Check-in

Review Lecture: Flowgraphs

Draw the CFG of this procedure

```
f: () -> void{
     a:int;
    a = 256;
   while(true){
        if (a > 500){
              a = a++;
        }
    }
}
```

Announcements

Administrivia

Drew Davidson | University of Kansas

Dataflow

Previously…

Review Lecture: Flowgraphs

Control flow graphs:

A hybrid IR/ a structural overlay

- Rationale

  Useful for visualizing program flow

- Construction

  Identify basic blocks (BBLs)

  Connect edges on control transfer

- Uses

  Program understanding

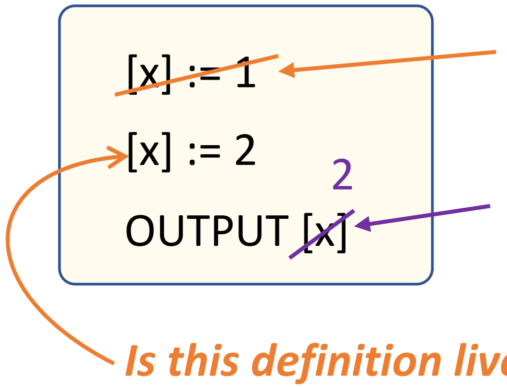
<u>You should know</u>

- Basic Blocks
- How to build a CFG
- The idea of some local optimizations
  - Dead Code Elimination
  - Common Subexpression Elimination
  - Constant/Copy Propagation

Optimization

Recall: Some Local Optimizations

Review - Basic Block Optimization



~~[x] := 1~~

[x] := 2

OUTPUT ~~[x]~~ 2

Dead Code Elimination: Guaranteed no future use of this definition (the assignment is not “live”)

Constant Propagation: Guaranteed previous static definition of this use

Is this definition live?

¯_(ツ)_/¯

Without knowing x’s use
outside this block
We have to keep it

Today’s Outline

Dataflow

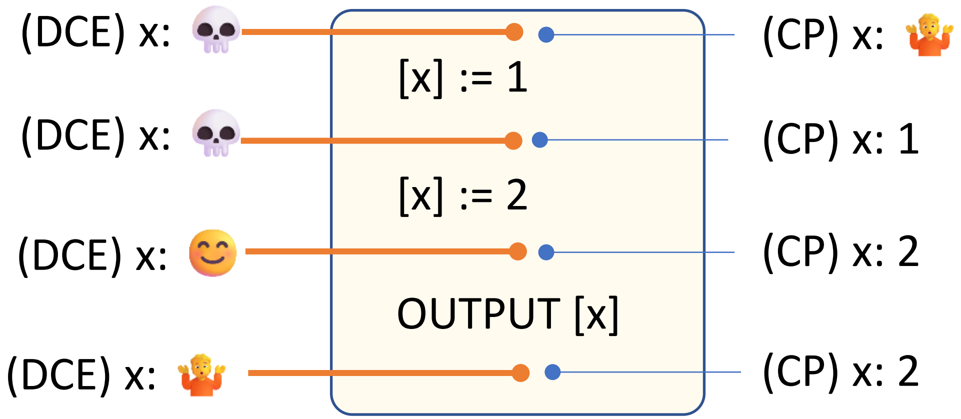
Dataflow analysis

- Intuition
- Concepts
- Dataflow frameworks

Optimization

Consider What Info We Know

Basic Block Optimization



Dead Code Elimination: Guaranteed no future use of this definition (the assignment is not "live")

Constant Propagation: Guaranteed previous static definition of this use

For Dead Code Elimination, definition could be marked

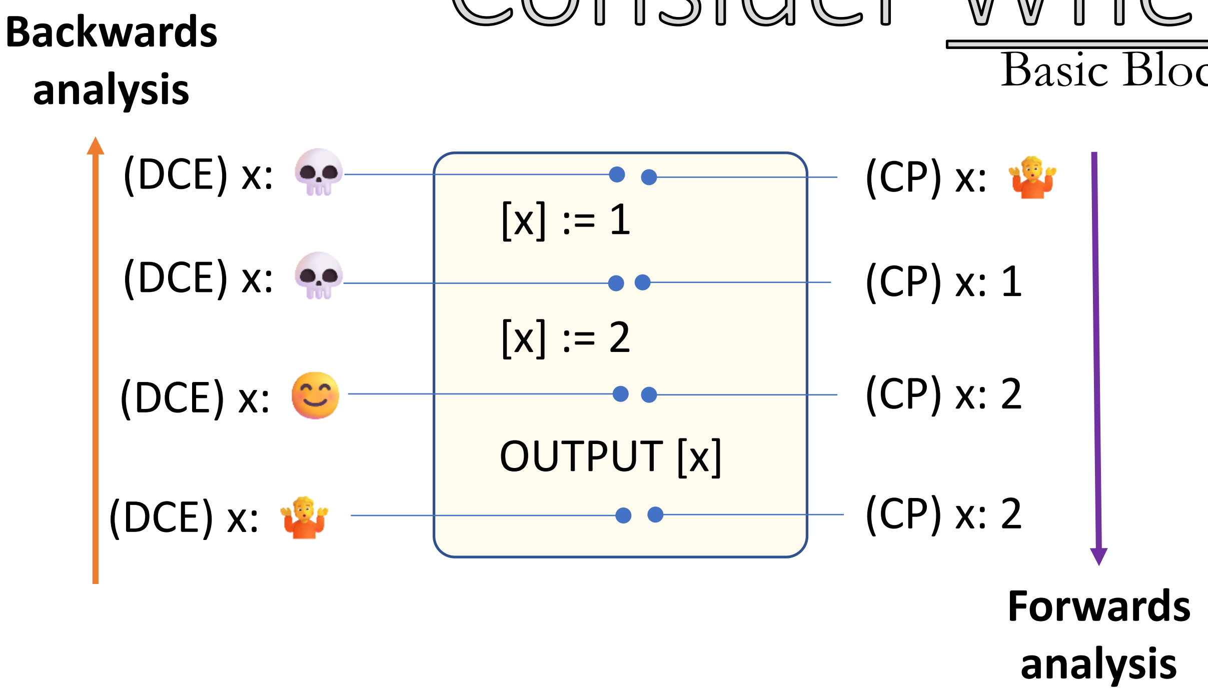
Known Live	Known Dead	Not Enough Info
😊	💀	🤷

For Constant Propagation, use could be marked

Guaranteed Constant	Guaranteed Non-Constant	Not Enough Info
<value>	> 1 value or ⚡	🤷

Consider Where We Learn Info

Basic Block Optimization



Backwards analysis

(DCE) x: 💀

(DCE) x: 💀

(DCE) x: 😊

(DCE) x: 🤷

[x] := 1

[x] := 2

OUTPUT [x]

(CP) x: 🤷

(CP) x: 1

(CP) x: 2

(CP) x: 2

Forwards analysis

Dead Code Elimination: Guaranteed no future use of this definition (the assignment is not "live")

Constant Propagation: Guaranteed previous static definition of this use

For Dead Code Elimination, definition could be marked

Known Live	Known Dead	Not Enough Info
😊	💀	🤷

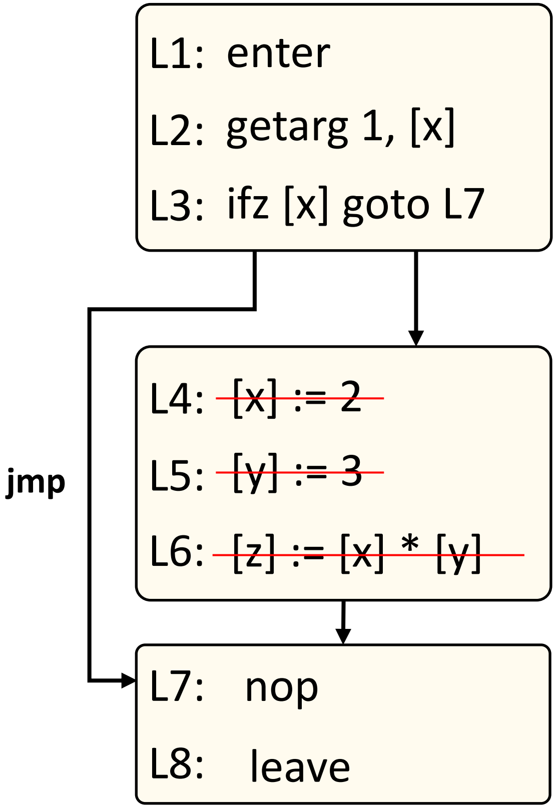
For Constant Propagation, use could be marked

Guaranteed Constant	Guaranteed Non-Constant	Not Enough Info
<value>	> 1 value or ⚡	🤷

Beyond Local Optimization

Dataflow

One possible CFG



L1: enter

L2: getarg 1, [x]

L3: ifz [x] goto L7

L4: ~~[x] := 2~~

L5: ~~[y] := 3~~

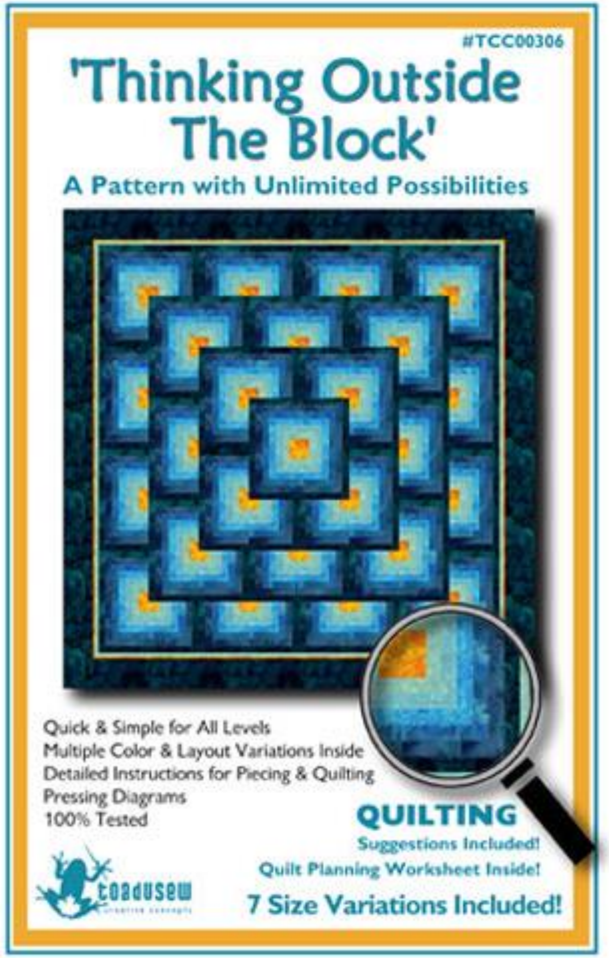
L6: ~~[z] := [x] * [y]~~

L7: nop

L8: leave

L6 is dead!

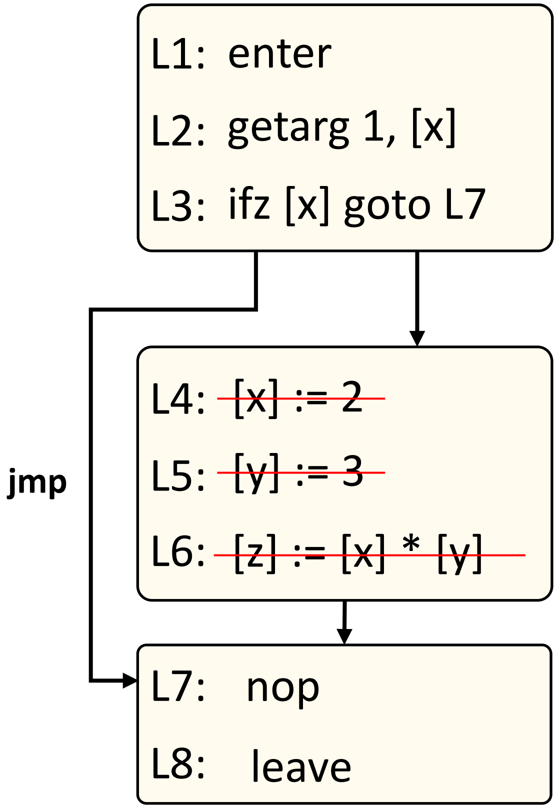
(causes L4 and L5 to be dead)


#TCC00306
'Thinking Outside The Block'
A Pattern with Unlimited Possibilities
Quick & Simple for All Levels
Multiple Color & Layout Variations Inside
Detailed Instructions for Piecing & Quilting
Pressing Diagrams
100% Tested
QUILTING
Suggestions Included!
Quilt Planning Worksheet Inside!
toadusew
7 Size Variations Included!

Beyond Local Optimization

Dataflow

One possible CFG



L1: enter

L2: getarg 1, [x]

L3: ifz [x] goto L7

L4: ~~[x] := 2~~

L5: ~~[y] := 3~~

L6: ~~[z] := [x] * [y]~~

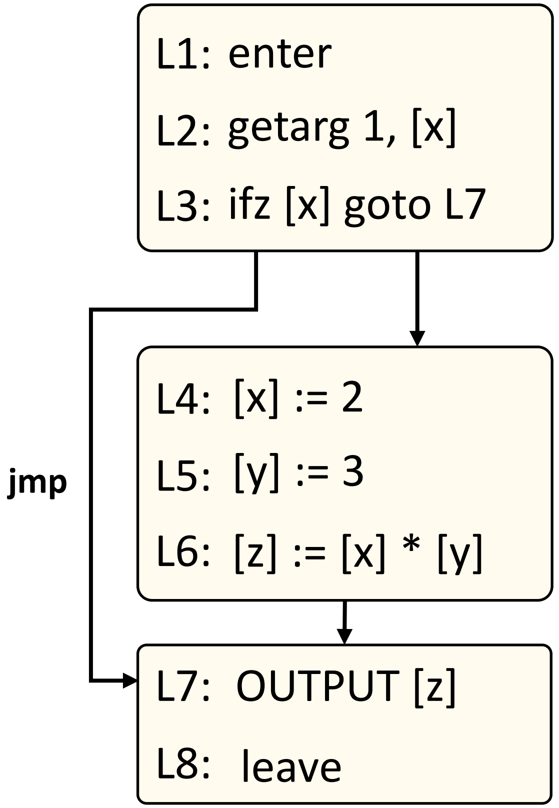
L7: nop

L8: leave

L6 is dead!

(causes L4 and L5 to be dead)

Another possible CFG



L1: enter

L2: getarg 1, [x]

L3: ifz [x] goto L7

L4: [x] := 2

L5: [y] := 3

L6: [z] := [x] * [y]

L7: OUTPUT [z]

L8: leave

L6 is live!

Cannot be removed

Today's Outline

Dataflow

Dataflow analysis

- Intuition
- Concepts
- Dataflow frameworks

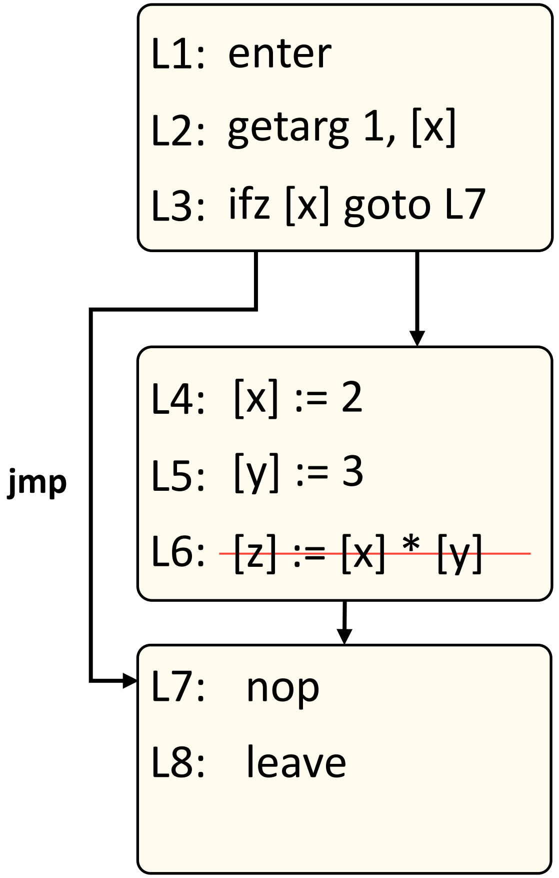
Optimization

Generalizing Dataflow Intuition

Dataflow Intuition

Let’s revisit the example, and ask some leading questions

One possible CFG



L1: enter

L2: getarg 1, [x]

L3: ifz [x] goto L7

L4: [x] := 2

L5: [y] := 3

L6: ~~[z] := [x] * [y]~~

L7: nop

L8: leave

Why is L6 dead? ≡ Why isn’t L6 live?

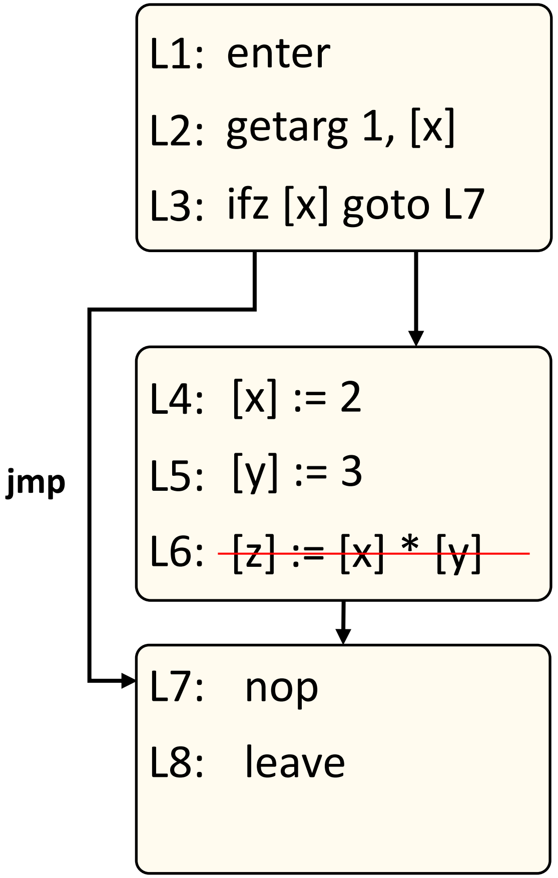
Returning to the scene of the crime

Generalizing Dataflow Intuition

Dataflow Intuition

Let’s revisit the example, and ask some leading questions

<u>**One possible CFG**</u>



```
L1:  enter
L2:  getarg 1, [x]
L3:  ifz [x] goto L7
```

```
L4:  [x] := 2
L5:  [y] := 3
L6:  ~~[z] := [x] * [y]~~
```

```
L7:   nop
L8:   leave
```

Why is L6 dead? ≡ **Why isn’t L6 live?**

The thing defined was no longer useful

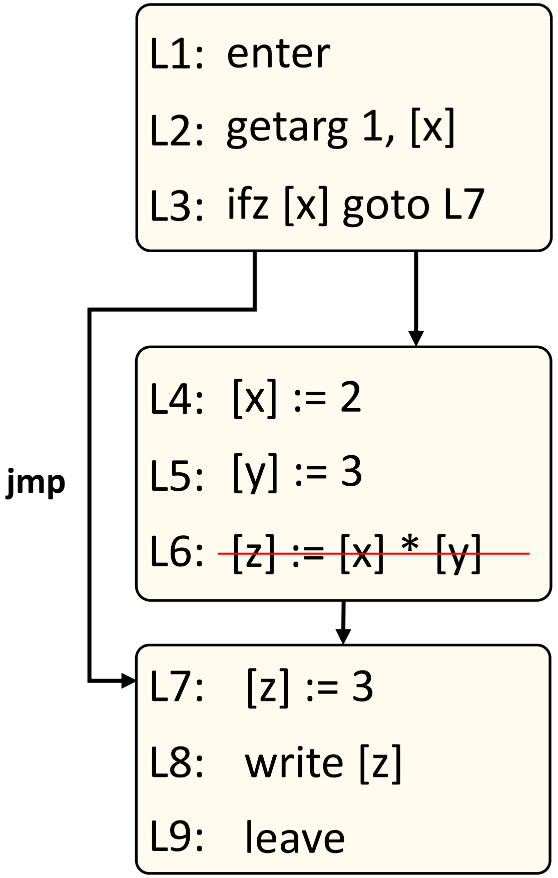
“died of natural causes”

Generalizing Dataflow Intuition

Dataflow Intuition

Let’s revisit the example, and ask some leading questions

One possible CFG



L1: enter

L2: getarg 1, [x]

L3: ifz [x] goto L7

L4: [x] := 2

L5: [y] := 3

L6: ~~[z] := [x] * [y]~~

L7: [z] := 3

L8: write [z]

L9: leave

Why is L6 dead? ≡ Why isn’t L6 live?

The thing defined was no longer useful

“died of natural causes”

The thing defined was redefined before use

“it was killed!”

Need to *gather some facts* to tell if a statement is dead

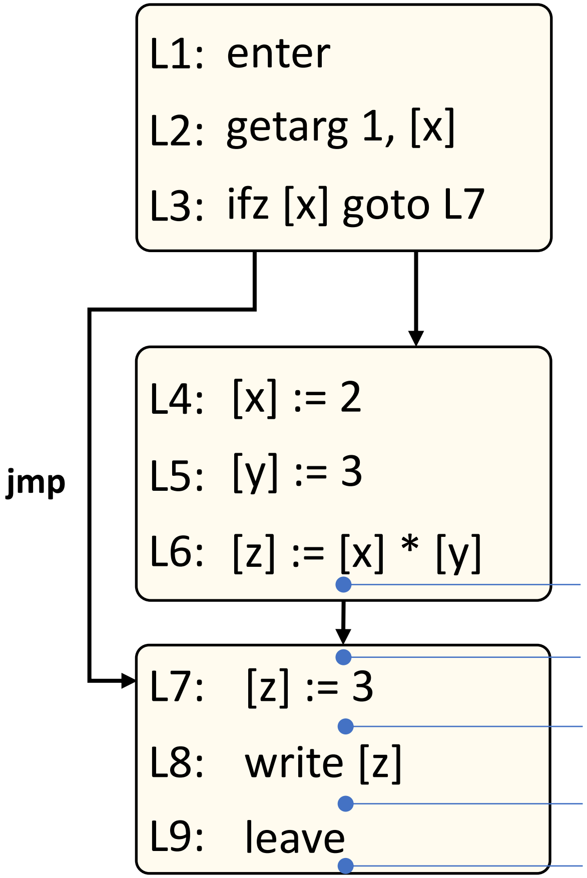
What variables are useful at each program point?

What variables are killed at each program point?

Generalizing Dataflow Intuition

Dataflow Intuition

One possible CFG



```
L1:  enter
L2:  getarg 1, [x]
L3:  ifz [x] goto L7
```

```
L4:  [x] := 2
L5:  [y] := 3
L6:  [z] := [x] * [y]
```

```
L7:   [z] := 3
L8:   write [z]
L9:   leave
```

Need to *gather some facts* to tell if a statement is dead

What things are useful at each program point?

What things are killed at each program point?

x: 💀 , y: 💀 , z: 💀 nothing live at end of block, because nothing live at **entry to successors**

x: 💀 , y: 💀 , z: 💀 Previous definitions of z are not useful, because L7 killed them

x: 💀 , y: 💀 , z: 😊 z is useful, because it's used later on (in L8)

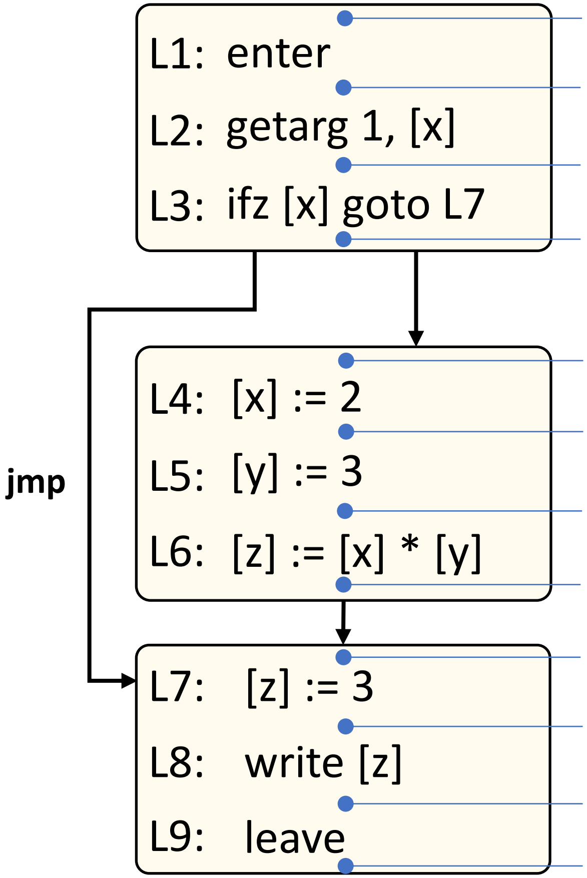
x: 💀 , y: 💀 , z: 💀 nothing is used by leave

x: 💀 , y: 💀 , z: 💀 (no more code to use anything)

Generalizing Dataflow Intuition

Dataflow Intuition

One possible CFG



```
L1:  enter
L2:  getarg 1, [x]
L3:  ifz [x] goto L7

L4:  [x] := 2
L5:  [y] := 3
L6:  [z] := [x] * [y]

L7:   [z] := 3
L8:   write [z]
L9:   leave
```

x: 💀, y: 💀, z: 💀 enter doesn’t use any variables

x: 💀, y: 💀, z: 💀 x is clobbered

x: 😊, y: 💀, z: 💀 x gets used in the predicate

x: 💀, y: 💀, z: 💀 nothing live at end of block, because nothing live at **entry to EITHER successor**

x: 💀, y: 💀, z: 💀 x was killed, previous definitions of x would not be useful past this point

x: 😊, y: 💀, z: 💀 y was killed but a definition of x will still be used later

x: 😊, y: 😊, z: 💀 Both x and y have been found to be used

x: 💀, y: 💀, z: 💀 nothing live at end of block, because nothing live at **entry to successor**

x: 💀, y: 💀, z: 💀 Previous definitions of z are not useful, because L7 killed them

x: 💀, y: 💀, z: 😊 z is useful, because it’s used later on (in L8)

x: 💀, y: 💀, z: 💀 nothing is used by leave

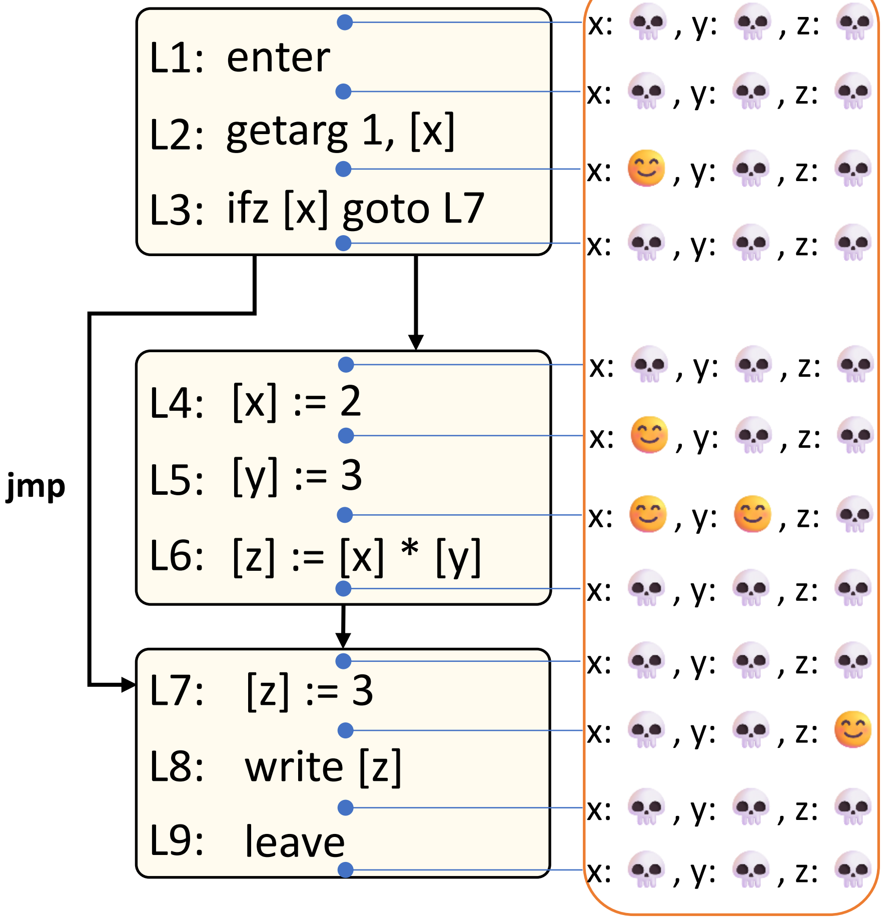
x: 💀, y: 💀, z: 💀 (no more code to use anything)

Generalizing Dataflow Intuition

Dataflow Intuition

One possible CFG

We call these sets “dataflow fact sets”



```
L1:  enter
L2:  getarg 1, [x]
L3:  ifz [x] goto L7

L4:  [x] := 2
L5:  [y] := 3
L6:  [z] := [x] * [y]

L7:   [z] := 3
L8:   write [z]
L9:   leave
```

x: 💀, y: 💀, z: 💀

x: 💀, y: 💀, z: 💀

x: 😊, y: 💀, z: 💀

x: 💀, y: 💀, z: 💀

x: 💀, y: 💀, z: 💀

x: 😊, y: 💀, z: 💀

x: 😊, y: 😊, z: 💀

x: 💀, y: 💀, z: 💀

x: 💀, y: 💀, z: 💀

x: 💀, y: 💀, z: 😊

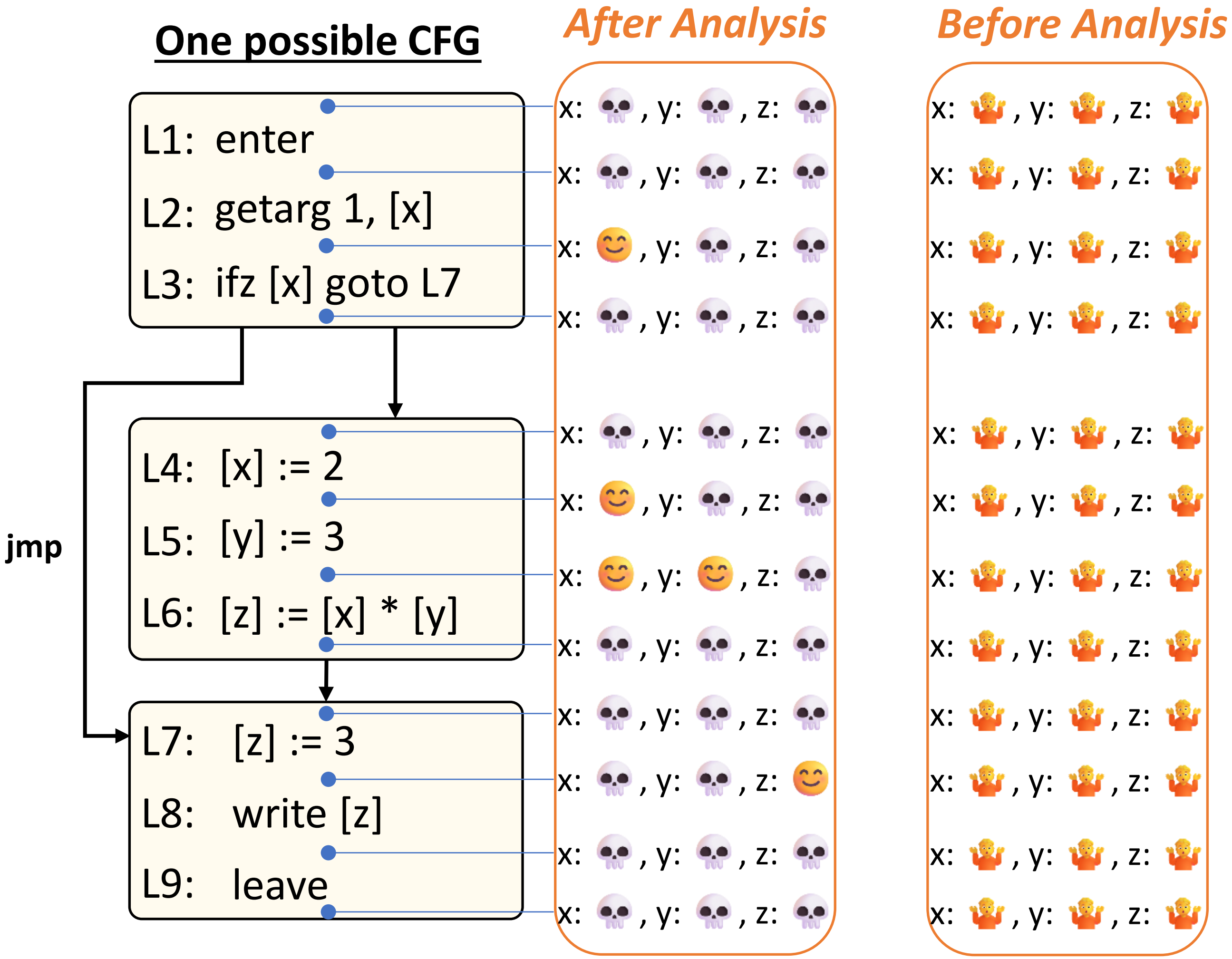
x: 💀, y: 💀, z: 💀

x: 💀, y: 💀, z: 💀

Initializing Fact Sets

Dataflow Intuition

Technically, we should start all fact sets as “Not enough info” (🤷). This will matter later



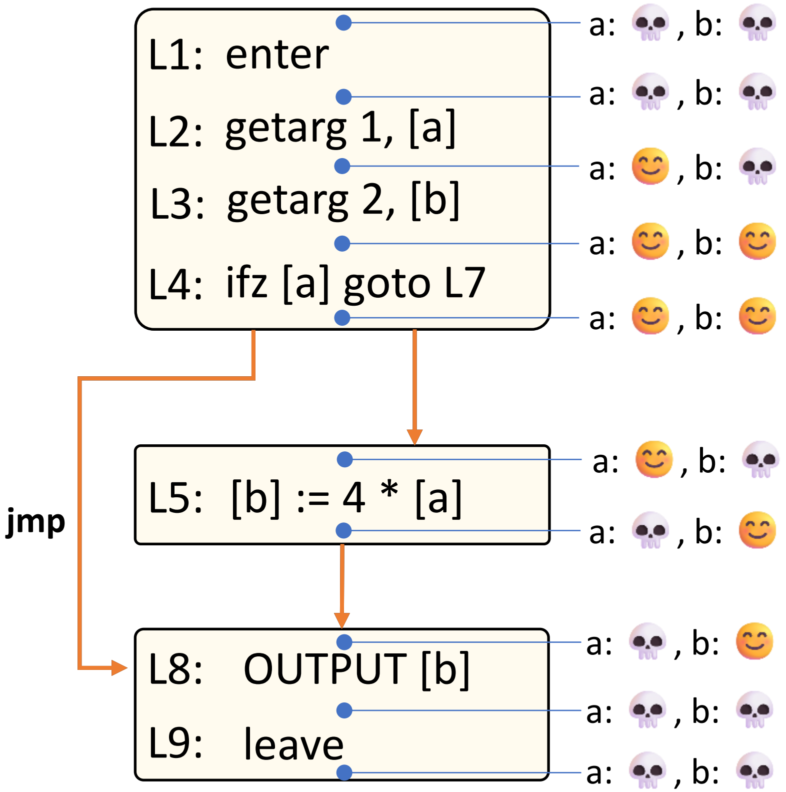
One possible CFG

```
L1:  enter
L2:  getarg 1, [x]
L3:  ifz [x] goto L7

L4:  [x] := 2
L5:  [y] := 3
L6:  [z] := [x] * [y]

L7:   [z] := 3
L8:   write [z]
L9:   leave
```

After Analysis	*Before Analysis*
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 😊, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 😊, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 😊, y: 😊, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 😊	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷
x: 💀, y: 💀, z: 💀	x: 🤷, y: 🤷, z: 🤷

Merging Fact Sets

Dataflow Intuition



Fact sets may be different when multiple successors/predecessors join

- Need to merge incoming fact sets

Merge as conservatively as possible

- Don’t do anything without a guarantee!
- Plan for all possible flows

Example: is L3 live? *(consider both block paths)*

- L3 definition clobbered on the fallthrough branch (at L5)
- L3 definition not clobbered on the jump branch

Today’s Outline

Dataflow

Dataflow analysis

- Intuition
- Dataflow frameworks
- Abstract Interpretation

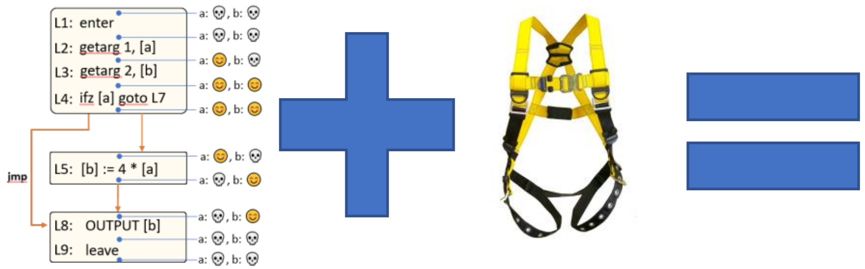
Optimization

Harnessing Commonalities of Dataflow Analyses

Dataflow Frameworks

Basic algorithms for many dataflow analyses follow a common template with minor variations

- Idea: restate each analysis in terms of its variations
- Profit: reuse the same algorithm to get results



```
L1:  enter
L2:  getarg 1, [a]
L3:  getarg 2, [b]
L4:  ifz [a] goto L7

L5:  [b] := 4 * [a]

L8:   OUTPUT [b]
L9:   leave
```

$$$

Harnessing Commonalities of Dataflow Analyses

Dataflow Frameworks

Basic algorithms for many dataflow analyses follow a common template with minor variations

- Idea: restate each analysis in terms of its variations
- Profit: reuse the same algorithm to get results

Variations

- What information is tracked
- How fact sets are merged
- The direction of the analysis

Templated Information Tracking

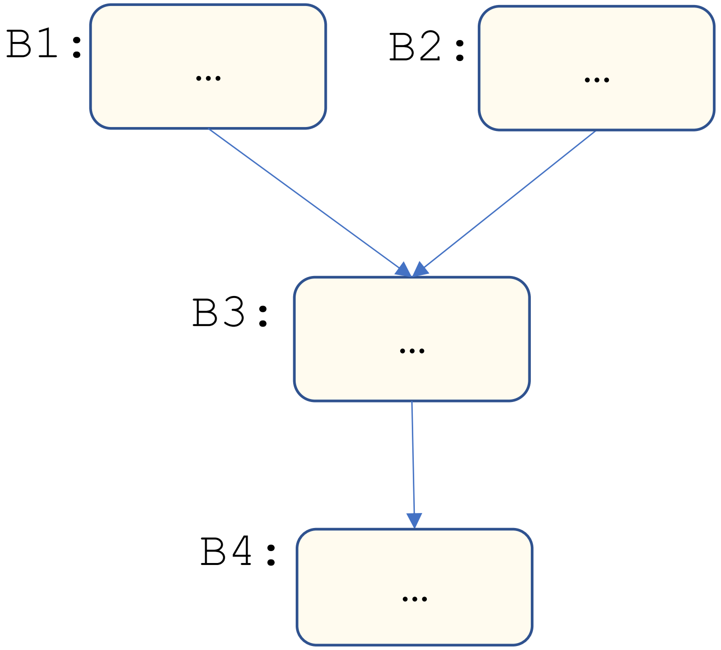
Dataflow Frameworks

Framework tracks the “interplay between data” at basic blocks boundaries

For a given basic block b:

- IN(b): facts true on entry to b
- OUT(b): facts true on exit from b
- GEN(b): facts created in b
- KILL(b): facts removed in b

For a *backwards analysis*
IN is at the bottom of the block
OUT is at the top of the block



$$\mathrm{IN}(B) = \bigcup_{p \text{ in } pred(b)} \mathrm{OUT}(p)$$

$$\mathrm{OUT}(b) = \mathrm{GEN}(b) \cup (\mathrm{IN}(b) - \mathrm{KILL}(b))$$

Dataflow Sets: Example

IN(b): facts true on entry to b

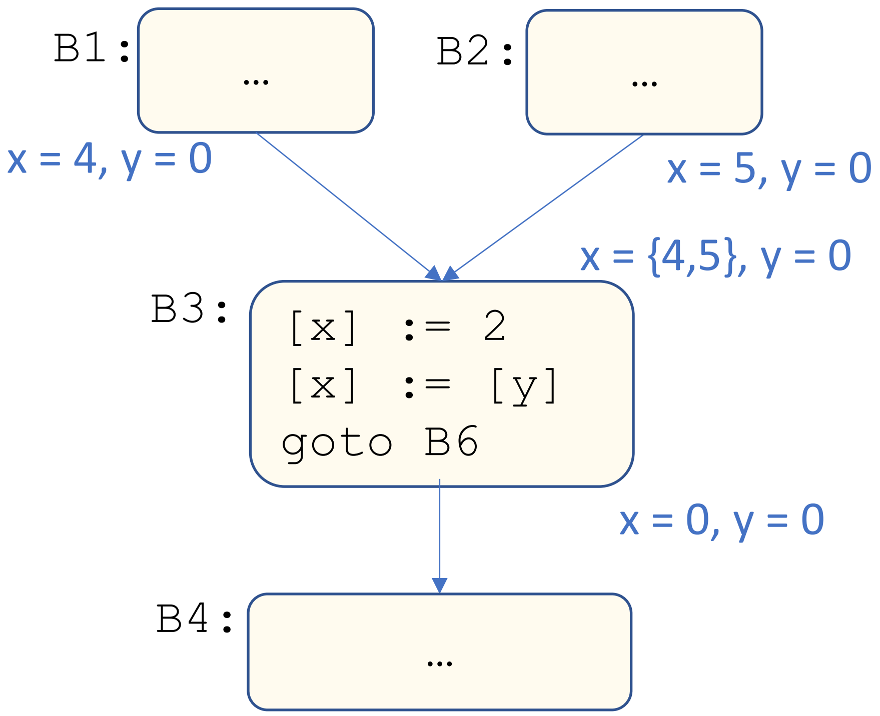
OUT(b): facts true on exit from b

GEN(b): facts created in b

KILL(b): facts removed in b

$$\mathrm{IN}(B) = \bigcup_{p \text{ in } pred(b)} \mathrm{OUT}(p)$$

$$\mathrm{OUT}(b) = \mathrm{GEN}(b) \cup (\mathrm{IN}(b) - \mathrm{KILL}(b))$$



Benefits of the Framework

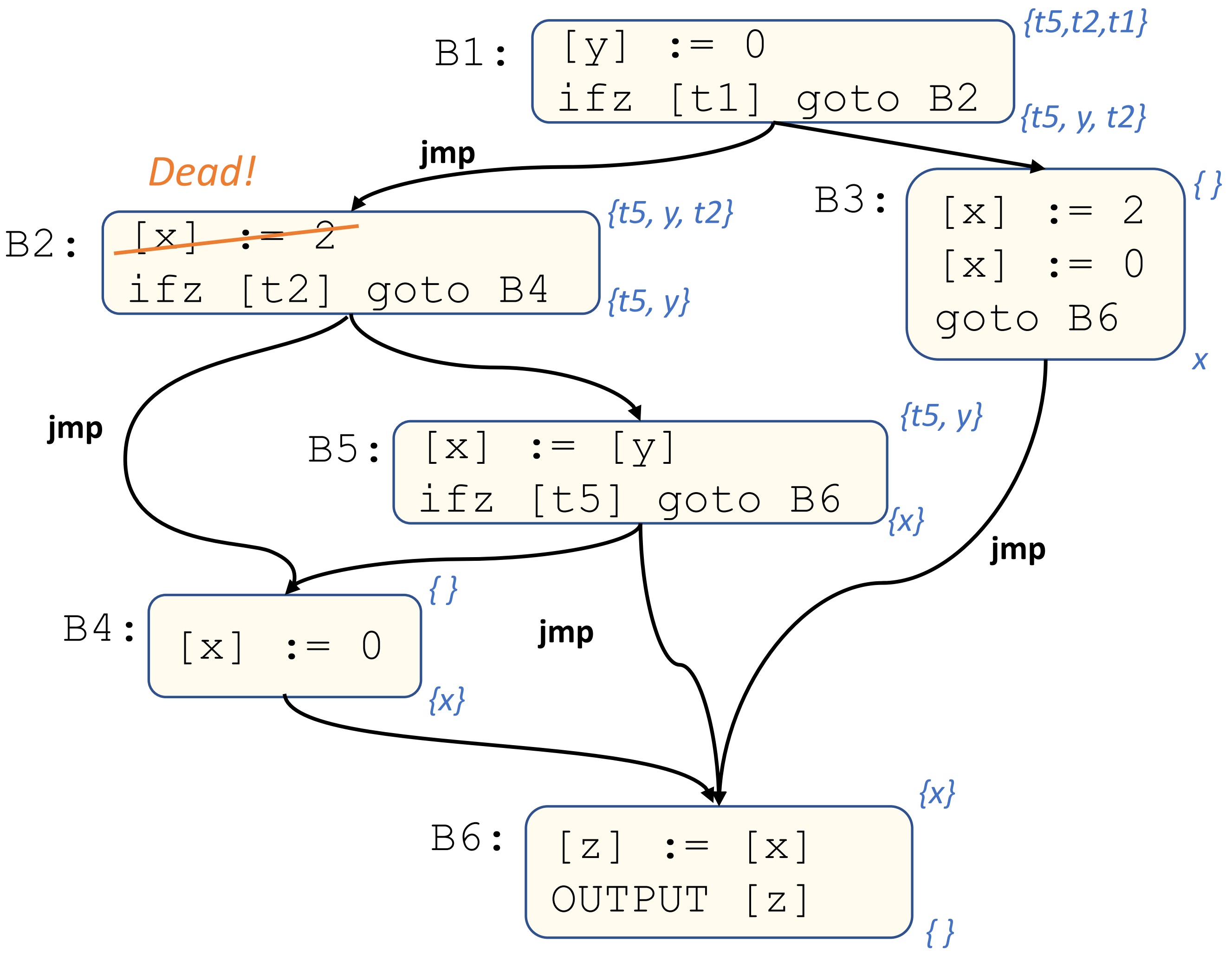
Dataflow Frameworks

When set up properly…

- Safety of the analysis is guaranteed
- Termination of the analysis is guaranteed
- Order of analysis (which block you process) is unimportant

Compute Live Variables

Dataflow: Formalization - Example



What values are live at B6?

Example Analyses

Dataflow: Formalization

Let’s do some examples in this light

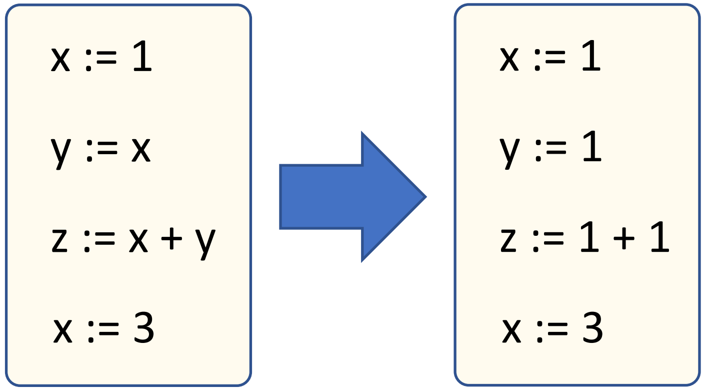
✓ A slightly bigger dead code elimination example

- Constant propagation
  - Recall: replace a variable with it’s known constant value
  - Forward analysis
  - Fact sets: variable to (sets of) known values

Refresh Constant/Copy Propagation

Copy Propagation

- Replace RHS of simple assigns with value of assign (if known)
- Forward analysis

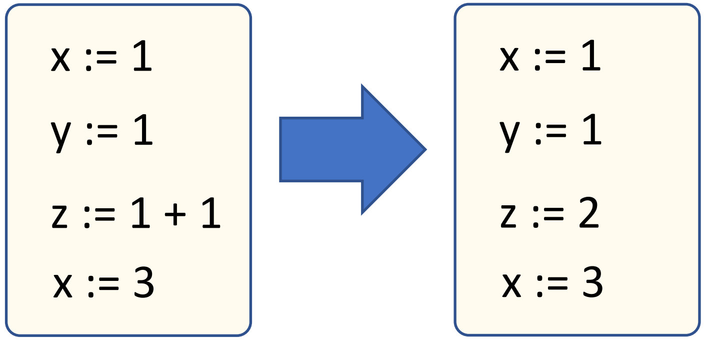


```
x := 1
y := x
z := x + y
x := 3
```

```
x := 1
y := 1
z := 1 + 1
x := 3
```

Constant folding

- Replace constant RHS expressions with value
- Traversal order isn’t important

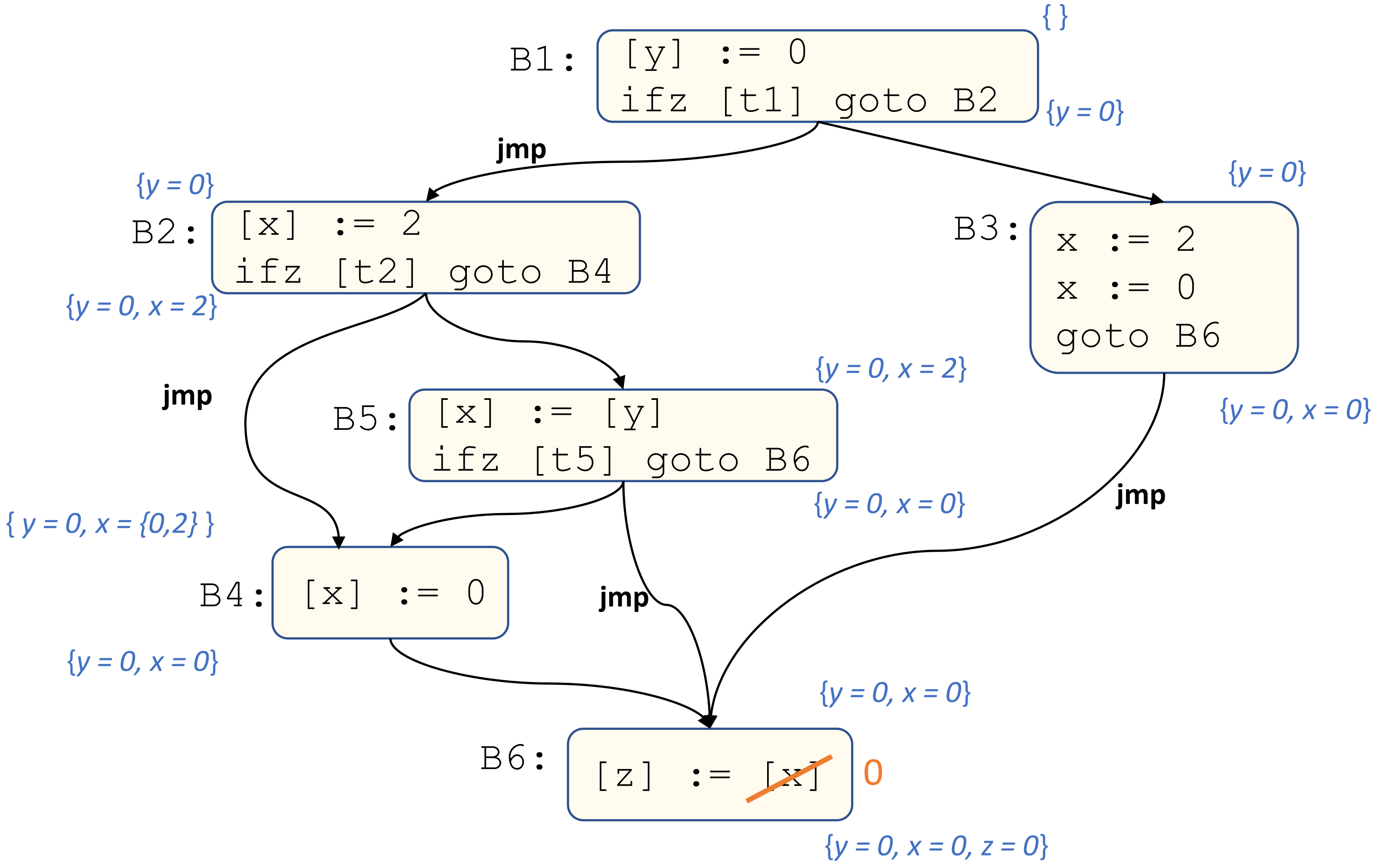


```
x := 1
y := 1
z := 1 + 1
x := 3
```

```
x := 1
y := 1
z := 2
x := 3
```

Example Constant Propagation

Dataflow: Formalization - Example



What values can x take on at B6?

Handling Practical Data Abstractions

Global Dataflow: Formalization

Global variables

- We only have visibility into 1 procedure
- Be conservative about the effect of other procedures
    - Reset fact sets across a call
    - Consider global variables live at function end

Analysis Termination

Dataflow: Formalization

In the previous examples, we completed in one pass over the CFG

- This won’t always be the case, due to a fundamental construct…

Loops

Dataflow: Formalization

Loops complicate dataflow analysis

- Create cyclic dependencies
- Complicate fact sets

Oh bröther, you might have some lööps

Loops: Dependency cycles

Dataflow: Formalization

Solution: Saturate fact sets

- Start sets "TBD" (🤷) value
- Run the algorithm until sets don't change

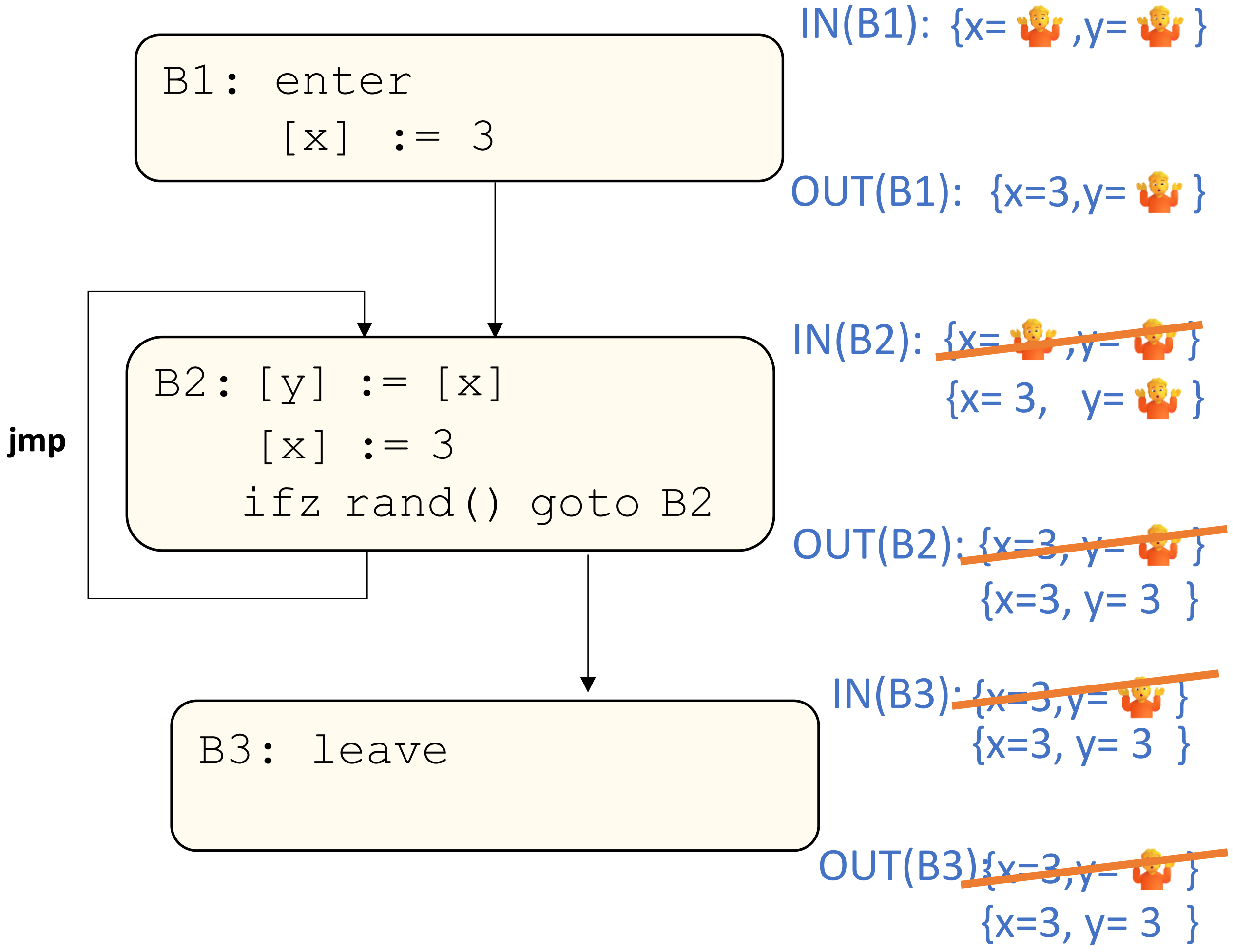
We've seen the saturation approach before

- (FIRST and FOLLOW sets)

Constant propagation

IN(B2) requires knowing OUT(B2)

OUT(B2) requires knowing IN(B2)



```
B1: enter
     [x] := 3
```

```
B2: [y] := [x]
     [x] := 3
    ifz rand() goto B2
```

```
B3: leave
```

IN(B1): {x= 🤷 ,y= 🤷 }

OUT(B1): {x=3,y= 🤷 }

IN(B2): ~~{x= 🤷 ,y= 🤷 }~~
{x= 3, y= 🤷 }

OUT(B2): ~~{x=3, y= 🤷 }~~
{x=3, y= 3 }

IN(B3): ~~{x=3,y= 🤷 }~~
{x=3, y= 3 }

OUT(B3) ~~{x=3,y= 🤷 }~~
{x=3, y= 3 }

Summary

Underview

Covered some key optimization concepts

- Inter-block (global) analysis
- Dataflow frameworks:
  - Define fact sets and how they interact

Next Time – Static Single Assignment (SSA)

- A program form that eases and enhances optimization