

# **CSC2/455 Software Analysis and Improvement**

## **Live variable analysis**

---

Sreepathi Pai

January 25, 2023

URCS

# Outline

Review

Live Variable Analysis

Live Variables using Iterative Data flow Analysis

Postscript

# Outline

Review

Live Variable Analysis

Live Variables using Iterative Data flow Analysis

Postscript

# What we know how to do so far

- Basic blocks to Control Flow Graphs
  - Section 5.4.3 in Cooper and Turczon
- Local optimizations (Value Numbering)

# Outline

Review

Live Variable Analysis

Live Variables using Iterative Data flow Analysis

Postscript

# Motivation

```
void example(int x) {  
    int a = 0;    /* def 1 */  
    int b = 0;    /* def 2 */  
  
    if(x > 1) {  
        a = b;    /* def 3 */  
    } else {  
        a = x;    /* def 4 */  
    }  
  
    return a;  
}
```

Must the compiler emit code to initialize a to 0?

# Motivation

```
void example(int x) {  
    int a = 0;    /* def 1 */  
    int b = 0;    /* def 2 */  
  
    if(x > 1) {  
        a = b;    /* def 3 */  
    } else {  
        a = x;    /* def 4 */  
    }  
  
    return a;  
}
```

`a` is *defined* (i.e. written) thrice in the code above. Which definitions are “alive” at return `a`?

# Terminology

- A variable  $x$  is live at a definition (i.e. a write) if there is a subsequent read of the variable along a program path from that definition.
  - Alternatively: is this variable overwritten along all paths before a subsequent read?
- Can we identify live variables just by analyzing basic blocks (i.e. local analysis)?



## More Motivation(?)

```
void test() {  
    int i;  
  
    // begin delay  
    for(i = 0; i < 10000; i++);  
    // end delay  
}
```

# GCC without optimization

```
test():
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], 0
.L3:
    cmp     DWORD PTR [rbp-4], 9999
    jg      .L4
    add     DWORD PTR [rbp-4], 1
    jmp     .L3
.L4:
    nop
    pop     rbp
    ret
```

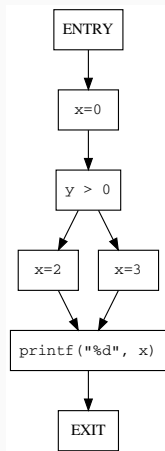
Code courtesy Compiler Explorer at [godbolt.org](http://godbolt.org) (this is more convenient than `gcc -Wa` if you have the Internet)

```
test():  
    ret
```

Code courtesy Compiler Explorer at [godbolt.org](http://godbolt.org)

# Example 1

```
x = 0; /* d1 */  
  
if(y > 0)  
    x = 2; /* d2 */  
else  
    x = 3; /* d3 */  
  
printf("%d\n", x);
```

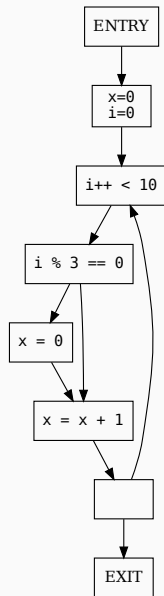


## Example 2

```
x = 0          /* d1 */
i = 0          /* d2 */
while(i++ < 10) { /* d3 */
    if(i % 3 == 0) {
        x = 0; /* d4 */
    }

    x = x + 1; /* d5 */
}
```

- Which definition(s) of  $x$  are live at entry of loop?
- Which definition(s) of  $x$  are live in the loop?
- Which definition(s) of  $x$  are live at exit of loop?



## Relook at the Loop example

```
    i = 1;
L1:  if(!(i < 10000)) goto L2;
     i++;
     goto L1;

L2:  return
```

Where in the CFG must *i* be “dead” to eliminate it?

## Some Definitions

For a basic block  $s$ :

- $DEF(s)$  (for “defined”) is the set of variables written by  $s$
- $USE(s)$  is the set of variables read by  $s$

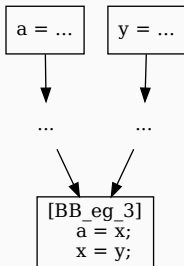
For the blocks below, what are the  $DEF$  and  $USE$  sets?

```
BB_eg_0:  
  i = i + 1
```

```
BB_eg_1:  
  a = x;  
  x = 3;
```

```
BB2_eg_2:  
  t = a;
```

## Effects of definitions and reads



- What happens to a definition of `a` in a basic block that is before `BB_eg_3` on some path through the CFG?
- What can we say about definitions of `y` in other basic blocks that are before `BB_eg_3` on some path through the CFG?



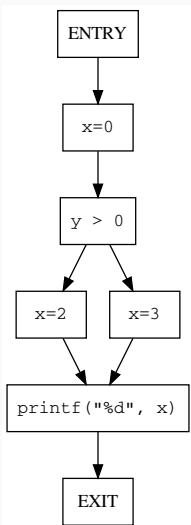
## More definitions

- Upwardly exposed  $UE(s)$  are those variables that are read *before* they are written to in basic block  $s$ .
  - e.g.  $x$  and  $y$  in  $BB_{eg\_3}$  in the previous slide
- A variable is alive in a basic block if it is in the set:
  - ?
- A variable is killed in a basic block if it is in the set:
  - ?

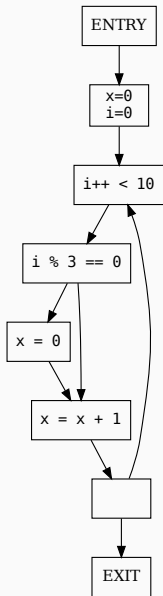
## Finding out if a definition is live

- Start from the definition, and see if it “reaches” a use
  - i.e., it is not killed on *some* path (in the CFG) to a use
- Start from a use, and walk backwards to see if you can reach the definition
  - Repeat for all uses

# Example #1



## Example #2



# Limitations

When does enumerating paths work in practice?

## A More Practical Way: Iterative Data flow Analysis

- Setup data flow equations
- Solve these data flow equations

# Outline

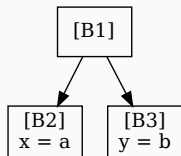
Review

Live Variable Analysis

Live Variables using Iterative Data flow Analysis

Postscript

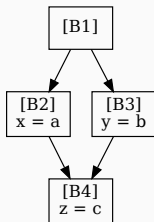
## Equations for Live Variables



- Each basic block has “incoming” edges (from predecessors) and “outgoing” edges (to successors).
- When is an “incoming” variable live “into” a block?
  - it is in UE for that block
- When is a variable live out of a block  $n$ ?
  - it is live into a successor block  $m$
  - first try:  $LIVEOUT(n) = \cup_m UE(m)$  (where  $m \in succ(n)$ )
  - what are we missing?



## Complete equation



$$\text{LIVEOUT}(n) = \cup_{m \in \text{succ}(n)} \text{UE}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{DEF}(m)})$$

The variables live out of block  $n$  must:

- be upwardly exposed in a successor  $m$
- OR be live out of  $m$  and NOT be killed by  $m$  (i.e.  $\overline{\text{DEF}(m)}$ )
  - note this is equivalent to  $(\text{LIVEOUT}(m) - \text{DEF}(m))$

How would you begin solving this equation?

# Iterative Data flow Analysis Algorithm for Live Variables

- Live variables is a *backwards dataflow analysis*
  - “Facts” flow from a node’s successors to it
- Initialization:
  - Compute  $DEF(s)$  for all blocks  $s$
  - Compute  $UE(s)$  for all blocks  $s$
  - Set  $LIVEOUT(s) = \emptyset$  for all  $s$
- Repeat (iterate):
  - Compute  $LIVEOUT(s)$  for all blocks  $s$
  - Until no set  $LIVEOUT(s)$  changes – i.e. until a fixpoint is reached.

# Termination and correctness

- Termination
  - Note  $LIVEOUT(s)$  only increases in size or remains the same in every iteration
  - What is the maximum size of  $LIVEOUT(s)$ ?
- Correctness
  - When the algorithm terminates, all computed  $LIVEOUT(s)$  sets satisfy the equation
  - and hence meet the definition of “liveness” .

## Quality of solution?

How does the iterative solution differ from the path-based solution?

# Outline

Review

Live Variable Analysis

Live Variables using Iterative Data flow Analysis

Postscript

## References

- Chapter 8 of Cooper and Turczon
  - Section 8.6.1
- Also recommended:
  - Aho, Lam, Sethi and Ullman, Chapter 9, Section 9.2.5
- Spare time reading (esp. for PhD students):
  - Frances E. Allen, John Cocke, 1971, A Catalogue of Optimizing Transforms