CS202: Software Tools and Techniques for CSE

Lecture 8

Shouvick Mondal

shouvick.mondal@iitgn.ac.in

August 2025

Memory Optimization

- Reuse memory / register wherever possible.
- z and y can reuse memory / register.

```
0  int x = 2, y = 3, z = 1;
1  if (x == 2) {
2      y = z;
3      x = 9;
4      y = 7;
5      x = x – y;
6  } else {
7      y = x + z;
8      ++x;
9  }
10 printf(“%d”, y);
```

Live Ranges

x : (0,1), (3,4,5), (0,1,6,7,8)

y : (4,5,10), (7,8,9,10)

z : (0,1,2), (0,1,6,7)

This optimization demands computation of live variables.

DFA for Live Variables

Domain	Sets of variables
Transfer function	$\mathrm{in}(B) = \mathrm{use}(B) \cup (\mathrm{out}(B)\quad \mathrm{def}(B))$ $\mathrm{out}(B) = \cup\ \mathrm{in}(S)$ where S is a successor of B
Direction	Backward
Meet / confluence operator	$\cup$
Initialization	$\mathrm{in}(B) = \{\ \}$

Definition: A variable v is live at a program point p if v is used along some path in the flow graph starting at p. Otherwise, the variable v is dead.

How to compute live variables?

Algorithm for Live Variable Analysis

```
for each basic block B
  compute gen(B) and kill(B)
  out(B) = { }

do {
  for each basic block B
      in(B) = U out(P) where P ∊ pred(B)
      out(B) = gen(B) U (in(B)–kill(B))
} while in(B) changes for any basic block B
```

Algo for **reaching definitions**

Domain	Sets of variables
Transfer function	in(B) = use(B) U (out(B) def(B)) out(B) = U in(S) where S is a successor of B
Direction	Backward
Meet / confluence operator	U
Initialization	in(B) = { }

Parameters for **live variable analysis**

Direction and Confluence

	Forward	Backward
∪	Reaching Definitions	Live Variables
∩	Available Expressions	Very Busy Expressions

An expression is available at a program point P if the expression is computed <u>along each path</u> to P (from START) without getting invalidated.

An expression is very busy at a program point P if <u>along each path</u> from P (to END) the expression is computed without getting invalidated.

Data Flow Framework

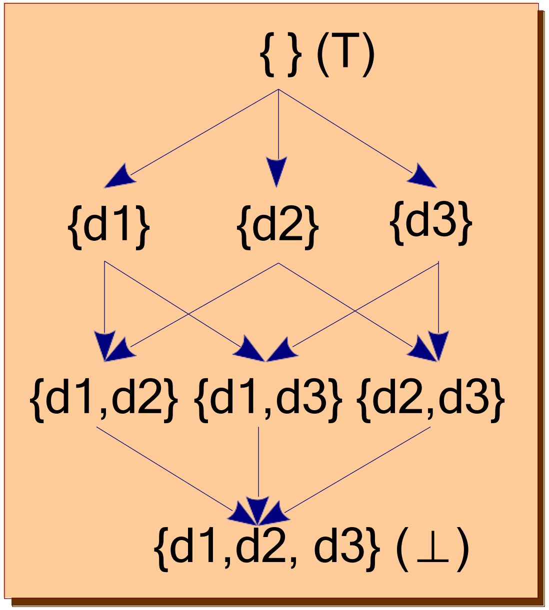
- Point: start or end of a basic block
- Information flow direction: forward / backward
- Transfer functions
- Meet / confluence operator

- One can define a transfer function over a path in the CFG $f_k(f_{k1}(\ldots f_2(f_1(f_0(T))\ldots))$ // small k(block)
- $MOP(x) = \Pi f_K(T) \quad K \in Paths(x)$ // capital K (path)

Meet over all paths
Path enumeration is expensive

Structure in Data Flow Framework

- A semilattice L with a binary meet operator $\sqcap$, such that $a, b, c \in L$
  - Idempotency: $a \sqcap a = a$
  - Commutativity: $a \sqcap b = b \sqcap a$
  - Associativity: $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$
- $\sqcap$ imposes an order on L
  - $a >= b \Leftrightarrow a \sqcap b = b$
- L has a bottom element $\perp$, $a \sqcap \perp = \perp$
- L has a top element T, $a \sqcap T = a$



Reaching Definitions Lattice

Analysis Dimensions

An analysis's precision and efficiency is guided by various design decisions.

- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

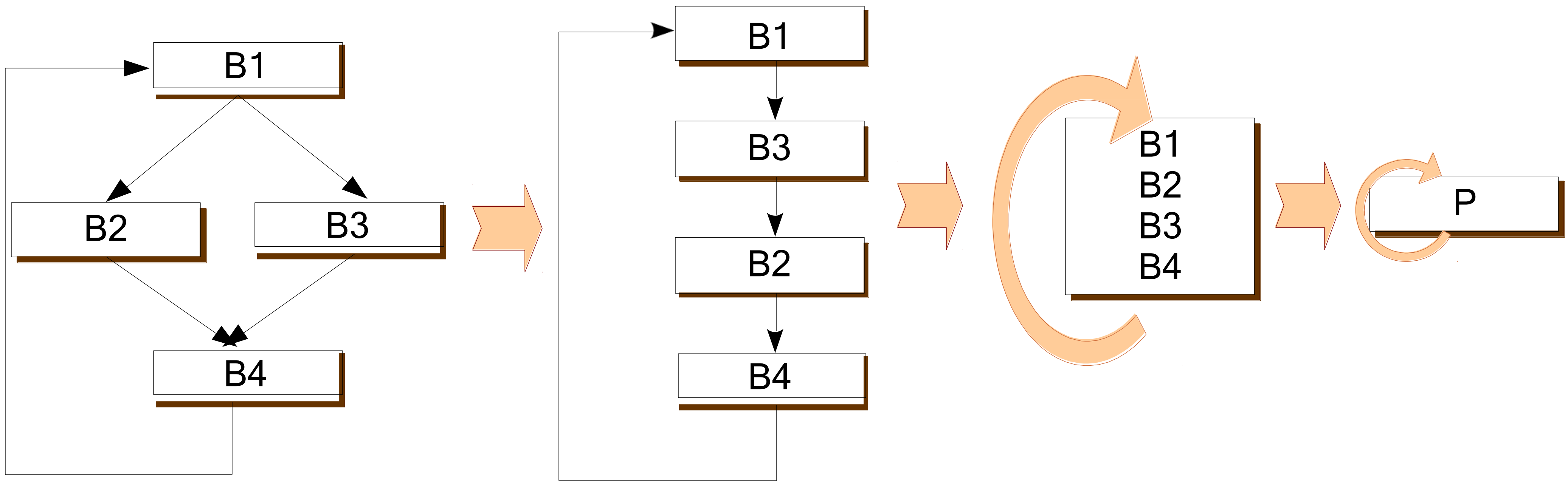
How many hands are required to know the time precisely?

Flow-sensitivity

```
L0: a = 0;
L1: a = 1;
L2: ...
```

Flow-sensitive solution: *at L1 a is 0, at L2 a is 1*
Flow-insensitive solution: *in the program a is in {0, 1}*

Flow-insensitive analyses ignore the control-flow in the program.

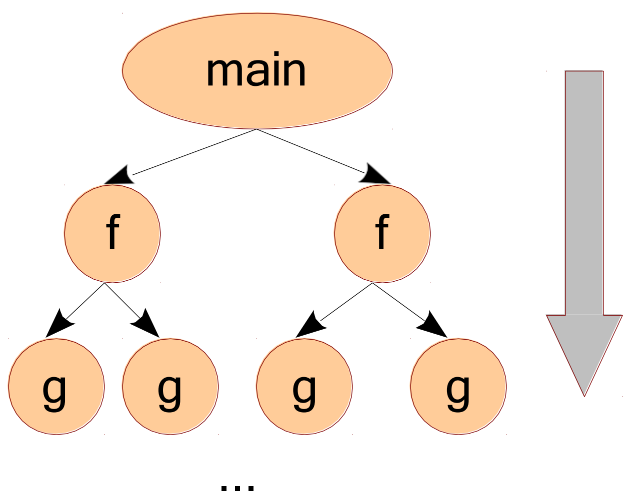


Context-sensitivity

```
main() {                    fun(int x) {
      L0:  fun(0);                y = x;
      L1:  fun(1);          }
}
```

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:
y is in {0, 1} in the program



Exponential Number of contexts

Along main-f1-g1, …
Along main-f1-g2, …
Along main-f2-g1, …
Along main-f2-g2, ...

Exponential time requirement

Exponential storage requirement

Context-sensitivity

main() {
L0: fun(0);
L1: fun(1);
}
	fun(int x) {
	y = x;
	}

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:

Inter-procedural → *y is in {0, 1} in the program*

intra-procedural → *y is in {-∞, +∞} in the program*

Path-sensitivity

```
if (a == 0)
        b = 1;
else
        b = 2;
```

Path-sensitive solution:

b is 1 when a is 0, b is 2 when a is not 0

Path-insensitive solution:

b is in {1, 2} in the program

```
if (c1)
   while (c2) {
     if (c3)
       …
     else
       for (; c4; )
          ...
   }
else
   ...
```

c1 and c2 and c3, …
c1 and c2 and !c3 and c4, …
c1 and c2 and !c3 and !c4, …
c1 and !c2, …
!c1 ...
...

Field-sensitivity

```
struct T s;

s.a = 0;
s.b = 1;
```

Field-sensitive solution:
s.a is 0, s.b is 1

Field-insensitive solution:
s is in {0, 1}

Aggregates are collapsed into a single variable.
e.g., arrays, structures, unions.

This reduces the number of variables tracked during the analysis and reduces precision.

Texts, References, and Acknowledgements

Online:

- Continuous Integration and Delivery (**CircleCI**: https://circleci.com)
- http://www.cse.iitm.ac.in/~rupesh/teaching/pa/jan19

Textbook:

- Sharp, J. (2022). *Microsoft Visual C# Step by Step*, 10th edition, Microsoft Press.
- Watson, K., Nagel, C., Pedersen, J. H., Reid, J. D., & Skinner, M. (2008). *Beginning Microsoft Visual C# 2008*. John Wiley & Sons.
- Mark J. Price (2024). *C# 13 and .NET 9 – Modern Cross-Platform Development Fundamentals*, 9th edition, Packt Publishing Ltd.

Reference:

- Soni, M. (2016). *DevOps for Web Development*. Packt Publishing Ltd.
- Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. *How different are different diff algorithms in Git? Use* `--histogram` *for code changes*. Empirical Softw. Engg. 25, 1 (Jan 2020), 790–823.