	$< k, 2 >, < a, 4 >, < x, 5 >$
BB7	$< k, 2 >, < a, 4 >$	$< k, 2 >, < a, 4 >, < x, 8 >$
BB8	$< k, 2 >, < a, 4 >$	$< k, 4 >, < a, 4 >$
BB9	$< a, 4 >$	$< a, 4 >$
BB10	$< a, 4 >$	$< a, 4 >, < b, 2 >$
BB11	$< a, 4 >, < b, 2 >$	$< k, 8 >, < a, 4 >, < b, 2 >$
BB12	$< a, 4 >, < b, 2 >$	$< a, 4 >, < b, 2 >, < y, 8 >$
BB13	$< a, 4 >, < b, 2 >, < y, 8 >$	$< k, 5 >, < a, 4 >, < b, 2 >, < y, 8 >$
BB14	$< a, 4 >$	$< a, 4 >$
BB15	$< a, 4 >$	$< a, 4 >$

Table 11: Very Busy Expression Algorithm Execution Table