CS614: Advanced Compilers

Optimizations based on SSA

Manas Thakur
CSE, IIT Bombay

Spring 2025

Things we learnt last week

"We need someone with an overview of the situation."

- Constant propagation (+folding) allows partial execution of a program.
- IDFAs allow flow-sensitive constant propagation.
- Flow-insensitive constant propagation is faster but may lose precision.
- SSA form allows us to identify a unique definition for each use.
- A program can be converted to SSA form in two steps:
  - inserting phi-functions at respective iterative dominance frontiers;
  - renaming variables to use the closest definition.

Recall Simple Constant Propagation

```
for each n {
    for each v:
        IN[n,v] = \top
    for each v:
        OUT[n,v] = \top
}
repeat
    for each n {
        save older values of IN and OUT
        for each v in USE[n] {
            IN[n,v] = IN[n,v] \meet OUT[p,v] for each predecessor p of n
        }
        OUT[n,v] = copy(IN[n,v])
        for each v in DEF[n] {
            switch (n) {
                case "v = \cons":
                    OUT[n,v] = \cons
                case "v = w":
                    OUT[n,v] = IN[n,w]
                case "v = w1 op w2":
                    OUT[n,v] = IN[n,w1] op IN[n,w2]
            }
        }
    }
until fixed-point
```

Initialization

Iterate over all nodes until fixed point

Dataflow computation

Flow sensitive: Precise but very expensive

Flow insensitive: Cheap but very imprecise

```
for each variable v:
    VAL(v) = \top
repeat
    for each n {
        for each v in DEF[n] {
            switch (n) {
                case "v = \cons":
                    VAL[v] = VAL[v] \meet \cons
                case "v = w":
                    VAL[v] = VAL[v] \meet VAL[w]
                case "v = w1 op w2":
                    VAL[v] = VAL[v] \meet (VAL[w1] op VAL[w2])
            }
        }
    }
until fixed-point
```

Single global information about all variables

Dataflow computation until fixed point

Sparse Simple Constant Propagation (using SSA)

```
for each variable v:
   VAL(v) = \top
for each statement 's' of the form "v = \cons" {
   VAL(v) = \cons
   worklist.addAll(du edges originating from 's')
}
while !worklist.isEmpty() {
   e = worklist.removeOne()
   evaluate e.dst using VAL(v)
   if VAL(DEF(e.dst)) changed:
      worklist.addAll(du edges originating at e.dst)
}
```

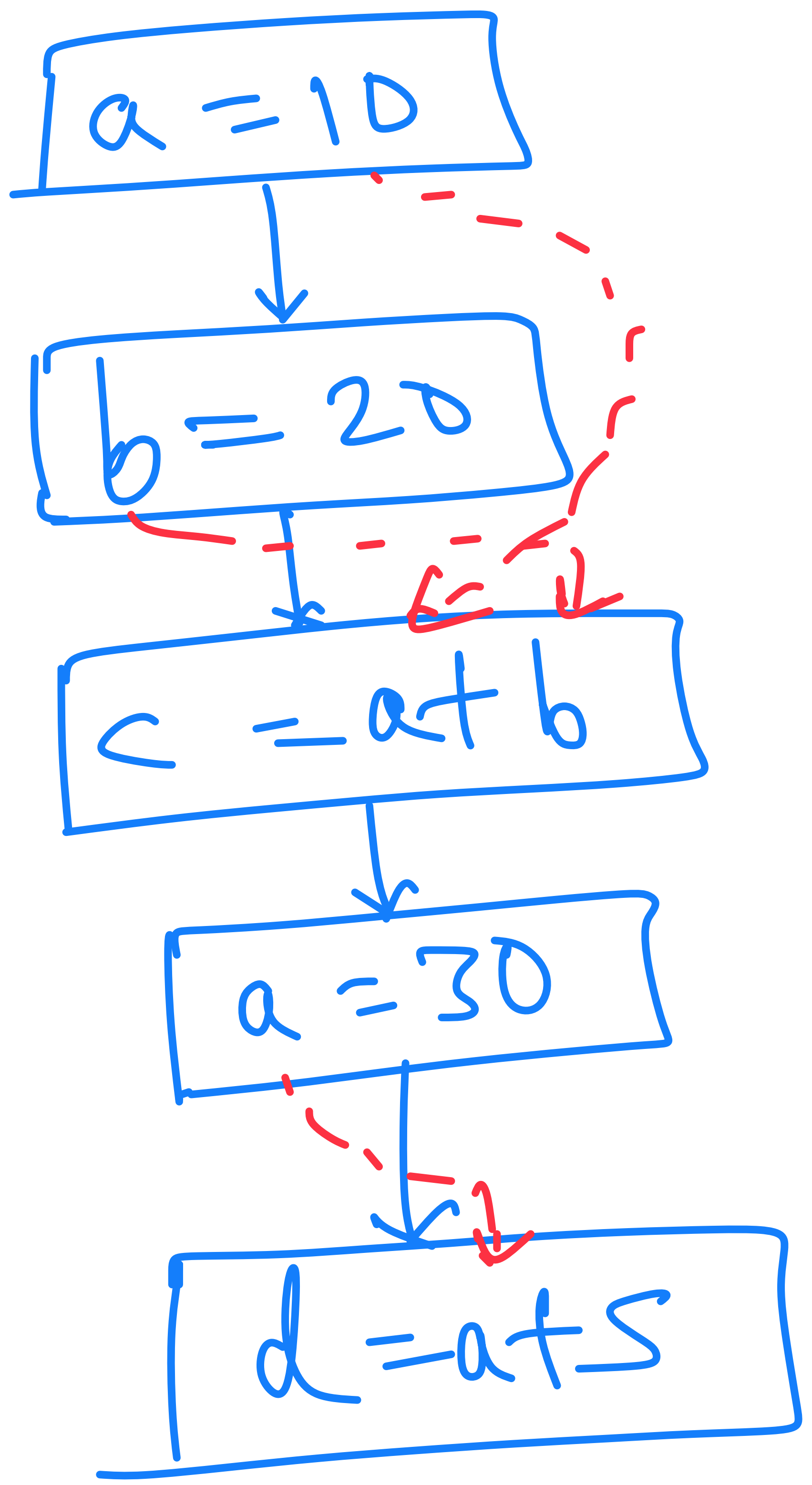
Single global information about all variables

Propagation only over SSA (du) edges

Efficient yet precise!

SSCP: Practice

```
a = 10;
b = 20;
c = a + b;
a = 30;
d = a + 5;
```



Solid edges are flow edges
Dashed edges are SSA edges

```
a = 10;
b = 20;
c = 30;
a = 30;
d = 35;
```

Example: Limitations of Simple CP

```
a = 10;
b = 20;
if (a == 10)
   x = a;
else
   x = b;
y = x;
```

```
a = 10;
b = 20;
if (a == 10)
   x = 10;
else
   x = 20;
y = x;
```

Can we handle conditionals better and propagate only over *executable* branches?

Optimization 1: Sparse Conditional Constant Propagation

Conditional Constant Propagation

- ➤ Add `true` and `false` constants to the lattice
- ➤ Maintain ***executability*** of CFG edges

```
for each n {
    for each v:
        IN[n,v] = \top
        OUT[n,v] = \top
}
for each CFG edge e:
    EXEC(e) = false
```

Initialization similar to standard worklist-algorithm

Conditional Constant Propagation (Cont.)

```
EXEC(edges originating from entry) = true
worklist = {edges originating from entry}
while !worklist.isEmpty() {
   e = worklist.removeOne()
   evaluate e.dst (INs and OUTs)
   if OUT changed {
      switch (e.dst) {
         case "if (c)":
            if c evaluates to true:
               EXEC(e.dst --> true-succ) = true
            elif c evaluates to false:
               EXEC(e.dst --> false-succ) = true
            else:
               EXEC(SUCC(e.dst)) = true
         default:
            EXEC(SUCC(e.dst)) = true
      }
   }
   worklist.addAll(marked EXEC edges)
}
```

Start with forward flow

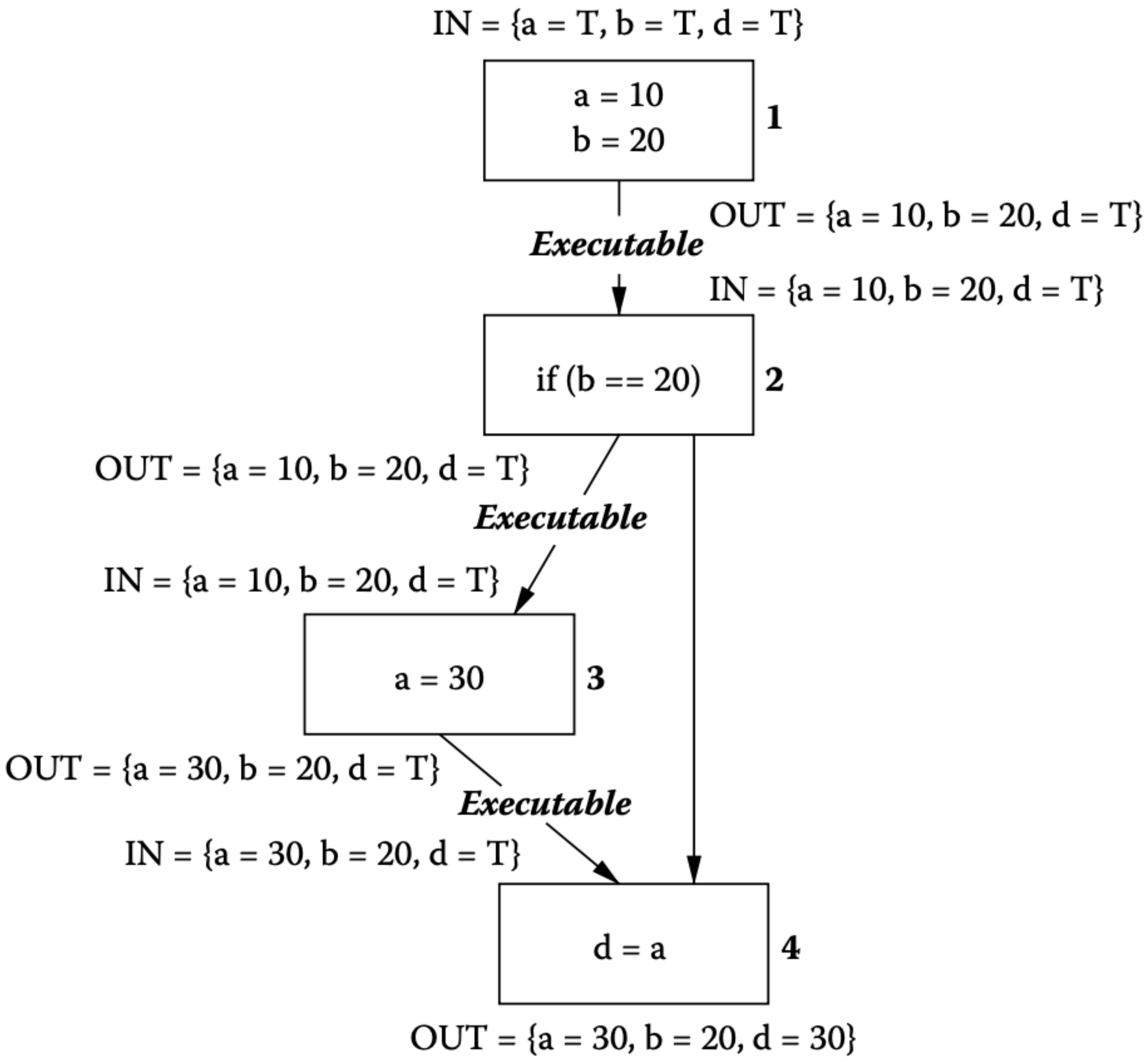
Dataflow computation as usual

Process only executable branches

Higher precision than Simple CP

Example: Conditional CP

```
a = 10;
b = 20;
if (b == 20)
   a = 30;
d = a;
```



Sparse Conditional Constant Propagation

```
for each variable v:
    VAL(v) = \top
```

Single global information

```
for each CFG edge e:
    EXEC(e) = false
```

Executability initialization

```
flowWorklist = {edges originating from entry}
ssaWorklist = \empty
```

Two different worklists

`flowWorklist`: Computation of `EXEC`

`ssaWorklist`: Propagation of values

Sparse Conditional Constant Propagation (Cont.)

```
while !flowWorklist.isEmpty() || !ssaWorklist.isEmpty() {
    e = flowWorklist.removeOne()
    if !EXEC(e) {
        EXEC(e) = true
        if e.dst is a phi node:
            call visitPhi(e.dst)
        if e.dst has never been visited:
            call visitNode(e.dst)
        if e.dst has only one successor 's':
            flowWorklist.add(e.dst --> s)
    }
    e = ssaWorklist.removeOne()
    for each e.dst {
        if e.dst is a phi node:
            call visitPhi(e.dst)
        else {
            if EXEC(e) is true for any incoming edge of e.dst:
                call visitNode(f.dst)
        }
    }
}
```

Process both worklists together

Special treatment of phi nodes

Propagate over SSA edges only on executable branches

Sparse Conditional Constant Propagation (Cont.)

```
visitPhi(n: x = \phi(e1,e2,...,en)) {
    evaluate x by taking a meet of the values coming
      only from executable incoming branches
}

visitNode(n) {
    evaluate n
    if n is of the form "if (c)" {
        if c evaluates to true:
            EXEC(n --> true-succ) = true
        elif c evaluates to false:
            EXEC(n --> false-succ) = true
        else:
            EXEC(SUCC(n)) = true
        flowWorkList.addAll(marked EXEC edges)
    }
    elif VAL[DEF[n]] changed {
        // assignment statement
        ssaWorklist.add(SSA_EDGES(n))
    }
}
```

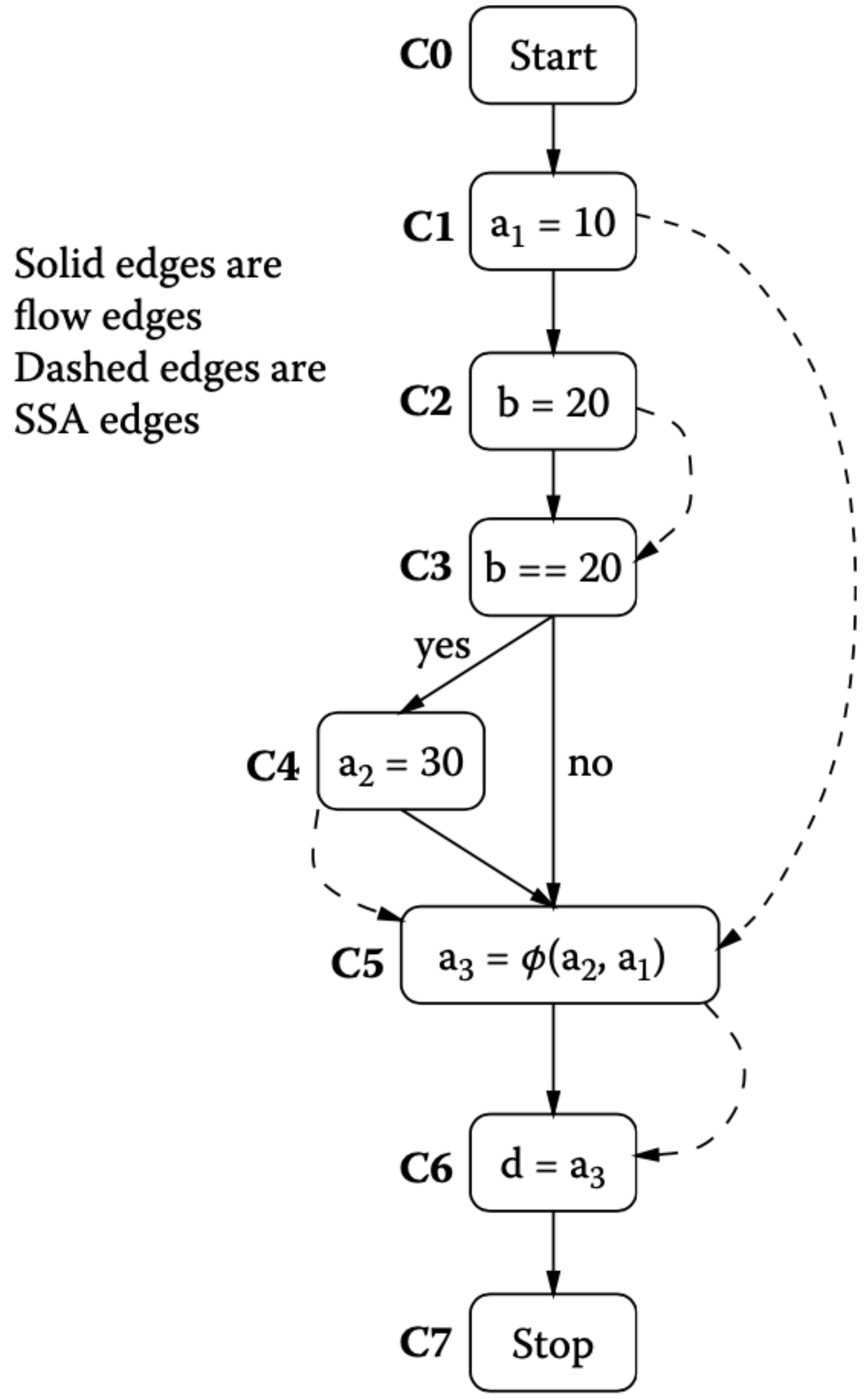
Ignore phi inputs from non-executable branches

Similar to Conditional CP but single global information

More precise than Simple CP; faster than plain CCP (due to SSA)

Example: SCCP

```
a = 10;
b = 20;
if (b == 20)
    a = 30;
d = a;
```




```
a = 10;
b = 20;
if (b == 20)
    a = 30;
d = 30;
```

Common sub-expression elimination

- **Idea:** If a program computes the same value multiple times, reuse the value.

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

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

- How about the following code?

```
x = a + b;
y = a;
z = y + b;
```

Next Class

Global Value Numbering

- We need something more powerful that exact expression matching.

A2: SCP + CHA_MI