Introduction and Overview

Abstract

This assignment covers some common problems of compilers' middle-end, namely **Very Busy Expressions – VBE**, **Dominator Analysis – DA** and **Constant Propagation – CP**, addressing them with DFA – Data Flow Analysis – framework.

Summary

All the aforementioned problems can be framed as **data-flow problems**; solving them would allow us to drive new optimizations.
The table's In/Out notation 1 is chosen under the assumption that data flow direction doesn't affect blocks' input-output naming, i.e. the **input** of a block is always its **entry point** and the **output** of a block is always its **exit point**, regardless of the problem-specific *direction*.

	Very Busy Expressions	**Dominators Analysis**	**Constant Propagation**
Domain	Set of expressions	Set of basic blocks	Set of tuples (var, lit)
Direction	Backward	Forward	Forward
Boundary Conditions	$In[BB_{Exit}] = \emptyset$	$Out[BB_{Entry}] = \{BB_{Entry}\}$	$Out[BB_{Entry}] = \emptyset$
Transfer Function	$In[BB_i] = Gen[BB_i] \cup (Out[BB_i] \setminus Kill[BB_i])$	$Out[BB_i] = In[BB_i] \cup \{BB_i\}$	$Out[BB_i] = Gen[BB_i] \cup (In[BB_i] \setminus Kill[BB_i])$
Meet Operator	$\cap$	$\cap$	$\cap$
Internal Points Init	$In[BB_i] = U \forall i \neq Exit$	$Out[BB_i] = U \forall i \neq BB_{Entry}$	$Out[BB_i] = U \forall i \neq BB_{Entry}$

Table 1: DFA Framework Summary Table

Very Busy Expressions

Problem Definition

DEF. An expression is **very busy** in a given point p **if and only if**, regardless of the path taken after p – i.e. for each path taken from p to *Exit* – , the expression is used before any of its operands is **redefined** (since, as a result, the expression is invalid).
Figuring out which expressions are **busy** in p, would enable us to hoist the expression itself to the point p.

DFA Solution

We defined the problem's **domain** as a **static set of expressions**, which is made up of all RHS expressions in the **CFG**.
The data flow direction is **backward**, since we only care about expressions that appear in the "future".
Set intersection is the **meet operator** that better fit the definition: *"[...] for each path from p to Exit [...]"*.
Given that kind of meet operator, along with the need to keep the intersection with uncomputed interior points neutral, we opted for an initialization according to the following formula: $In[BB_i] = U \forall i \neq Exit$.
The analysis is kick-started by $In[BB_{Exit}] = \emptyset$, and equivalently $Out[BB_{Exit}] = \emptyset$.

Transfer Function's Sets Definition

Each basic block BB_i **inherits** a set of expressions from its **successor** BB_{i+1}; this set makes up the output set (input from expressions-propagation POV) of block B_i:

$$Out[BB_i] = In[BB_{i+1}] \; \forall BB_i \in CFG : i \neq Exit$$

Also, each block **kills** the set of expressions invalidated by its internal definitions. An expression $E = OP1 \oplus OP2$ is killed by a basic block BB if either **OP1 or OP are defined inside the block**:

$$Kill[BB_i] = \{E = OP1 \oplus OP2 : OP1 \; redefined \wedge OP2 \; redefined\} \; \forall BB_i \in CFG$$

Lastly, the **Gen-set** is defined as the set of expressions that appear as right-hand-side of BB's

internal definitions:

$$Gen[BB_i] = \{OP1_j \oplus OP2_j : E_j = OP1_j \oplus OP2_j \forall E_j \in BB_i\} \; \forall BB_i \in CFG$$

Illustrative Iteration

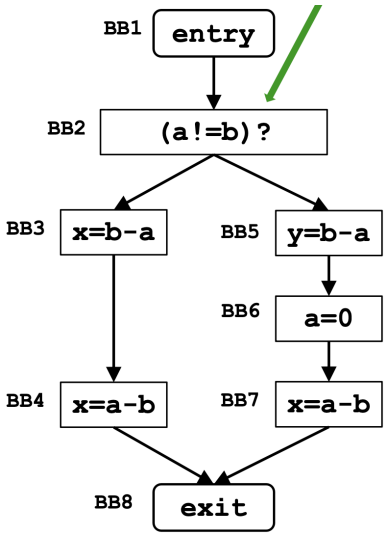


Figure 1: Reference CFG

Basic Block Name	Gen(BB)	Kill(BB)
BB2	Ø	Ø
BB3	b - a	Ø
BB4	a - b	Ø
BB5	b - a	Ø
BB6	Ø	b - a , a - b
BB7	a - b	Ø

Table 2: Gen-Kill Table

Node	In(BB)	Out(BB)
Entry	b - a	b - a
BB2	b - a	b - a
BB3	b - a , a - b	a - b
BB4	a - b	Ø
BB5	b - a	Ø
BB6	Ø	a - b
BB7	a - b	Ø
Exit	Ø	Ø

Table 3: Input-Output Table for Iteration 1

The execution stops after the first iteration since there are no back-edges in the CFG.

Dominator Analysis

Problem Definition

In a **CFG**, we say that a node X **dominates** another node Y if node X appears in every path of the graph that leads from the **Entry** block to block Y.

- We annotate each basic block B_i with a set **DOM**$[B_i]$
  - $B_i \in DOM[B_j]$, if and only if B_i dominates B_j.
- By definition, a **node dominates itself**
  - $B_i \in DOM[B_i]$

DFA Solution

We defined the domain of the problem as a **static set of blocks**, which are the **nodes** of the **CFG**.
The data flow direction is **forward**, since the concept of dominance is defined in the "past".
The **meet operator** can be inferred by the definition: *... for each path from Entry to Y ...*, is equivalent to set intersection.
Given that kind of meet operator, along with the need to keep the **intersection** with uncomputed interior points **neutral**, interior points are initialized according to the following formula: $Out[BB_i] = U \; \forall i \neq Entry$.
The analysis is kick-started by $Out[BB_{Entry}] = Entry$, since according to the definition: *each node dominates itself.*

Transfer Function's Sets Definition

Each basic block BB_i **inherits** a set of expressions from its **predecessor** BB_{i-1}; this set makes up the block's input set B_i:

$$In[BB_i] = Out[BB_{i-1}] \; \forall BB_i \in CFG : i \neq Entry$$

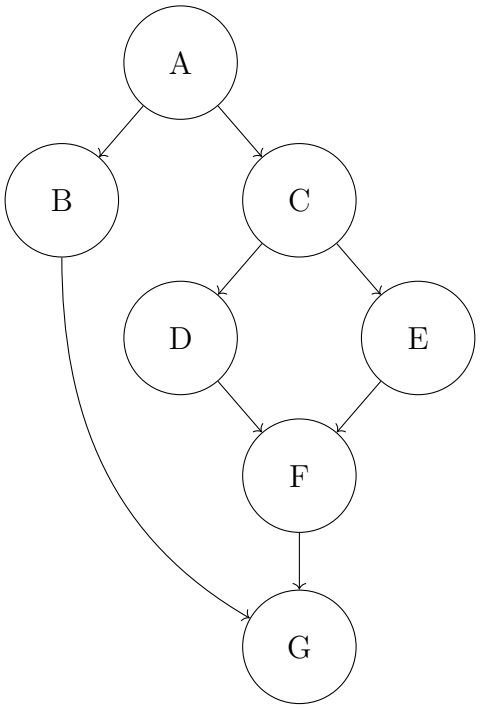
The **kill set is empty** because, considering each node BB_i, name conflicts don't show up. Each node **add** to the dominators' set **itself** without deleting other elements. Some elements could be removed only by applying the **intersection** between the **dominators sets coming from different paths**.

$$Kill[BB_i] = \emptyset \; \forall BB_i \in CFG$$

Lastly, as mentioned above, the **Gen-set** of the BB_i is defined as a set with only the BB_i since **each node dominates itself**:

$$Gen[BB_i] = \{BB_i\} \; \forall BB_i \in CFG$$

Illustrative Iteration

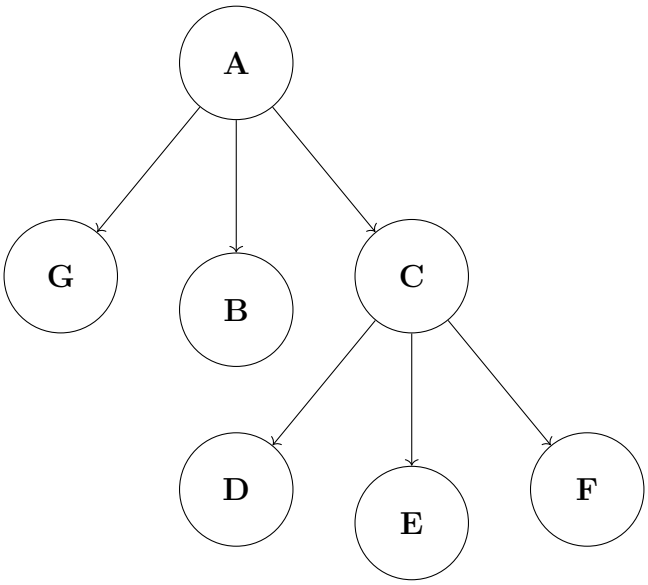


Node	In(BB)	Out(BB)
A	∅	A
B	A	A , B
C	A	A , C
D	A , C	A , C , D
E	A , C	A , C , E
F	A , C	A , C , F
G	A	A , G

Table 4: Input-Output Table for Iteration 1

The execution stops after the first iteration since there are no back-edges in the CFG.

Dominator Tree



Constant Propagation

Problem Definition

The goal of **constant propagation** is to determine at which points in the program variables have a constant value.
The information to be computed for each node **n** of the Control Flow Graph (**CFG**) is a set of pairs of the form < **variable, constant value** >.
If we have the pair < **x, c** > at node n, it means that x is guaranteed to have the value c every time n is reached during program execution.

DFA Solution

This solution works under the assumption that the transfer function can integrate some piece of logic. In particular, some sort of expression-solving mechanism is needed to compute expressions such as $k + 2$, expanding them to a constant value. For instance, $k + 2$ expression in BB3 of 2, should become 4 after passing through BB3's transfer function.
Consequently, the domain of this problem, defined as the set of tuples < **variable, constant value** >, is variable-sized, thus making unfeasible a bit-vector implementation. This is due to the fact we have no prior knowledge about the number of tuples that could appear while algorithm is executing. As a matter of fact,
For a solution that admits a bit-vector implementation, have a look at our considerations in **Constant Propagation With Fixed-size Domain**.
We defined the domain of the problem as a set of definitions with literal or constant values as RHS, represented as tuples < **variable, constant value** >.
The data flow is oriented forward because it's essential to reference past values of variables to ensure which ones remain constant.
The meet operator is equivalent to set intersection because a variable is deemed constant only if it's confirmed across every potential path.
Given that kind of meet operator, along with the need to keep the intersection with uncomputed interior points neutral, interior points are initialized according to the following formula: $Out[BB_i] = U \forall i \neq Entry$.
The analysis is kick-started by $Out[BB_{Entry}] = \emptyset$.
The solution should use a dynamic size dictionary, because determining the value of a variable defined using operands with constant values requires computation. Each dictionary element's key consists of a pair < **variable, constant value** > associated to a boolean value representing whether the variable assumes a constant value at that program point.

Transfer Function's Sets Definition

The kill and gen set is defined using tuple of $<$ **variable, expression** $>$ to differentiate definition of the same variables which during the execution could have the same result. The Transfer Function will compute the result of the expression by inserting in the dictionary the tuple $<$ **variable, constant value** $>$. If in the point p of the program the input dictionary does not contain some of the variables in the expression which has to be computed, the tuple $< variable, expression >$ is discarded from the output set.
Each basic block BB_i inherits a set of expressions from its predecessor BB_{i-1}; this set makes up the block's input set B_i:

$$In[BB_i] = Out[BB_{i-1}] \; \forall BB_i \in CFG : i \neq Entry$$

Also, each block kills the tuple $<$ **variable, expression** $>$ invalidated by its internal definitions. A tuple $<$ **variable, expression** $>$ is killed by a basic block BB if the variable is redefined inside the block:

$$Kill[BB_i] = \{< variable, \; expression > \mid variable \; redefined\} \; \forall BB_i \in CFG$$

Lastly, the gen-set is defined as the set of tuple of $<$ **variable, expression** $>$ whenever the BB's internal definition of the variable is a *constant expression*, or rather a binary expression where one or both operands are variables whose constant value is known.
Let's define K as a set of constant values, and k as a constant value, $\forall k \in K$, we can declare an *Operand* with constant value as Op_k.
The **Gen-set** based on these assumptions is the following:

$$Gen[BB_i] = \{< variable, \; expression > \mid \forall E_j \in BB_i \; \wedge \; Op_k \; \forall k \in K\} \; \forall BB_i \in CFG$$

Illustrative Iteration

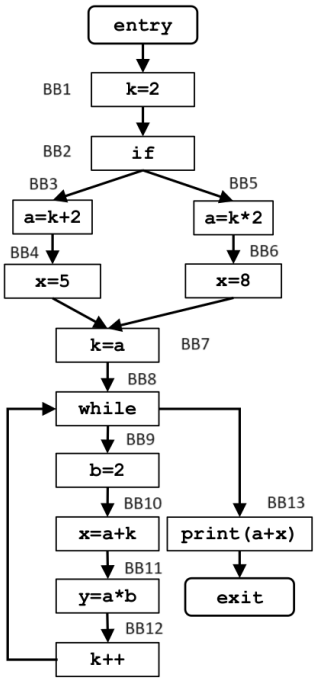


Figure 2: Reference CFG

Basic Block Name	Gen(BB)	Kill(BB)
BB1	$< k, 2 >$	$\emptyset$
BB2	$\emptyset$	$\emptyset$
BB3	$< a, k + 2 >$	$< a, k * 2 >$
BB4	$< x, 5 >$	$< x, 8 >$, $< x, a + k >$
BB5	$< a, k * 2 >$	$< a, k + 2 >$
BB6	$< x, 8 >$	$< x, 5 >$, $< x, a + k >$
BB7	$< k, a >$	$< k, 2 >$, $< k, k + + >$
BB8	$\emptyset$	$\emptyset$
BB9	$< b, 2 >$	$\emptyset$
BB10	$< x, a + k >$	$< x, 5 >$, $< x, 8 >$
BB11	$< y, a * b >$	$\emptyset$
BB12	$< k, k + + >$	$< k, 2 >$, $< k, a >$
BB13	$\emptyset$	$\emptyset$

Table 5: Gen-Kill Table

Iteration 1

Node	In(BB)	Out(BB)
Entry	$\emptyset$	$\emptyset$
BB1	$\emptyset$	$< k, 2 >$
BB2	$< k, 2 >$	$< k, 2 >$
BB3	$< k, 2 >$	$< a, 4 > , < k, 2 >$
BB4	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 5 >$
BB5	$< k, 2 >$	$< a, 4 > , < k, 2 >$
BB6	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 8 >$
BB7	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 4 >$
BB8	$< a, 4 > , < k, 4 >$	$< a, 4 > , < k, 4 >$
BB9	$< a, 4 > , < k, 4 >$	$< a, 4 > , < b, 2 > , < k, 4 >$
BB10	$< a, 4 > , < b, 2 > , < k, 4 >$	$< a, 4 > , < b, 2 > , < k, 4 > , < x, 8 >$
BB11	$< a, 4 > , < b, 2 > , < k, 4 > , < x, 8 >$	$< a, 4 > , < b, 2 > , < k, 4 > , < x, 8 > , < y, 8 >$
BB12	$< a, 4 > , < b, 2 > , < k, 4 > , < x, 8 > , < y, 8 >$	$< a, 4 > , < b, 2 > , < k, 5 > , < x, 8 > , < y, 8 >$
BB13	$< a, 4 > , < k, 4 >$	$< a, 4 > , < k, 4 >$
Exit	$< a, 4 > , < k, 4 >$	

Table 6: Input-Output Table, Iteration 1

The execution continues after the first iteration since there are back-edges in the CFG.

Iteration 2

Node	In(BB)	Out(BB)
Entry	$\emptyset$	$\emptyset$
BB1	$\emptyset$	$< k, 2 >$
BB2	$< k, 2 >$	$< k, 2 >$
BB3	$< k, 2 >$	$< a, 4 > , < k, 2 >$
BB4	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 5 >$
BB5	$< k, 2 >$	$< a, 4 > , < k, 2 >$
BB6	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 8 >$
BB7	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 4 >$
BB8	$< a, 4 >$	$< a, 4 >$
BB9	$< a, 4 >$	$< a, 4 > , < b, 2 >$
BB10	$< a, 4 > , < b, 2 >$	$< a, 4 > , < b, 2 >$
BB11	$< a, 4 > , < b, 2 >$	$< a, 4 > , < b, 2 > , < y, 8 >$
BB12	$< a, 4 > , < b, 2 > , < y, 8 >$	$< a, 4 > , < b, 2 > , < y, 8 >$
BB13	$< a, 4 >$	$< a, 4 >$
Exit	$< a, 4 >$	

Table 7: Input-Output Table, Iteration 2

In the BB10 the tuple of the gen set $<$ **x, a+k** $>$ is discarded because the variable k is not defined in the input dictionary.

Iteration 3

Node	In(BB)	Out(BB)
Entry	$\emptyset$	$\emptyset$
BB1	$\emptyset$	$< k, 2 >$
BB2	$< k, 2 >$	$< k, 2 >$
BB3	$< k, 2 >$	$< a, 4 > , < k, 2 >$
BB4	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 5 >$
BB5	$< k, 2 >$	$< a, 4 > ; < k, 2 >$
BB6	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 2 > , < x, 8 >$
BB7	$< a, 4 > , < k, 2 >$	$< a, 4 > , < k, 4 >$
BB8	$< a, 4 >$	$< a, 4 >$
BB9	$< a, 4 >$	$< a, 4 > , < b, 2 >$
BB10	$< a, 4 > , < b, 2 >$	$< a, 4 > , < b, 2 >$
BB11	$< a, 4 > , < b, 2 >$	$< a, 4 > , < b, 2 > , < y, 8 >$
BB12	$< a, 4 > , < b, 2 > , < y, 8 >$	$< a, 4 > , < b, 2 > , < y, 8 >$
BB13	$< a, 4 >$	$< a, 4 >$
Exit	$< a, 4 >$	

Table 8: Input-Output Table, Iteration 3

The execution stops after the third iteration since there are no changes in the input and output sets.

Constant Propagation With Fixed-size Domain