CMPT 745
Software Engineering

Dynamic Analysis

Nick Sumner
wsumner@sfu.ca

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?
  - Where am I spending time?

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?
  - Where am I spending time?
  - Where might I parallelize?

Dynamic Analysis

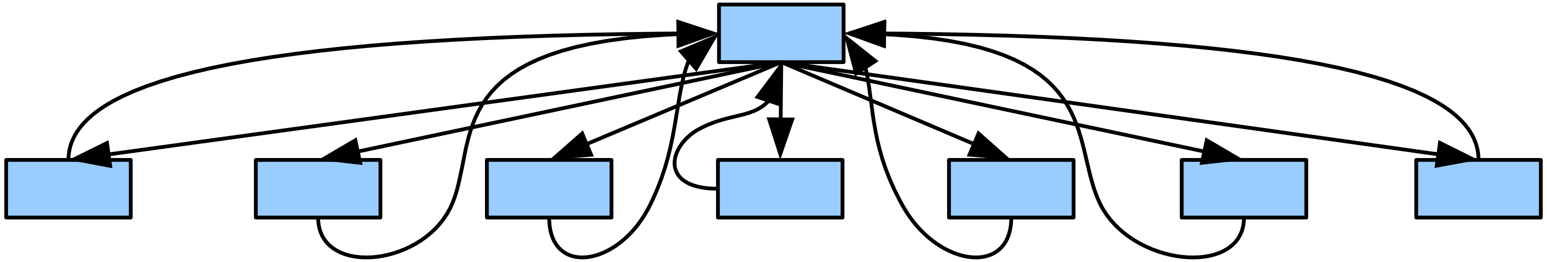
- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?
  - Where am I spending time?
  - Where might I parallelize?
  - Tolerate errors

Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?
  - Where am I spending time?
  - Where might I parallelize?
  - Tolerate errors
  - Manage memory / resources.

e.g. Reverse Engineering

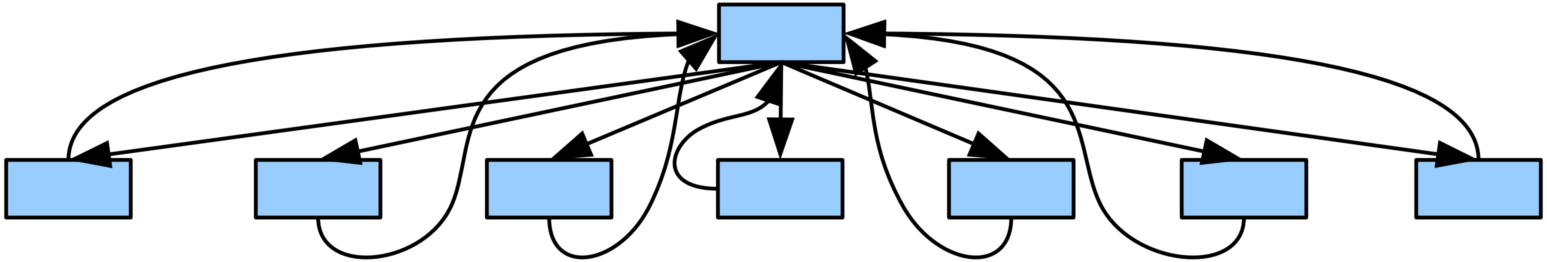
Static CFG (from e.g. Apple Fairplay):



This is the result of a control flow flattening obfuscaton.
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]

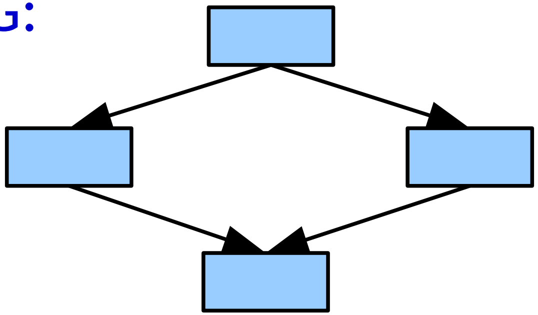
e.g. Reverse Engineering

Static CFG (from e.g. Apple Fairplay):



This is the result of a control flow flattening obfuscaton.
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]

Dynamically Simplified CFG:



How?

- Can record the execution

How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online

How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself

How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself

- Can do both!
  - Lightweight *recording*
  - Instrument a *replayed* instance of the execution

How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself

- Can do both!
  - Lightweight *recording*
  - Instrument a *replayed* instance of the execution

Some analyses only make sense *online*.
Why?

Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*

Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*

Profiling is a common dynamic analysis!

Simple Idea: Basic Block Profiling

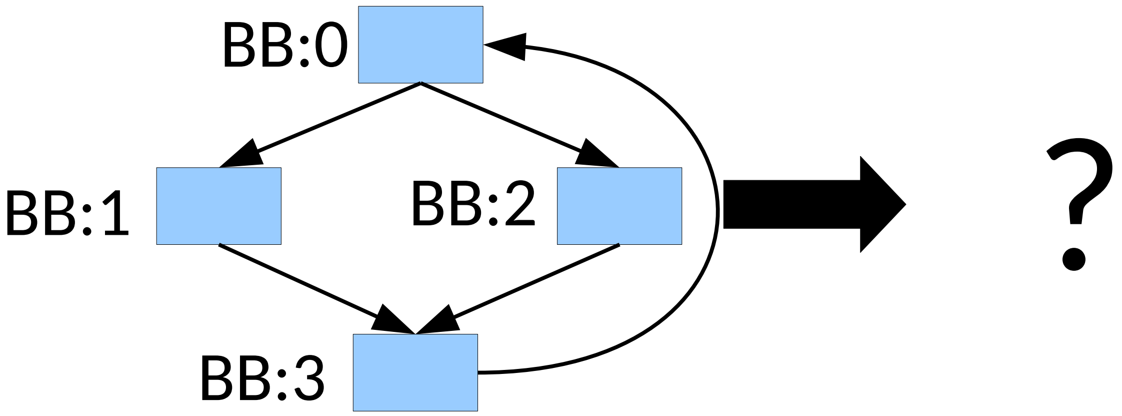
Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?

Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

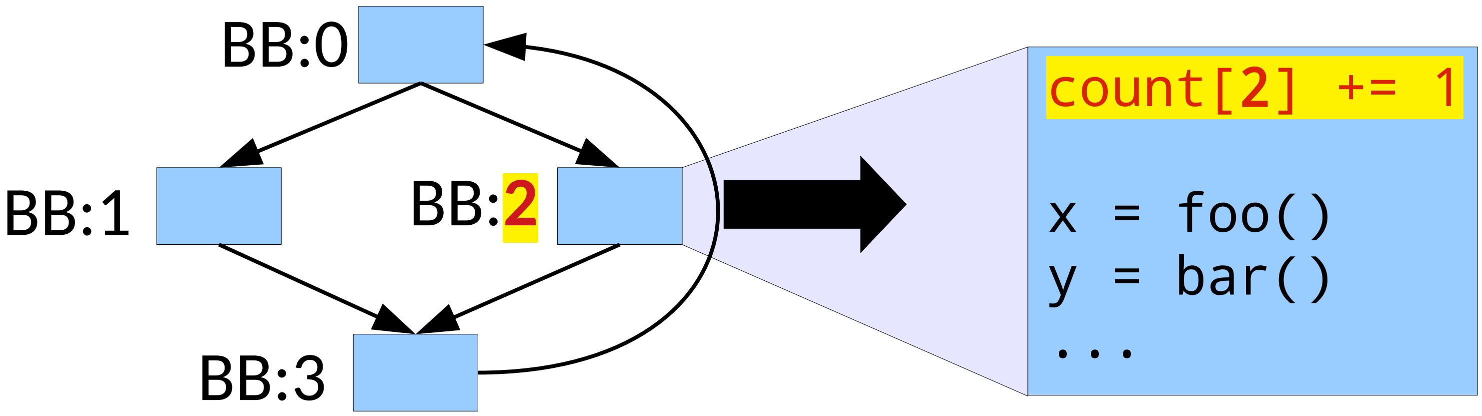
- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?



Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

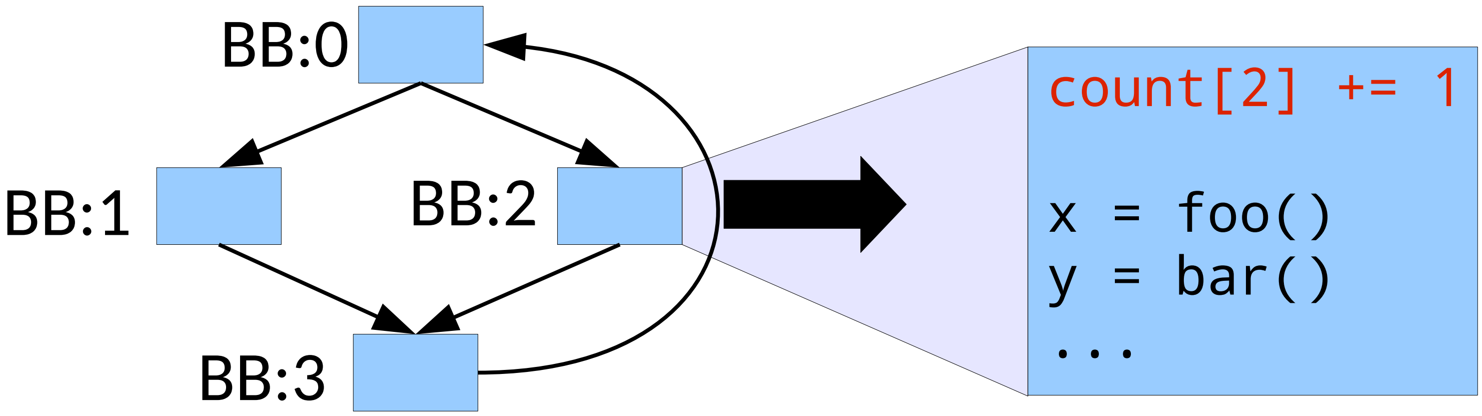
- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?



Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?



Start:

```
for i in BBs:
  count[i] = 0
```

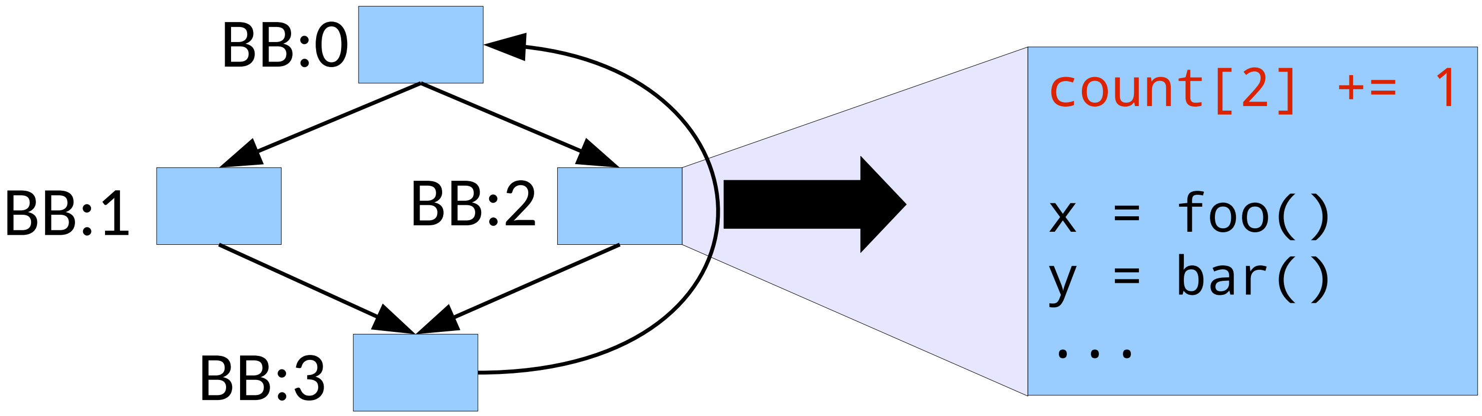
End:

```
for i in BBs:
  print(count[i])
```

Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?



Start:

```
for i in BBs:
  count[i] = 0
```

End:

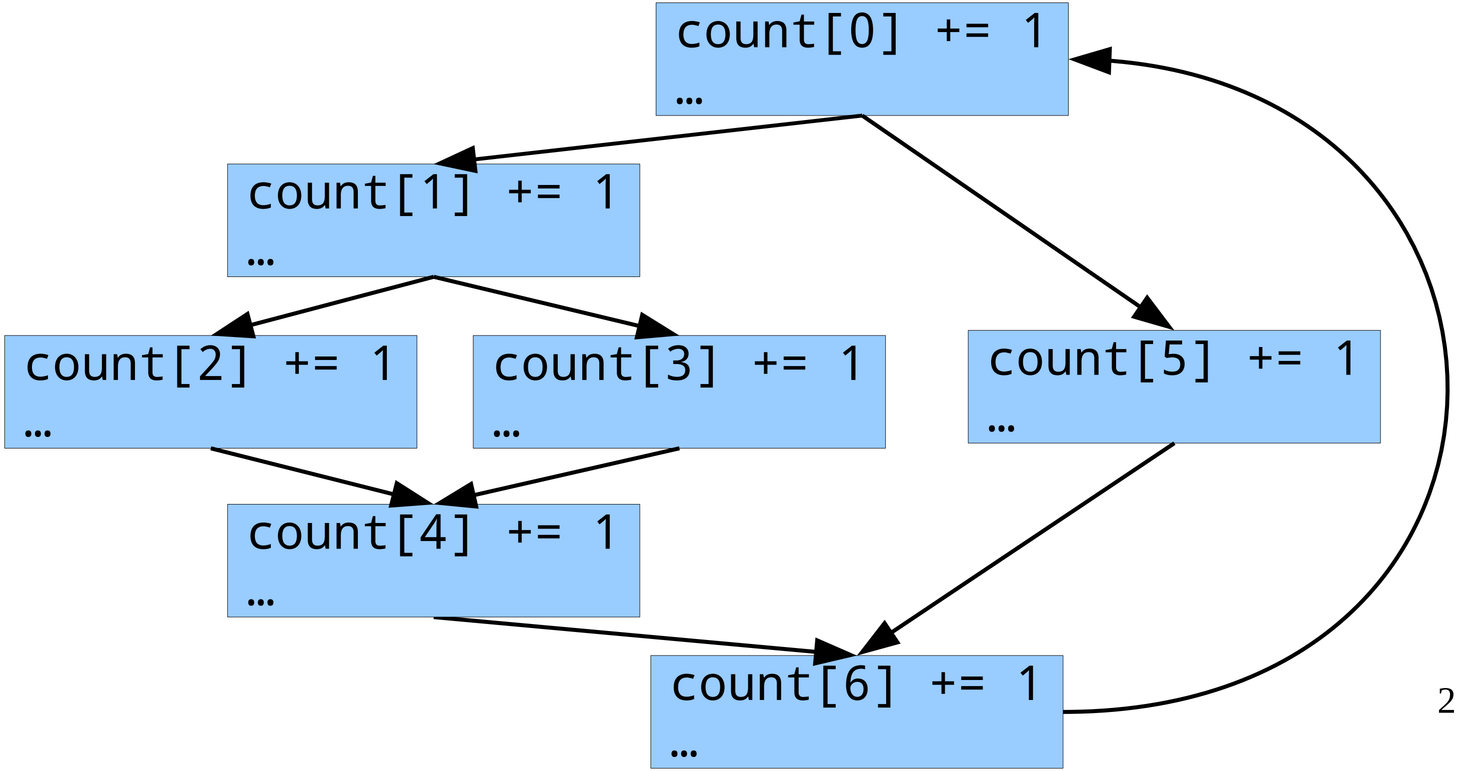
```
for i in BBs:
  print(count[i])
```

Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

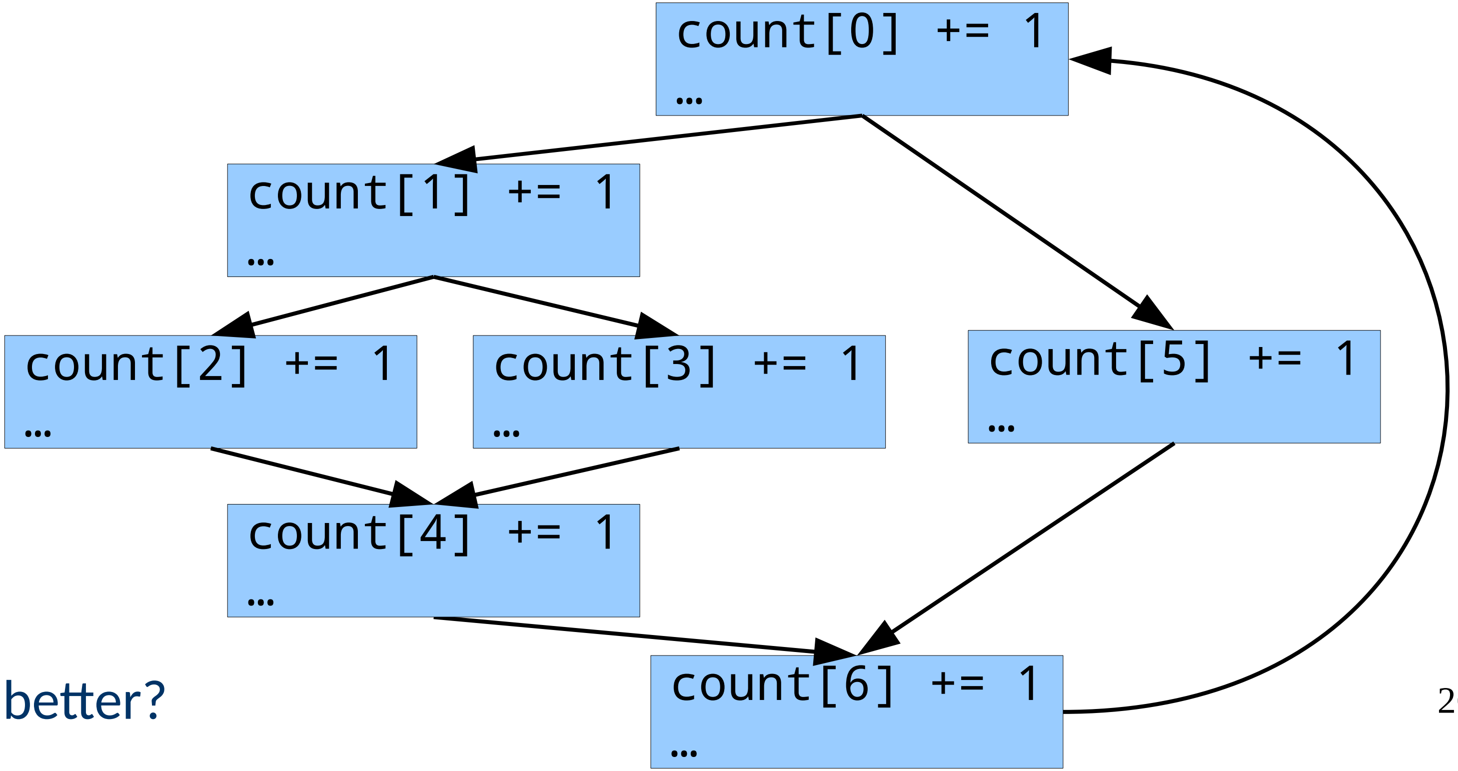
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool



Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

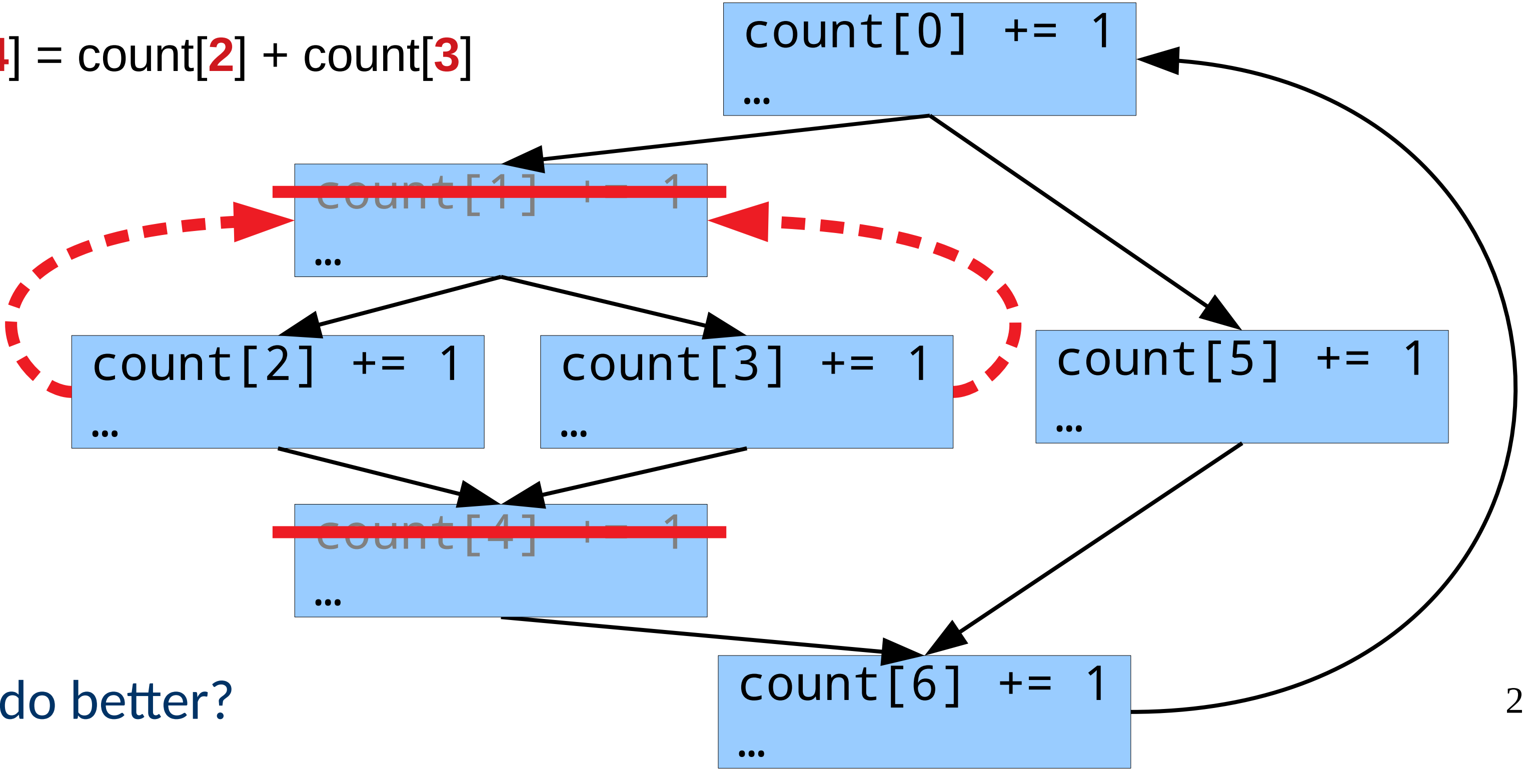


  - Can we do better?

Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

count[1] = count[4] = count[2] + count[3]

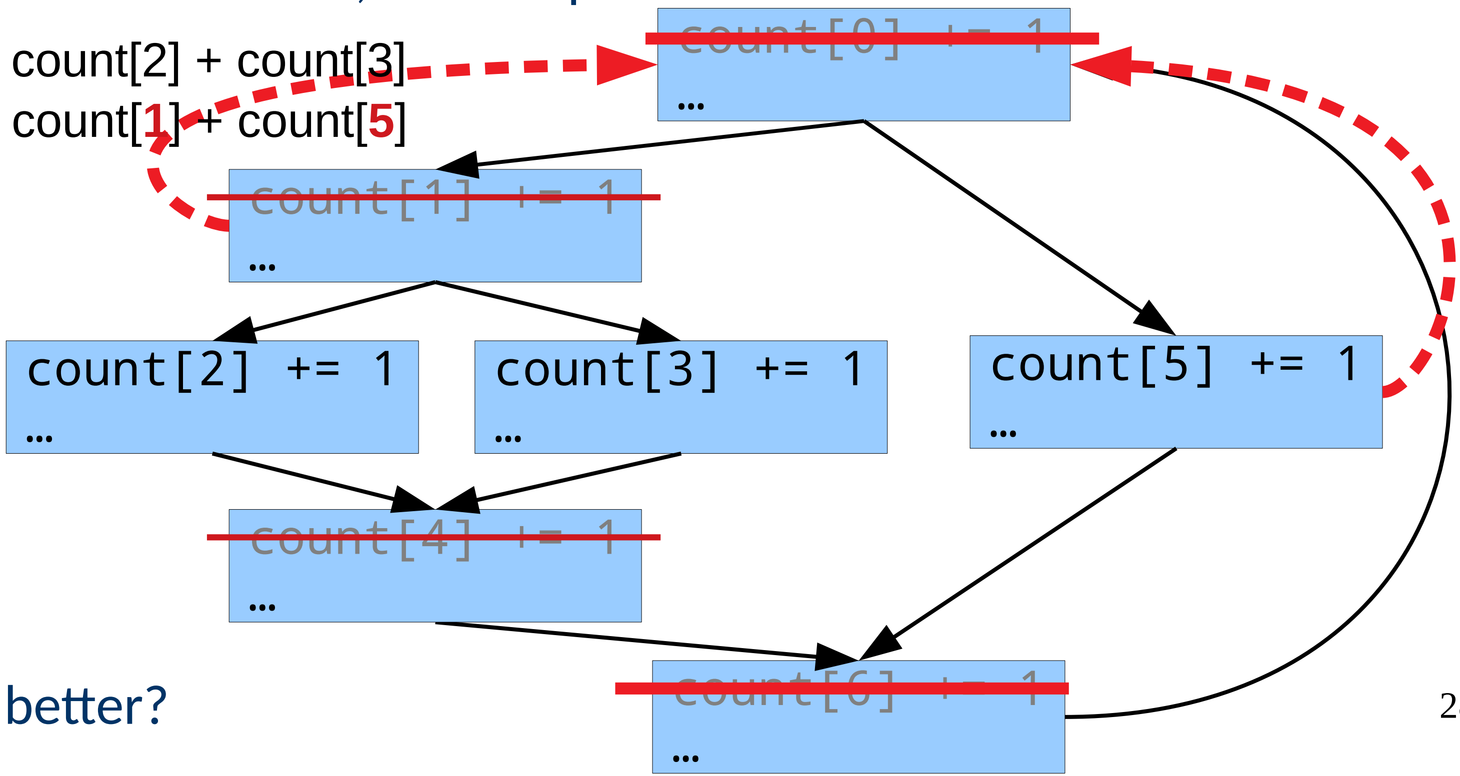


  - Can we do better?

Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

count[1] = count[4] = count[2] + count[3]
count[0] = count[6] = count[1] + count[5]

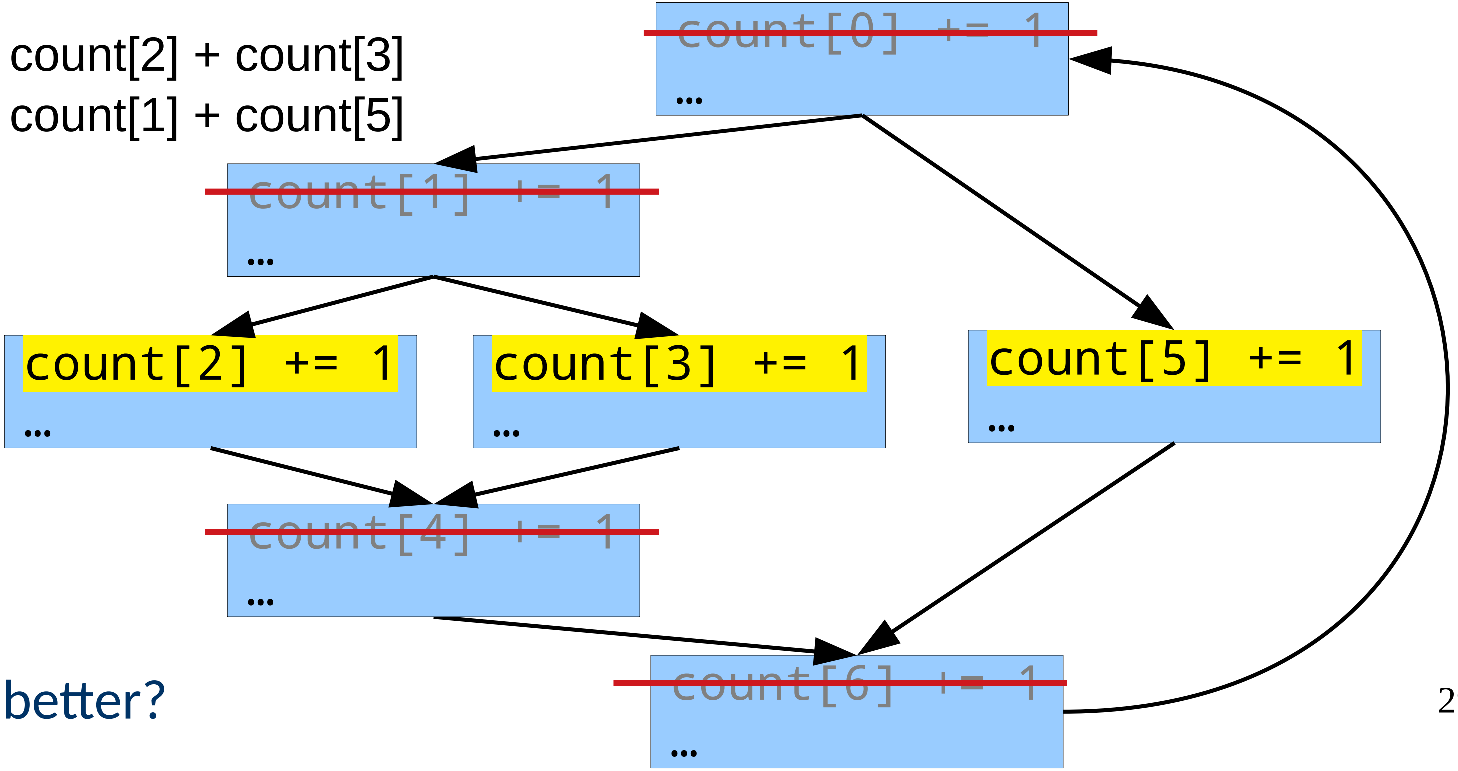


  - Can we do better?

Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

count[1] = count[4] = count[2] + count[3]
count[0] = count[6] = count[1] + count[5]



  - Can we do better?

Efficiency Tactics

- Abstraction

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
- Thread local analysis

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
- Thread local analysis
- Inference

How / When to Instrument

- Source / IR Instrumentation
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

How / When to Instrument

- Source / IR Instrumentation
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

- Static Binary Rewriting
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior

How / When to Instrument

- Source / IR Instrumentation
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build
- Static Binary Rewriting
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior
- Dynamic Binary Instrumentation
  - Valgrind, Pin, Qemu (& other Vms)
  - Can adapt at runtime, but less info than IR

How / When to Instrument

- Source / IR Instrumentation
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build
- Static Binary Rewriting
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior
- Dynamic Binary Instrumentation
  - Valgrind, Pin, Qemu (& other Vms)
  - Can adapt at runtime, but less info than IR
- *Black Box* Dynamic Analysis

Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) Instrumentation
   - Add code to the program for data collection/analysis

Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) Instrumentation
   – Add code to the program for data collection/analysis
2) Execution
   – Run the program an analyze its actual behavior

Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) Instrumentation
   – Add code to the program for data collection/analysis
2) Execution
   – Run the program an analyze its actual behavior
3) (Optional) Postmortem Analysis
   – Perform any analysis that can be deferred after termination

Phases of Dynamic Analysis

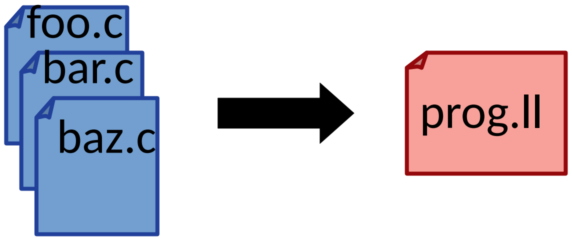
In general, 2-3 phases occur:

1) Instrumentation
   - Add code to the program for data collection/analysis
2) Execution
   - Run the program an analyze its actual behavior
3) (Optional) Postmortem Analysis
   - Perform any analysis that can be deferred after termination

Very, ***very*** common mistake to mix 1 & 2.

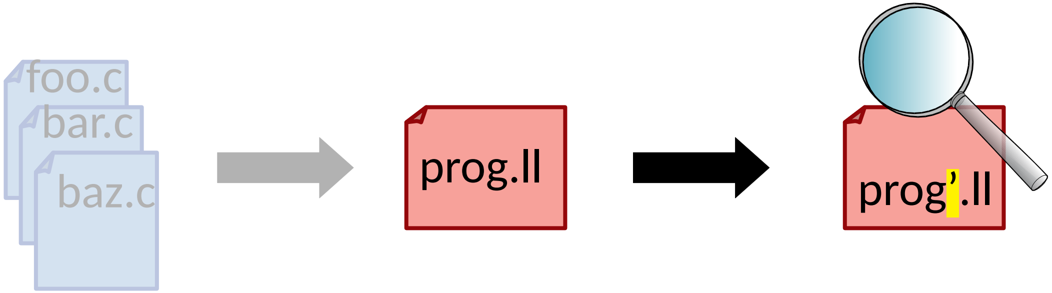
Static Instrumentation

1) Compile whole program to IR



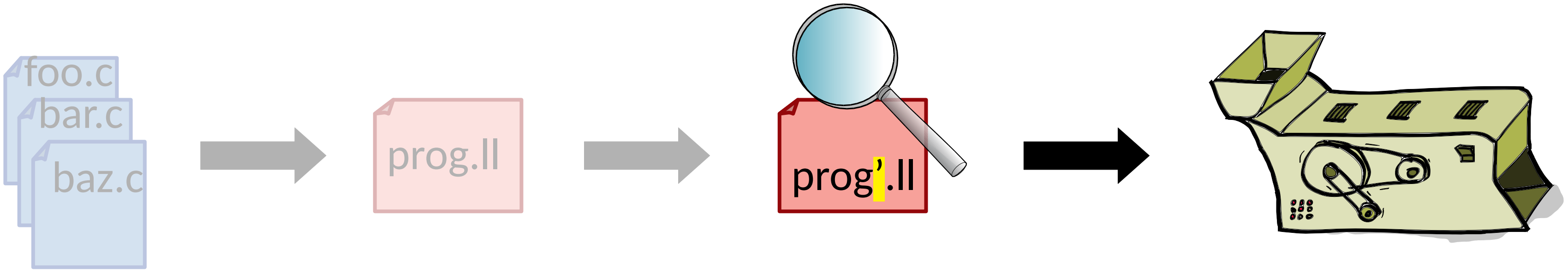
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR



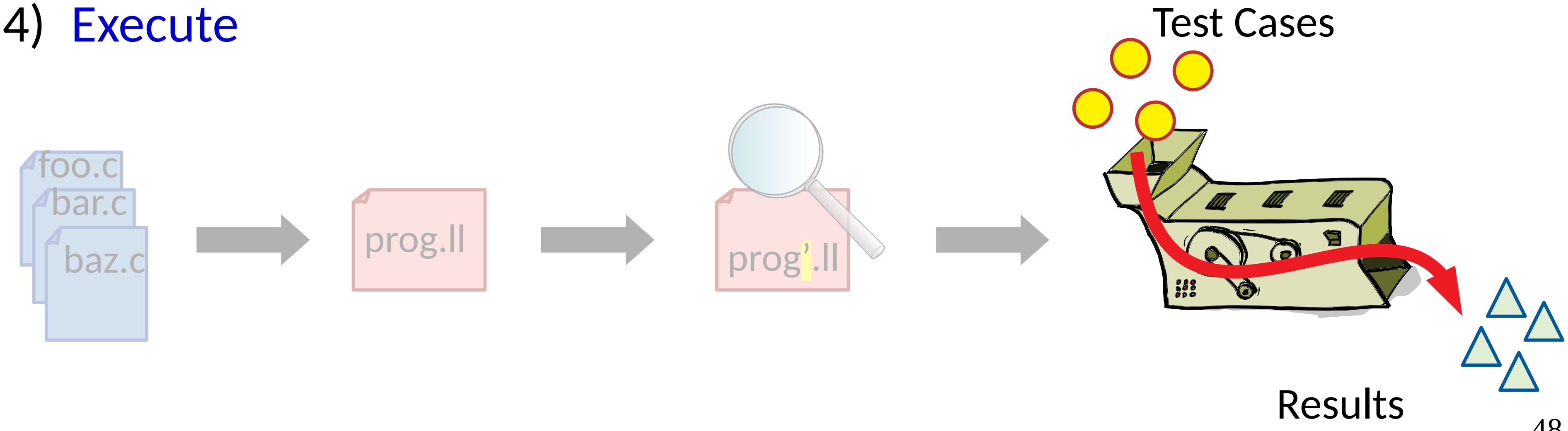
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
3) Generate new program that performs analysis



Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
3) Generate new program that performs analysis
4) Execute



Dynamic Binary Instrumentation (DBI)

1) Compile program as usual
2) Run program under analysis framework

(Valgrind, PIN, Qemu, etc)

```
valgrind --tool=memcheck ./myBuggyProgram
```

Dynamic Binary Instrumentation (DBI)

1) Compile program as usual
2) Run program under analysis framework

   (Valgrind, PIN, Qemu, etc)

3) Instrument & execute in same command:
   - Fetch & instrument each basic block individually
   - Execute each basic block

```
valgrind --tool=memcheck ./myBuggyProgram
```

Example: Test Case Reduction

Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough

Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior

Testing and Dynamic Analysis

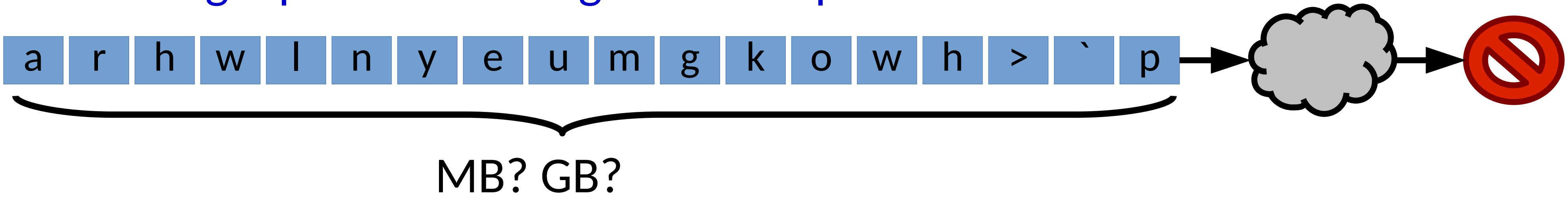
- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior
- Intuitively, even just testing is a dynamic analysis
  - It requires no transformation
  - The result is just the success or failure of tests

Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior
- Intuitively, even just testing is a dynamic analysis
  - It requires no transformation
  - The result is just the success or failure of tests
- But even that is interesting to consider….

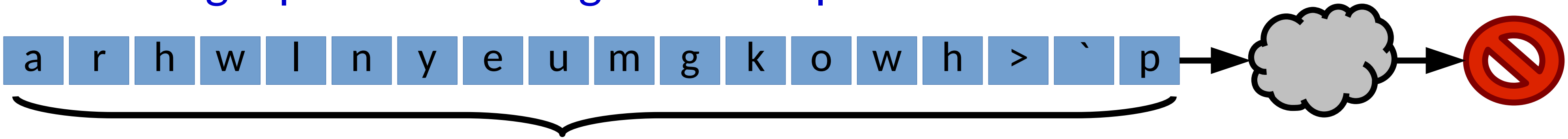
Bug reports are problematic

- Failing inputs can be large and complex



Bug reports are problematic

- Failing inputs can be large and complex

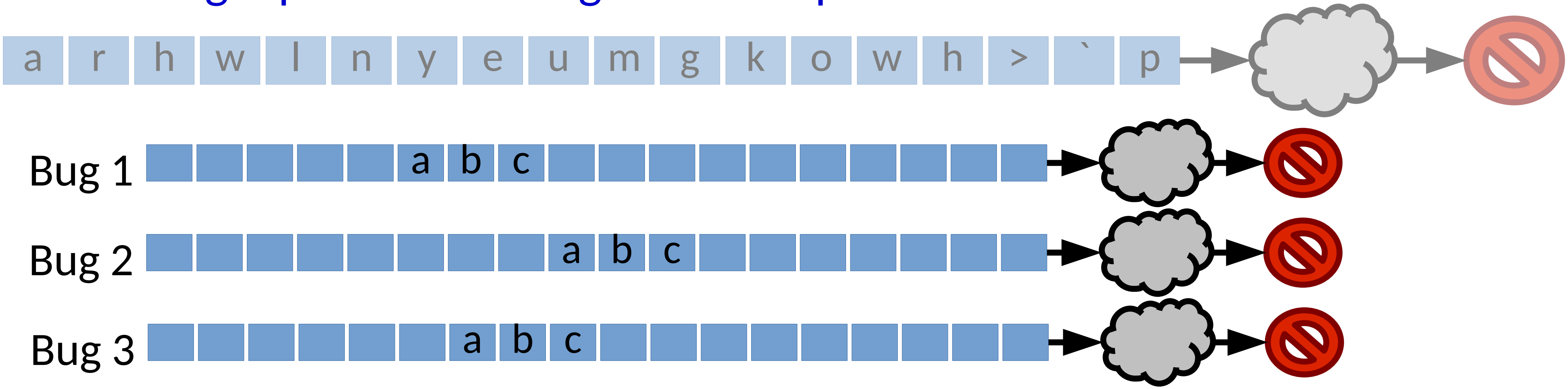


MB? GB?

What is relevant and essential to the bug?

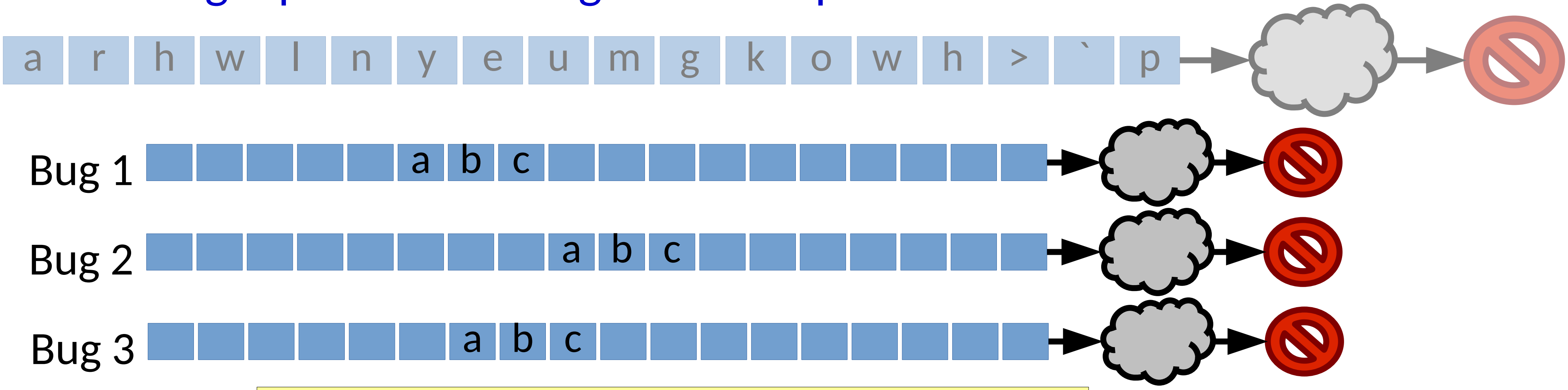
Bug reports are problematic

- Failing inputs can be large and complex



Bug reports are problematic

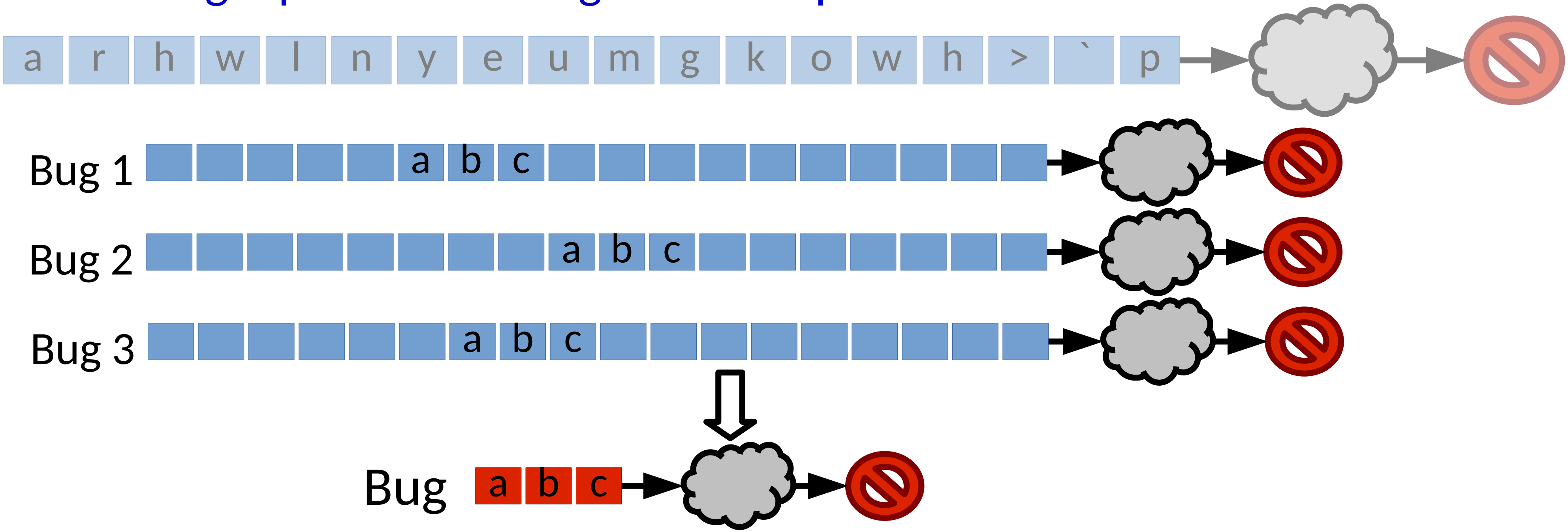
- Failing inputs can be large and complex



1) Are these reports the same bug?
2) Can we make it easier to reproduce?

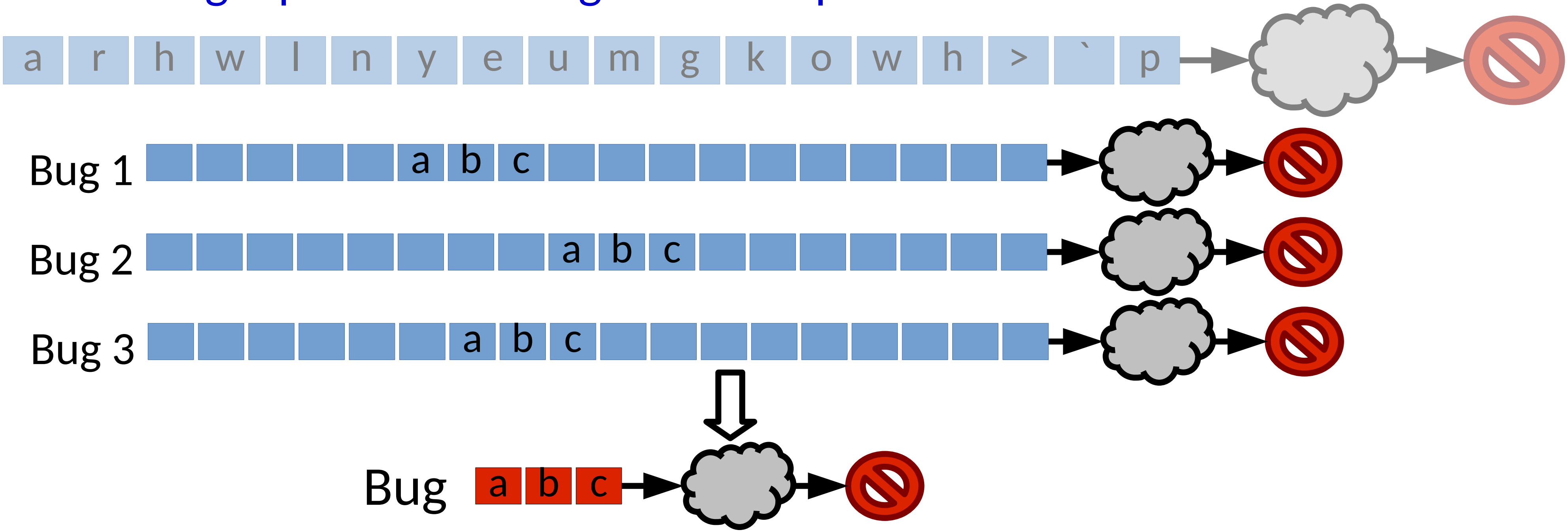
Bug reports are problematic

- Failing inputs can be large and complex



Bug reports are problematic

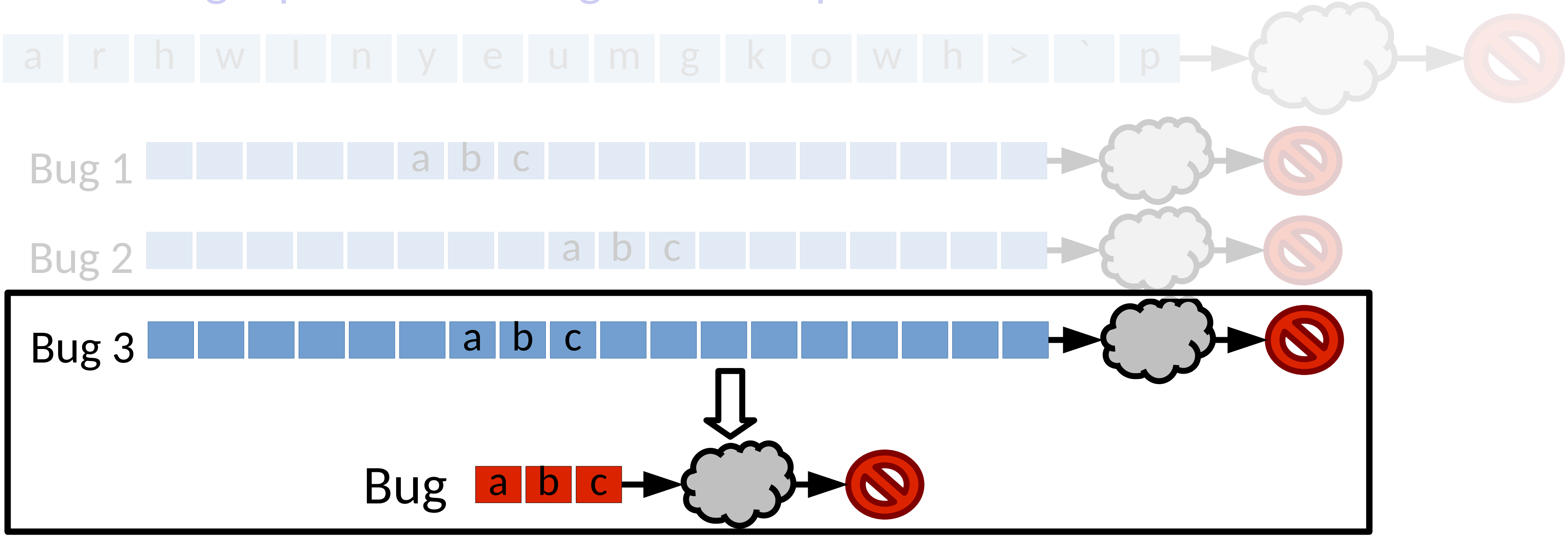
- Failing inputs can be large and complex



1) Same? Yes!
2) Easier? Yes! And easier to understand!

Bug reports are problematic

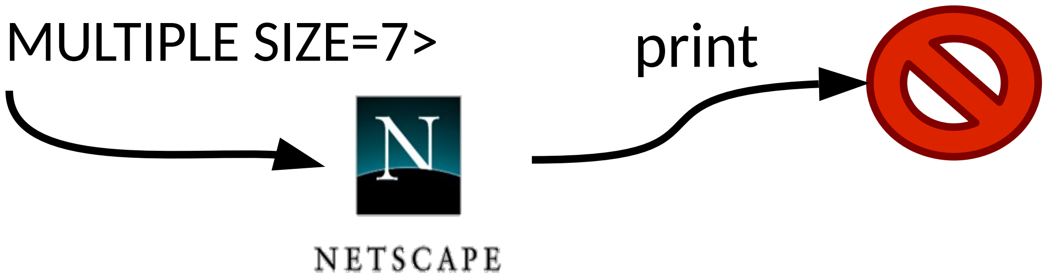
- Failing inputs can be large and complex



Test Case Reduction: finding smaller test cases that reproduce a failure

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>



http://en.wikipedia.org/wiki/File:Netscape_2_logo.gif

Classically – Delta Debugging

<SELECT ~~NAME="priority" MULTIPLE SIZE=7~~>

http://en.wikipedia.org/wiki/File:Netscape_2_logo.gif

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7> = **c**

Intuition: trial and error
1) Start w/ a failing text configuration **c**

Classically – Delta Debugging

<SELECT ~~NAME="priority"~~ MULTIPLE SIZE=7>

Intuition: trial and error

1) Start w/ a failing text configuration **c**

2) Try removing subsets (**Δ**) of input elements (**{δ}**)

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error

1) Start w/ a failing text configuration **c**

2) Try removing subsets (**Δ**) of input elements (**{δ}**)

3) Failure still exists → new input is “better”

Classically – Delta Debugging

<SELECT NAME="priority" ~~MULTIPLE SIZE=7~~>

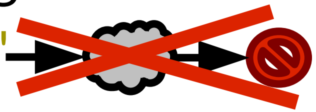
Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (**Δ**) of input elements (**{δ}**)
3) Failure still exists → new input is “better”
4) Repeat on the new input

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

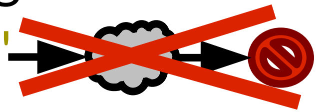
- Global Minimum: $c : \forall\ |c'| < |c|,\ c'$ 

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'| < |c|,\ c'$ 

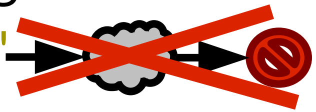
Smallest subset of the original input reproducing the failure

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'|<|c|, c'$ 

Completely impractical! Why?

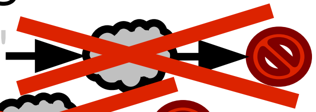
Smallest subset of the original input reproducing the failure

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'| < |c|,\ c'$ 
- Local Minimum: $c : \forall\ c' \subset c,\ c'$ 

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'|<|c|, c'$ 
- Local Minimum: $c : \forall\ c' \subset c, c'$ 

No subset of the result can
reproduce the failure.

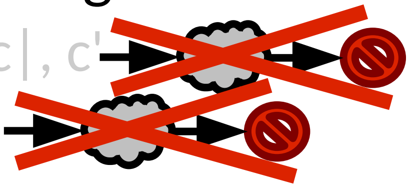
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'|<|c|, c'$
- Local Minimum: $c : \forall\ c' \subset c, c'$



How does this differ from a global minimum?
Is it still problematic?

No subset of the result can
reproduce the failure.

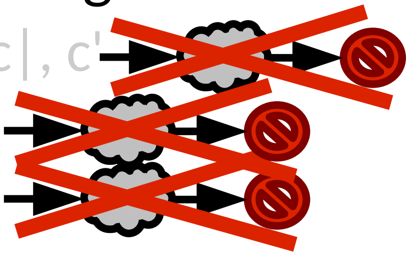
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (**Δ**) of input elements (**{δ}**)
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: c : ∀ |c'|<|c|, c'
- Local Minimum: c : ∀ c'⊂c, c'
- 1-Minimal: c: ∀ δ ∈ c, (c-{δ})



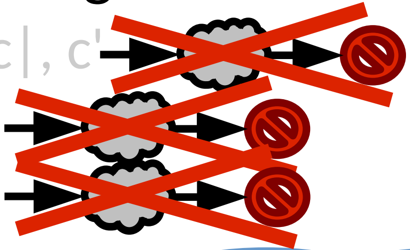
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (**Δ**) of input elements (**{δ}**)
3) Failure still exists → new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- Global Minimum: $c : \forall\ |c'| < |c|, c'$
- Local Minimum: $c : \forall\ c' \subset c, c'$
- 1-Minimal: $c: \forall\ \delta \in c, (c - \{\delta\})$



No one element can be removed
and still reproduce the failure

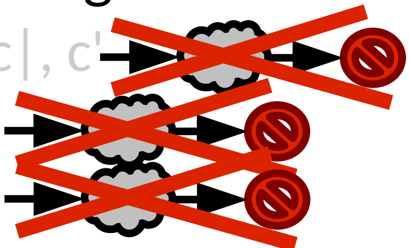
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration **c**
2) Try removing subsets (Δ) of input elements ({δ})
3) Failure still exists → new input is "better"
4) Repeat on the new input

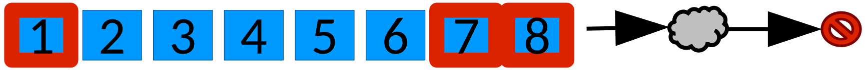
When do we stop? / What is our goal?

- Global Minimum: $c : \forall \ |c'| < |c|, c'$
- Local Minimum: $c : \forall \ c' \subset c, c'$
- 1-Minimal: $c: \forall \ \delta \in c, (c - \{\delta\})$



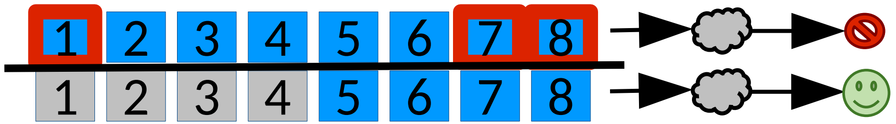
This is the classic goal.
In practice, the formalism may not pay for itself in terms of *quality* or *efficiency*! (Be pragmatic!)

Classically – Delta Debugging

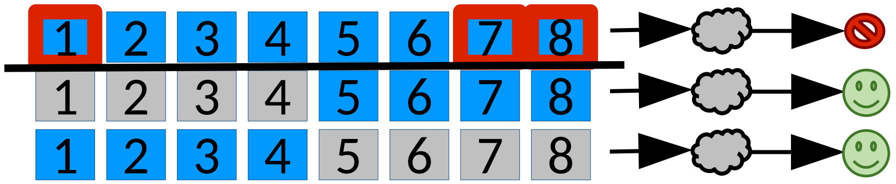


Does binary search work?

Classically – Delta Debugging

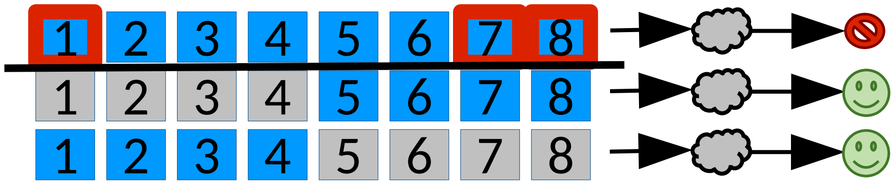


Classically – Delta Debugging



So what should we do?

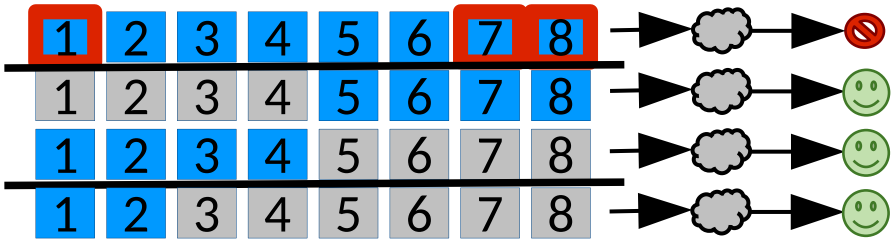
Classically – Delta Debugging



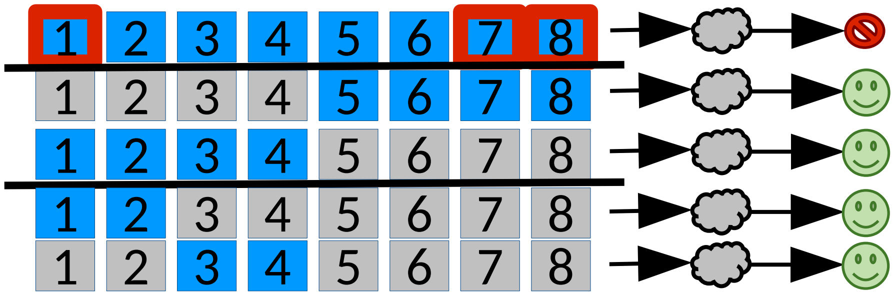
So what should we do?

We refine the granularity

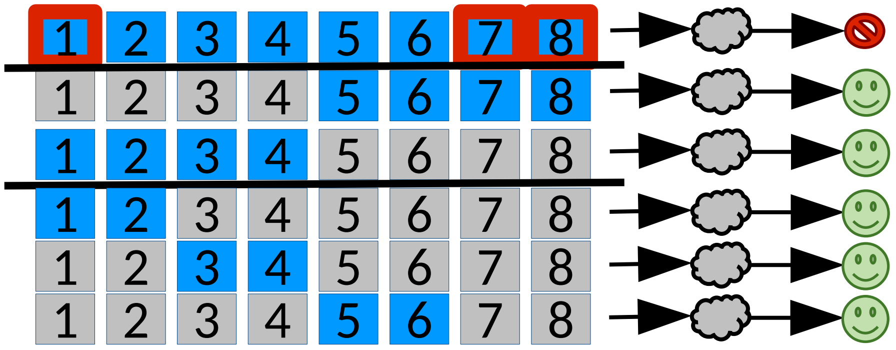
Classically – Delta Debugging



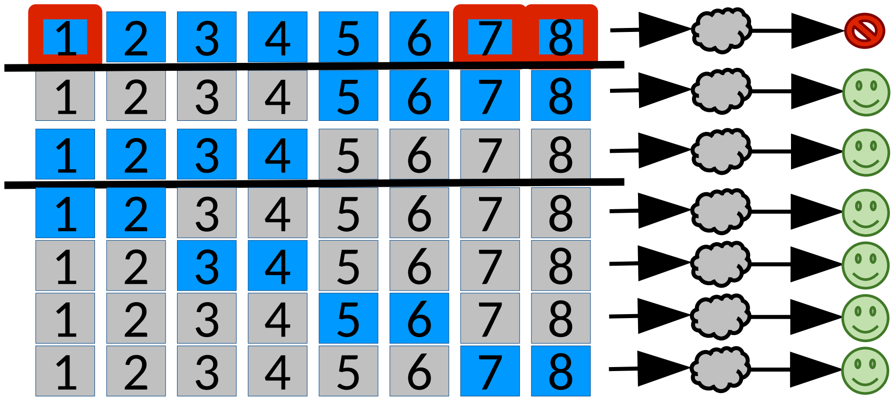
Classically – Delta Debugging



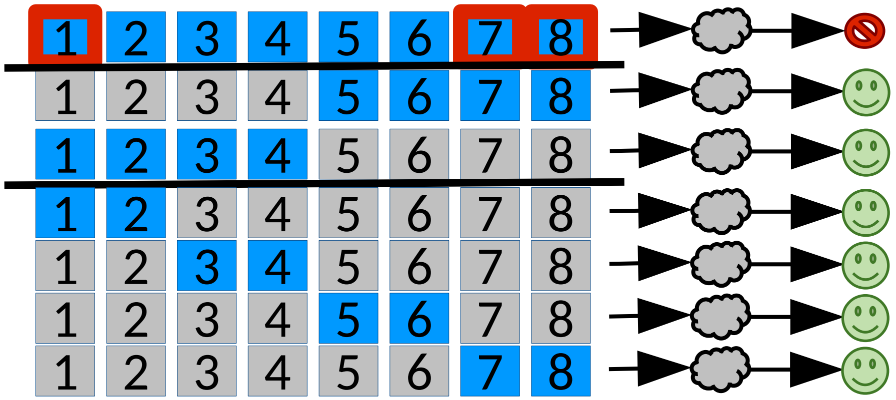
Classically – Delta Debugging



Classically – Delta Debugging

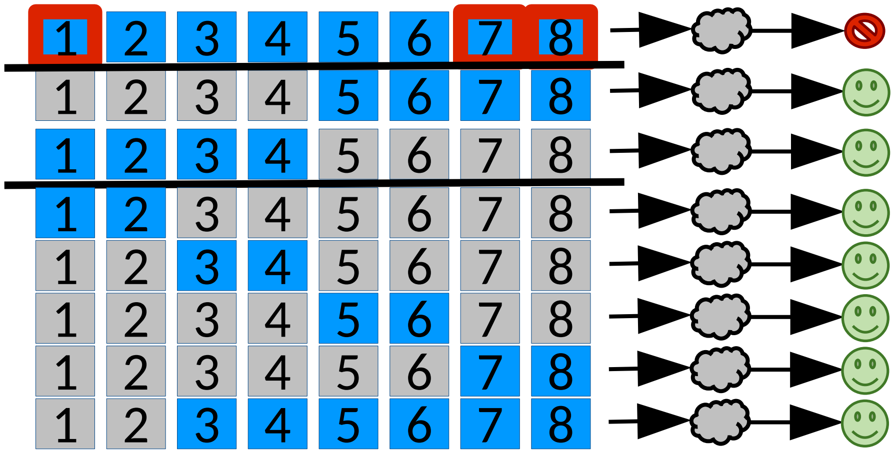


Classically – Delta Debugging

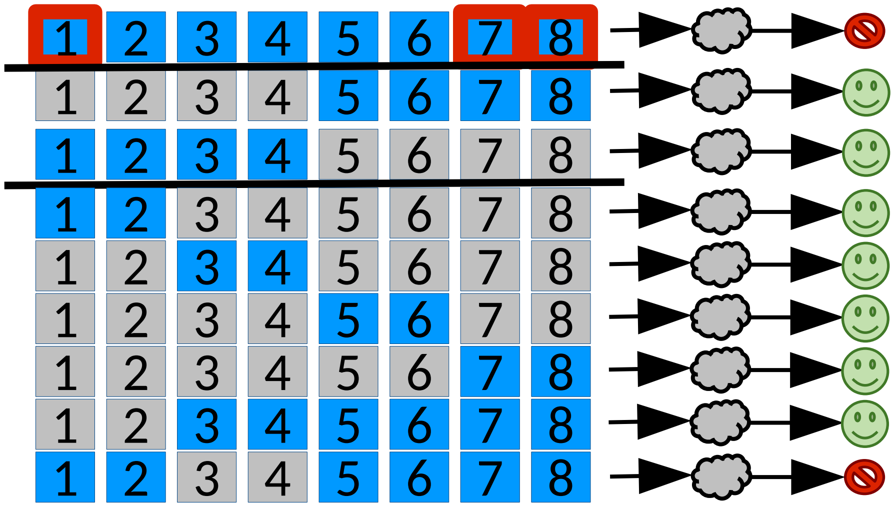


And now check complements

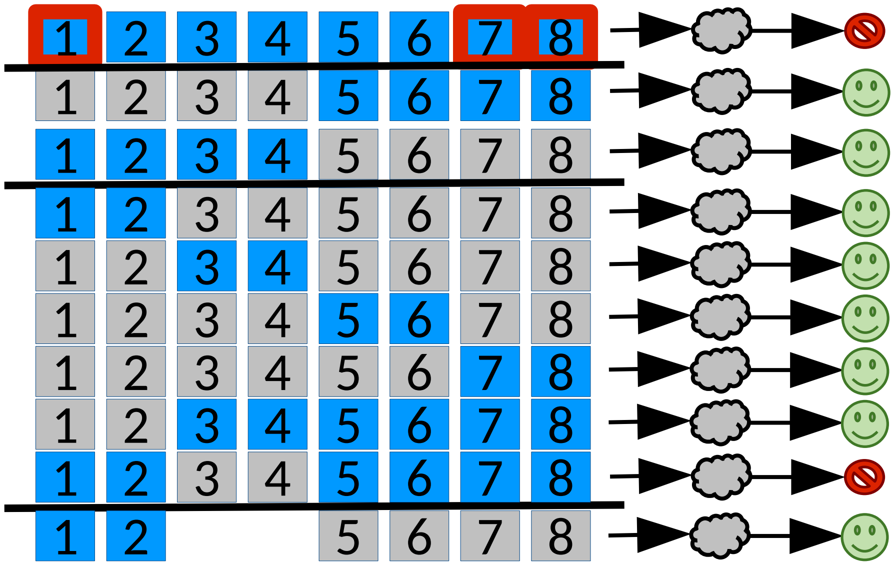
Classically – Delta Debugging



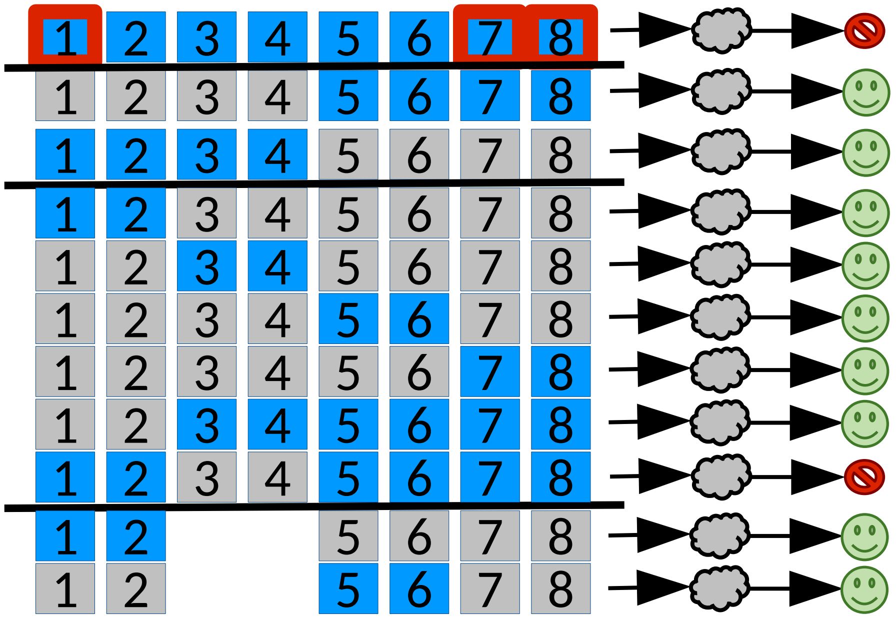
Classically – Delta Debugging



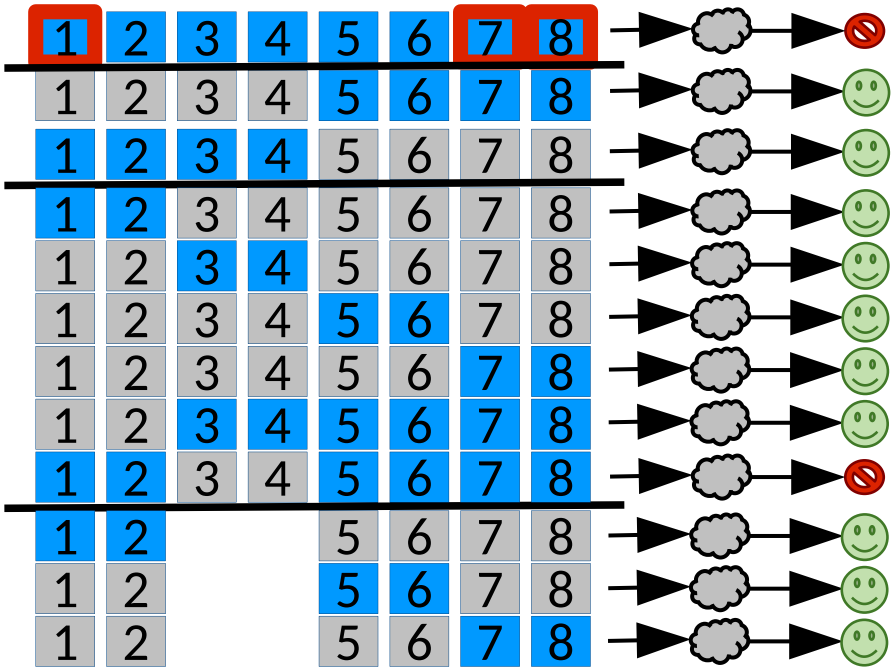
Classically – Delta Debugging



Classically – Delta Debugging

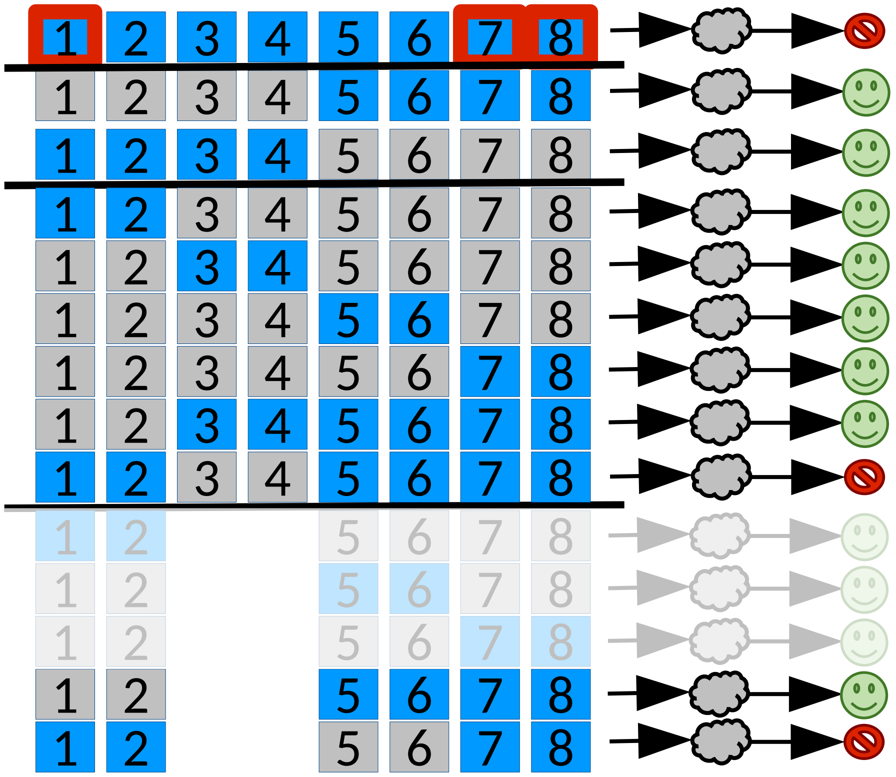


Classically – Delta Debugging

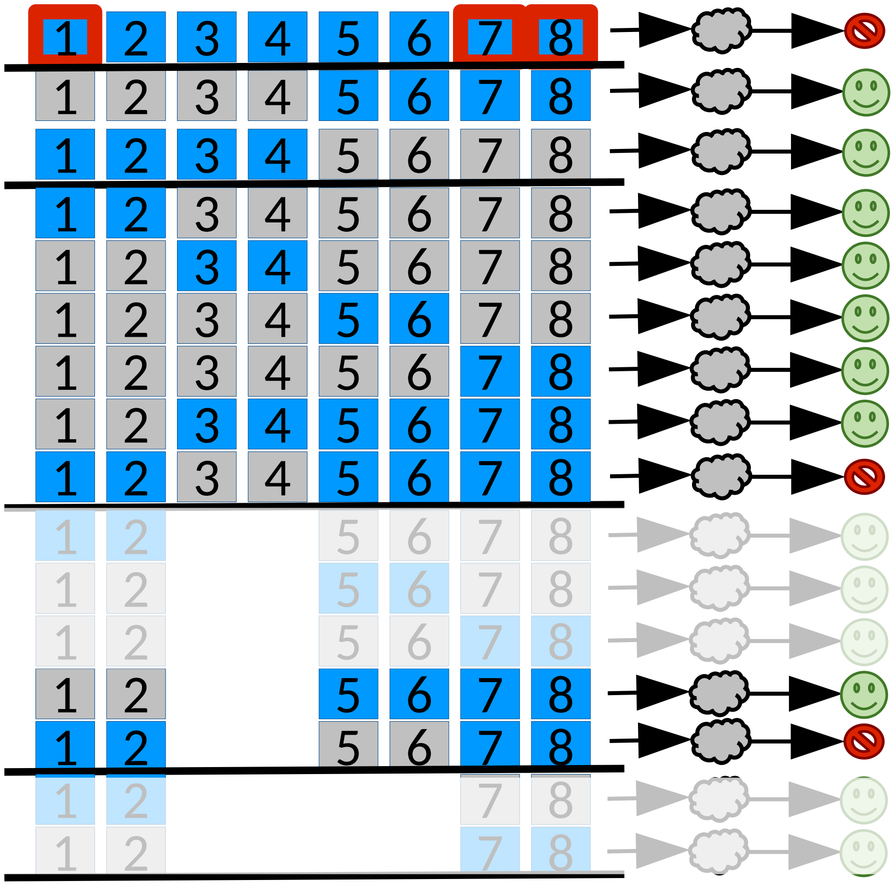


What's clever about how we recurse?

Classically – Delta Debugging

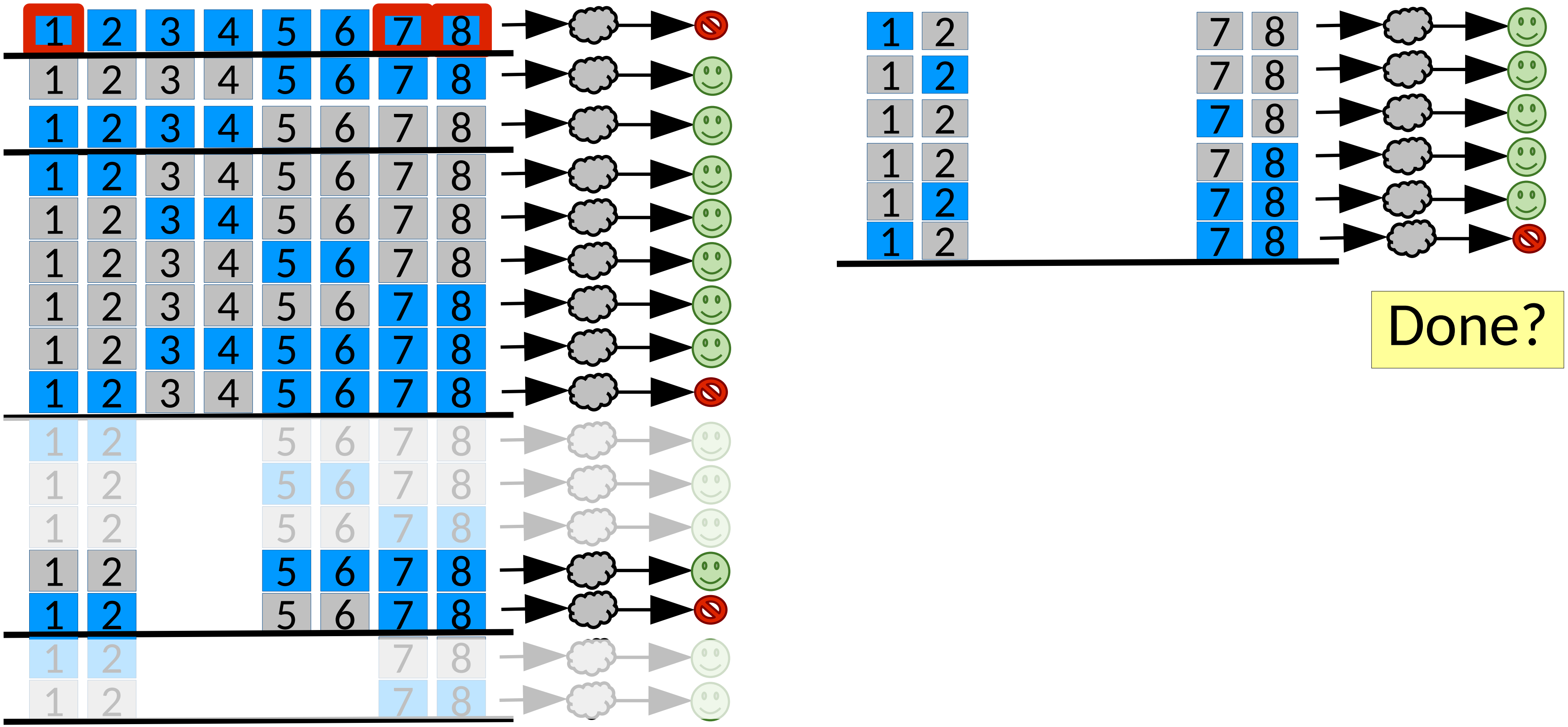


Classically – Delta Debugging

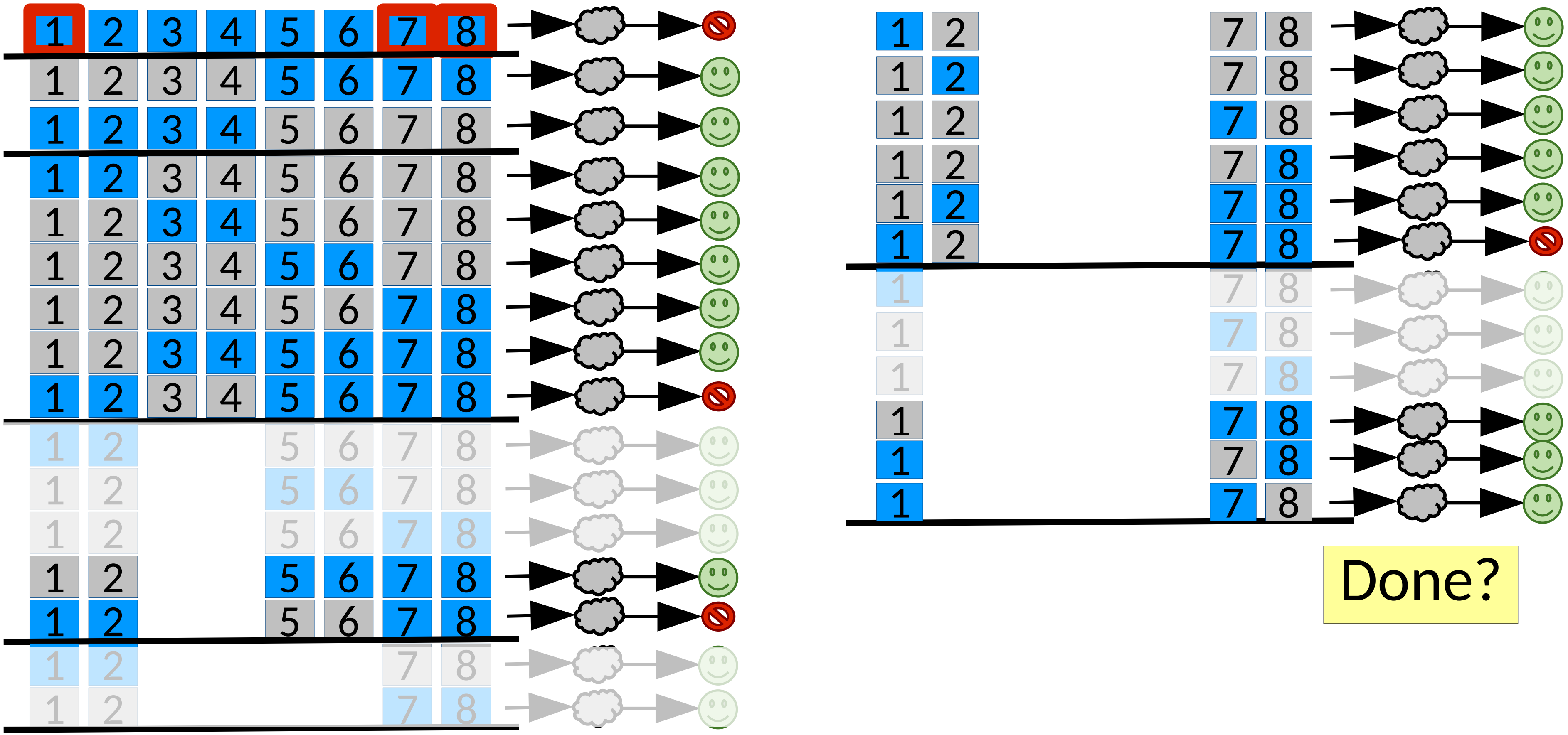


So close! How many more?

Classically – Delta Debugging



Classically – Delta Debugging



Classically – Delta Debugging

$ddmin(c) = ddmin_2(c, 2)$

Defined over
c - the input / configuration
n - the # of partitions

Classically – Delta Debugging

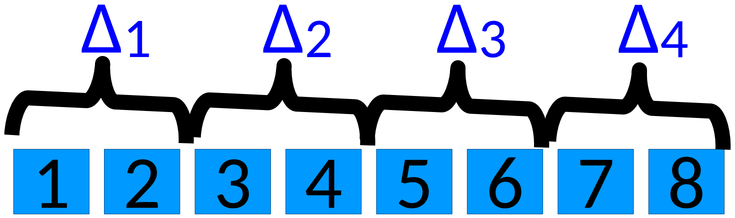
ddmin(c) = $ddmin_2$(c, 2)

c = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Classically – Delta Debugging

ddmin(**c**) = ddmin$_2$(**c**, **2**)

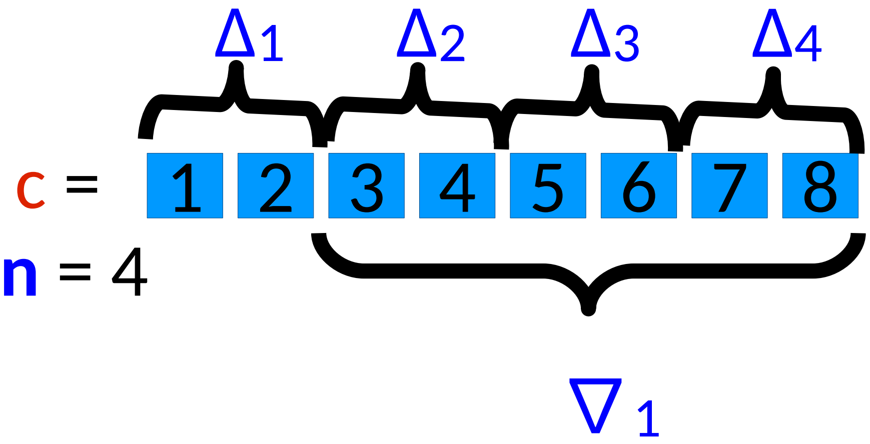
c =



n = 4

Classically – Delta Debugging

$ddmin(c) = ddmin_2(c, 2)$



Classically – Delta Debugging

$\text{ddmin}(c) = \text{ddmin}_2(c, 2)$

$$\text{ddmin}_2(c, n) = \begin{cases} \\ \\ \end{cases}$$

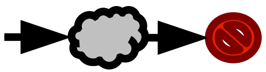
Classically – Delta Debugging

$ddmin(\textbf{c}) = ddmin_2(\textbf{c}, \textbf{2})$

$$ddmin_2(\textbf{c},\textbf{n}) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \end{cases}$$

Try each partition

$\Delta_i = \{3,4\}$ (a) 1 2 3 4 5 6 7 8



Classically – Delta Debugging

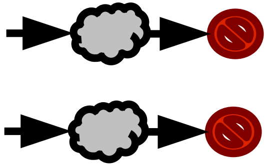
$ddmin(c) = ddmin_2(c, 2)$

$$ddmin_2(c,n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\ ddmin_2(\nabla_i, \max(n-1,2)) & \text{If ... (b)} \end{cases}$$

Try each complement

$\Delta_i = \{3,4\}$

(a)	1	2	3	4	5	6	7	8
(b)	1	2	3	4	5	6	7	8



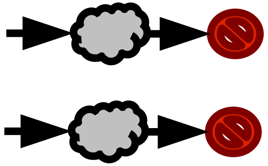
Classically – Delta Debugging

$ddmin(c) = ddmin_2(c, 2)$

$$ddmin_2(c,n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\ ddmin_2(\nabla_i, \max(n-1,2)) & \text{If ... (b)} \end{cases}$$

$\Delta_i = \{3,4\}$ (a) 1 2 3 4 5 6 7 8

(b) 1 2 3 4 5 6 7 8



Classically – Delta Debugging

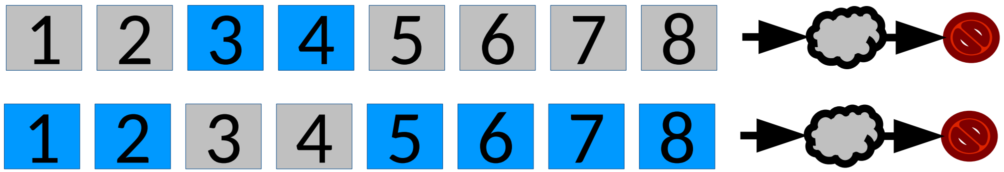
$ddmin(c) = ddmin_2(c, 2)$

$$ddmin_2(c,n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\ ddmin_2(\nabla i, \max(n-1,2)) & \text{If ... (b)} \\ ddmin_2(c, \min(|c|,2n)) & \text{If ... (c)} \end{cases}$$

Refine the granularity

$\Delta_i = \{3,4\}$ (a)

(b)



(c) $n < |c|$

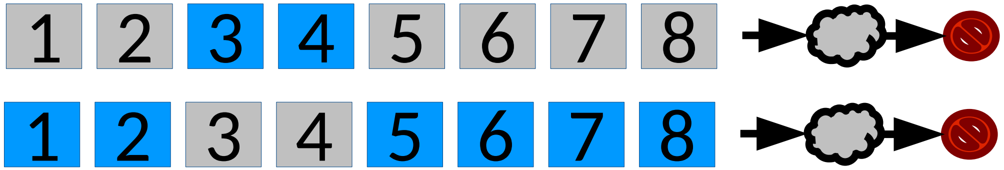
Classically – Delta Debugging

$ddmin(c) = ddmin_2(c, 2)$

$$ddmin_2(c,n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\ ddmin_2(\nabla_i, \max(n-1,2)) & \text{If ... (b)} \\ ddmin_2(c, \min(|c|,2n)) & \text{If ... (c)} \end{cases}$$

$\Delta_i = \{3,4\}$ (a)

(b)



(c) $n < |c|$

Classically – Delta Debugging

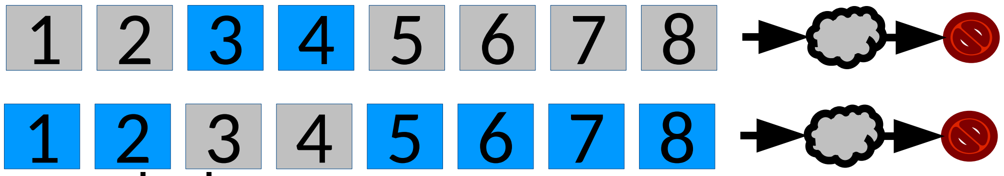
$ddmin(c) = ddmin_2(c, 2)$

$$ddmin_2(c,n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\ ddmin_2(\nabla i, \max(n-1,2)) & \text{If ... (b)} \\ ddmin_2(c, \min(|c|,2n)) & \text{If ... (c)} \\ c & \text{otherwise} \end{cases}$$

Finish

$\Delta_i = \{3,4\}$ (a)

(b)



(c) $n < |c|$

Classically – Delta Debugging

- Worst Case: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds

Classically – Delta Debugging

- Worst Case: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds
- Best Case: # tests $\leq 2\log_2(|c|)$
  - Falling back to binary search!

Classically – Delta Debugging

- Worst Case: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds
- Best Case: # tests $\leq$ $2\log_2(|c|)$
  - Falling back to binary search!
- Minimality
  - When will it only be locally minimal?
  - When will it only be 1-minimal?

Classically – Delta Debugging

- Worst Case: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds
- Best Case: # tests $\leq 2\log_2(|c|)$
  - Falling back to binary search!
- Minimality
  - When will it only be locally minimal?
  - When will it only be 1-minimal?
  - ***Does formal minimality even matter?***

Classically – Delta Debugging

- Observation:
  In practice DD may revisit elements in order to guarantee minimality

Classically – Delta Debugging

- Observation:
  In practice DD may revisit elements in order to guarantee minimality

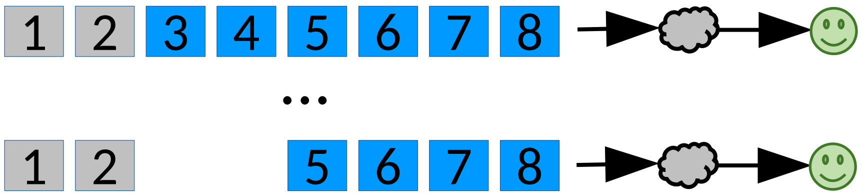
$\text{ddmin}_2(\nabla i, \max(n-1,2))$

1 2 3 4 5 6 7 8 

Classically – Delta Debugging

- Observation:
  In practice DD may revisit elements in order to guarantee minimality

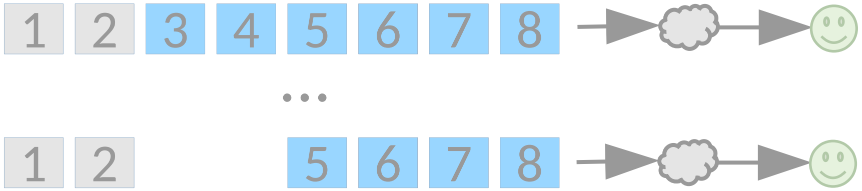
ddmin$_2$(∇**i**, **max(n-1,2)**)



Classically – Delta Debugging

- Observation:
  In practice DD may revisit elements in order to guarantee minimality

$ddmin_2(\nabla i, \max(n-1,2))$

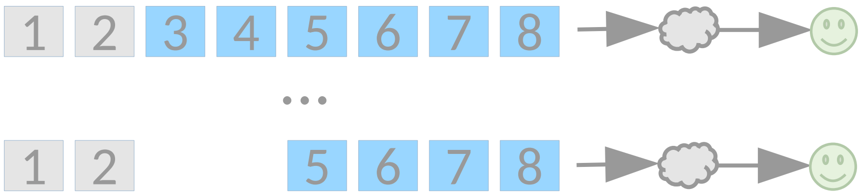


- If guaranteeing 1-minimality does not matter
  the algorithm can drop to linear time!
  - In practice this can be effective for what developers may care about

Classically – Delta Debugging

- Observation:
  In practice DD may revisit elements in order to guarantee minimality

$ddmin_2(\nabla i, \max(n-1,2))$



- If guaranteeing 1-minimality does not matter
  the algorithm can drop to linear time!
  - In practice this can be effective for what developers may care about

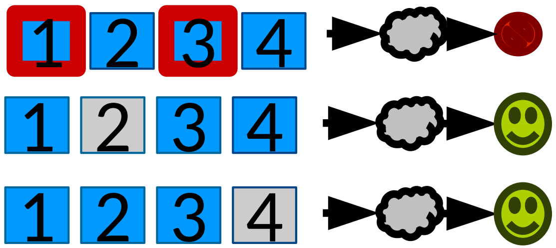
Don’t get bogged down by formalism
when it doesn’t serve you!

Test Case Reduction in Practice

- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly

Test Case Reduction in Practice

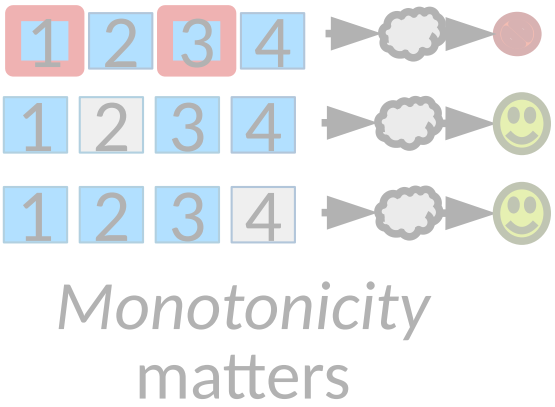
- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly
- What are the possible causes of problems?

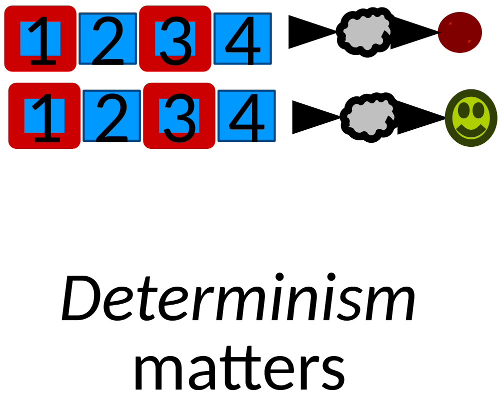


Monotonicity matters

Test Case Reduction in Practice

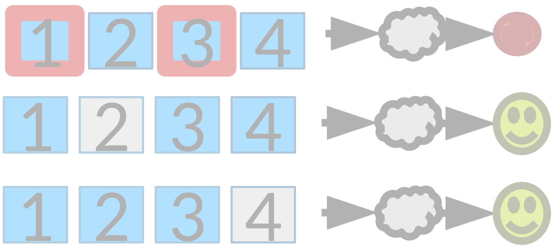
- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly
- What are the possible causes of problems?



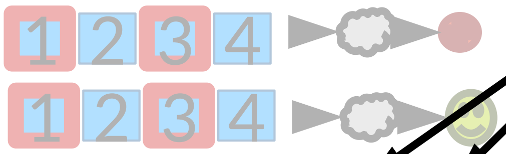


Test Case Reduction in Practice

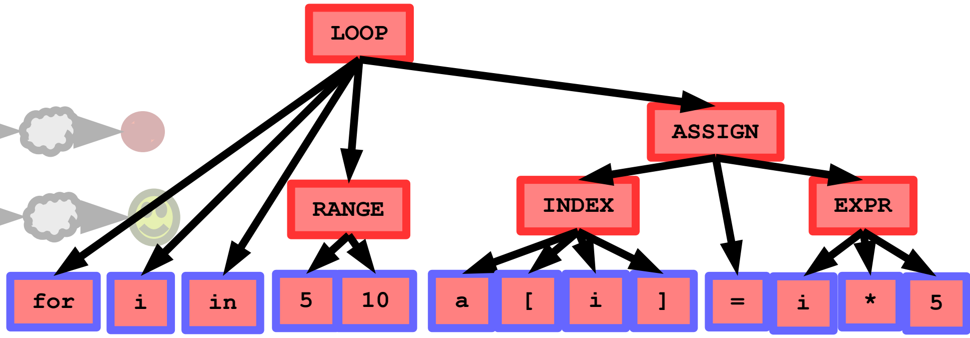
- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly
- What are the possible causes of problems?



Monotonicity matters



Determinism matters



Structure matters

Test Case Reduction for Compilers

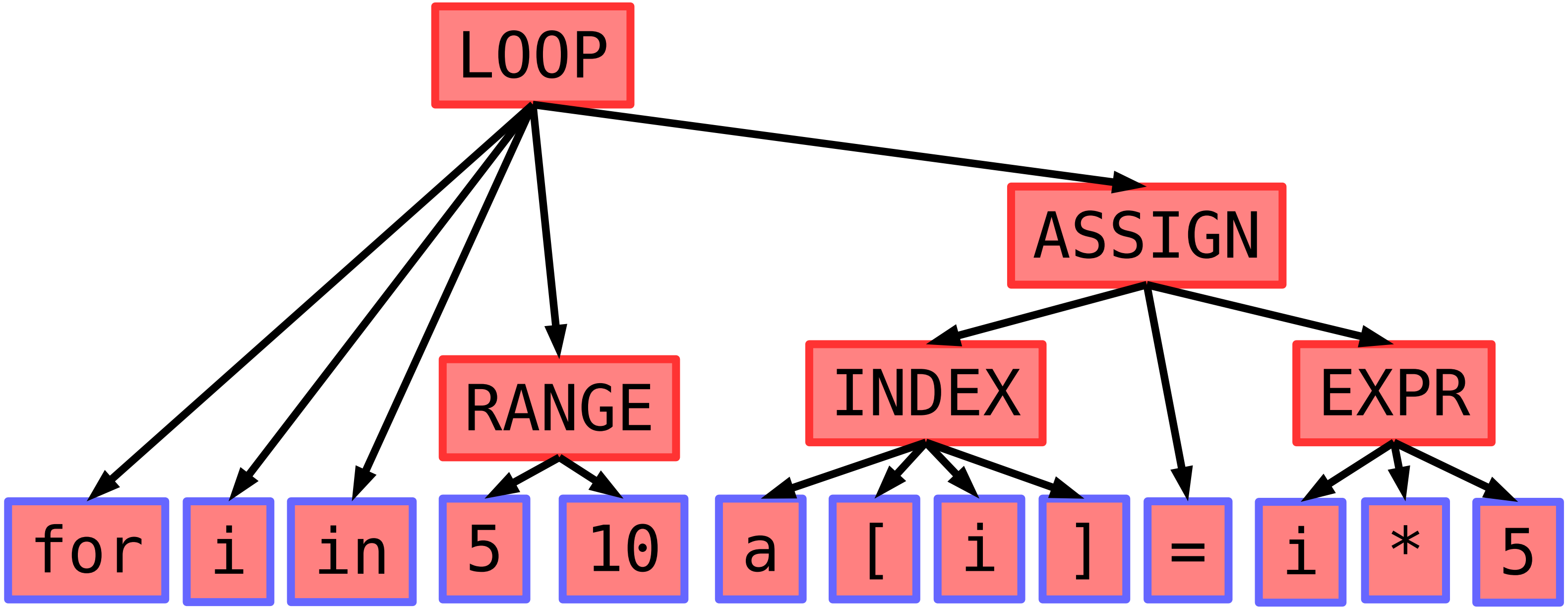
- Programs are highly structured, so TCR for compilers faces challenges

Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
- What structures could we use to guide the process?

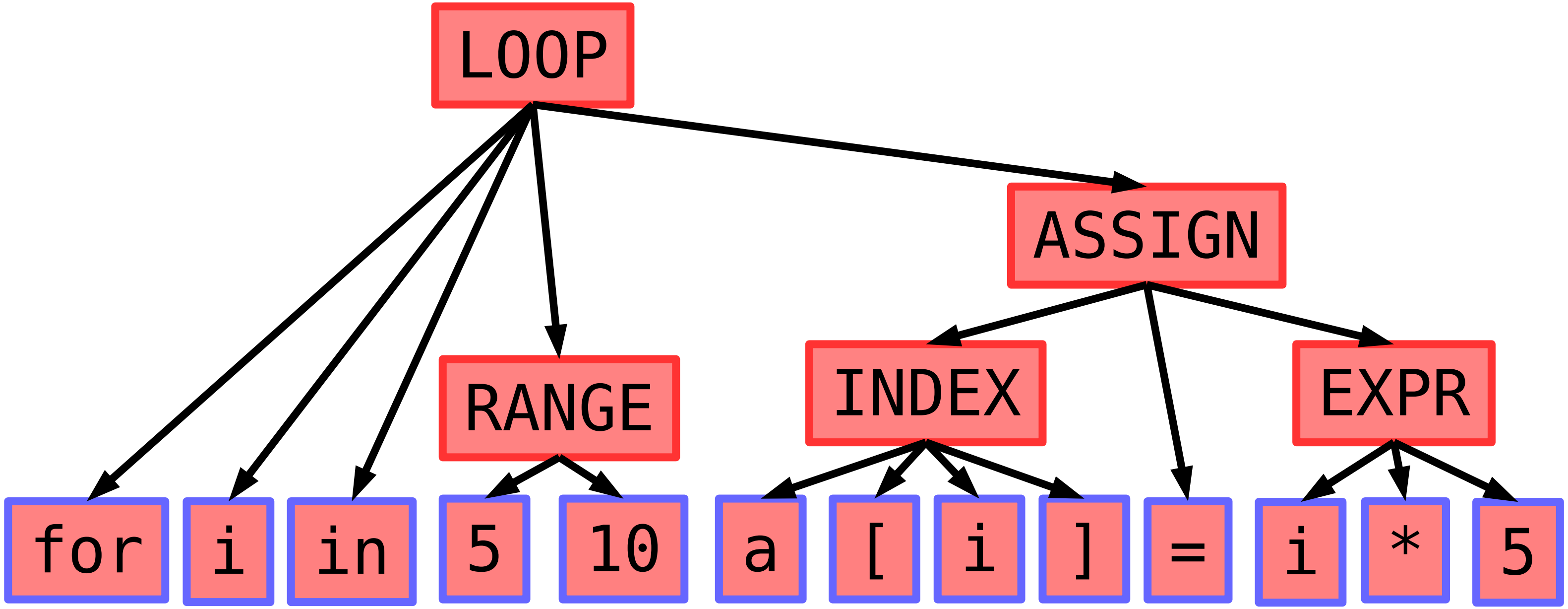
Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
- What structures could we use to guide the process?



Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
- What structures could we use to guide the process?
- What challenges still remain?



Generalizing Further

- What else could we think of as test case reduction?

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “   ” in a distributed system?
  - “   ” microservice application?

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - "   " in a distributed system?
  - "   " microservice application?
  - Automatically generated test cases?

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “ ” in a distributed system?
  - “ ” microservice application?
  - Automatically generated test cases?
  - …

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “ ” in a distributed system?
  - “ ” microservice application?
  - Automatically generated test cases?

- The ability to treat the program *as an oracle* is also very powerful

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - "   " in a distributed system?
  - "   " microservice application?
  - Automatically generated test cases?

- The ability to treat the program *as an oracle* is also very powerful
  - We can get new data by running the program
  - This can be combined with reinforcement learning to accomplish hard tasks

Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “ ” in a distributed system?
  - “ ” microservice application?
  - Automatically generated test cases?

- The ability to treat the program *as an oracle* is also very powerful
  - We can get new data by running the program
  - This can be combined with reinforcement learning to accomplish hard tasks
  - We saw this before when discussing test suites!

Example: Memory Safety Bugs

Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities

Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities
- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging

Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities
- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging
- Two main types of issues:
  - Spatial – Out of bounds stack/heap/global accesses
  - Temporal – Use after free

Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities
- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging
- Two main types of issues:
  - Spatial – Out of bounds stack/heap/global accesses
  - Temporal – Use after free
- We would like to automatically identify & provide assistance with high precision and low overhead
  - Suitable for testing & sometimes maybe deployment!

Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead

Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead
- Common implementations
  - Valgrind – DBI based
  - AddressSanitizer – static instrumentation based

Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead
- Common implementations
  - Valgrind – DBI based
  - AddressSanitizer – static instrumentation based

Note, the implementation style affects
which bugs can be recognized!
Why?

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`

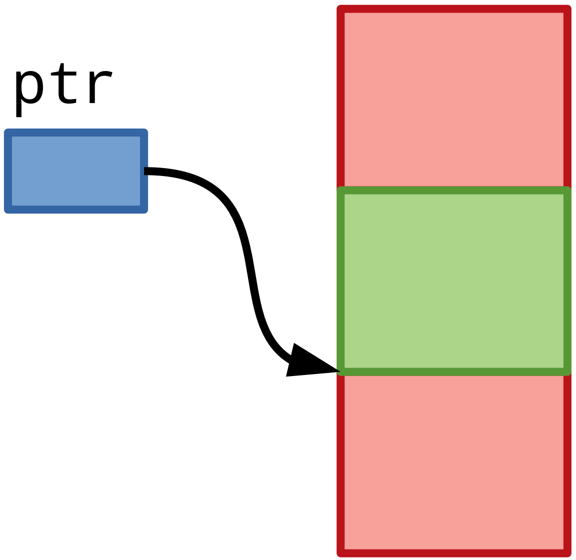
AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)

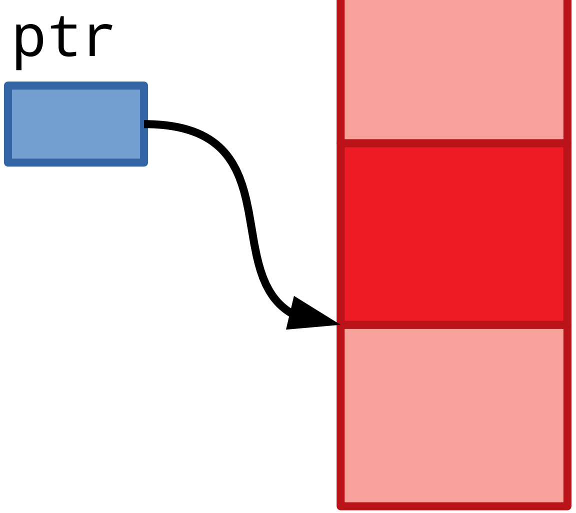
    1) around `malloc`ed chunks




```
ptr = malloc(sizeof(MyStruct));
```

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d

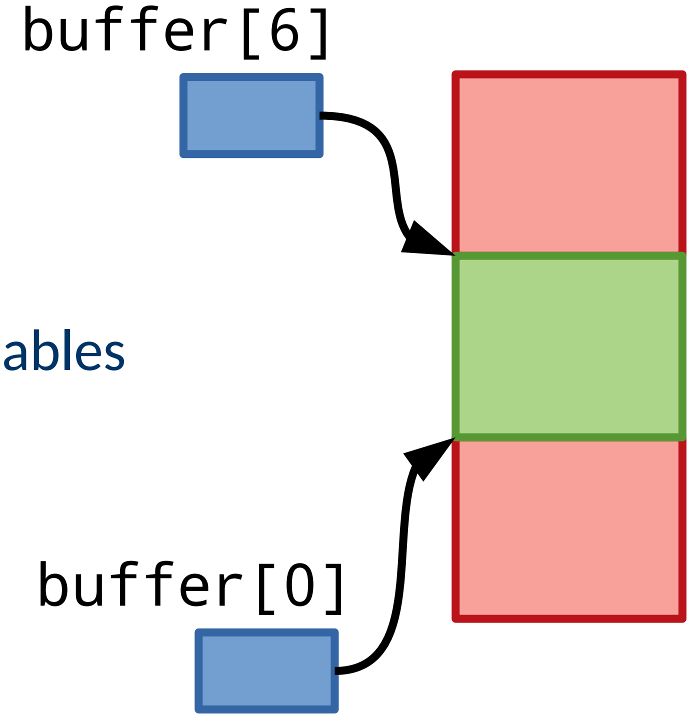



```
free(ptr);
```

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloced` chunks
    2) when it is `freed`
    3) around buffers and local variables

```
void foo() {
  int buffer[5];
  ...
}
```

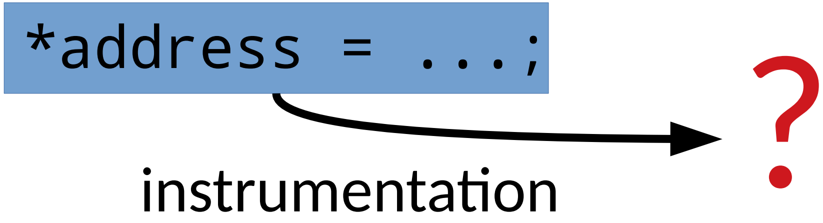


AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error



AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error

```
*address = ...;
```

instrumentation →

```
If (IsPoisoned(address, size)) {
  ReportError(address, size, isWrite);
}
*address = ...
```

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error
- The tricky part is tracking & efficiently checking redzones.

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error
- The tricky part is tracking & efficiently checking redzones.
  - Instrumenting *every* memory access is costly!

AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error
- The tricky part is tracking & efficiently checking redzones.
  - Instrumenting every memory access is costly!
  - We must track all memory … inside that same memory!

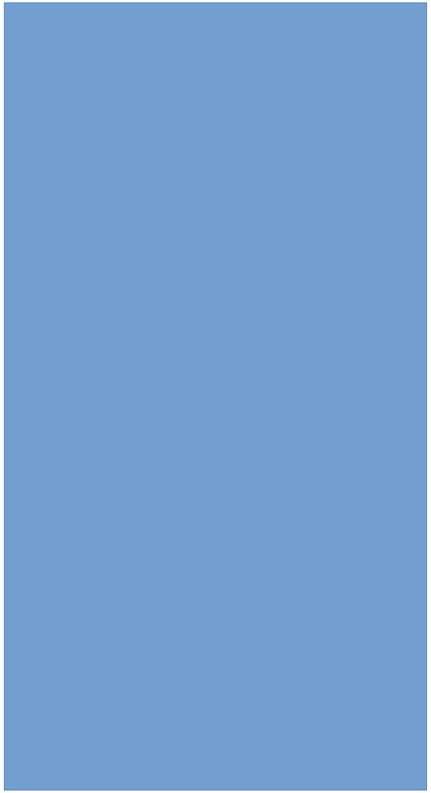
AddressSanitizer

- Need to track which memory is valid & check efficiently…
- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)

    1) around `malloc`ed chunks
    2) when it is `free`d
    3) around buffers and local variables
  - Access of poisoned memory causes an error
- The tricky part is tracking & efficiently checking redzones.
  - Instrumenting every memory access is costly!
  - We must track all memory … inside that same memory!

This kind of issue is common in dynamic analyses.

AddressSanitizer – Shadow Memory

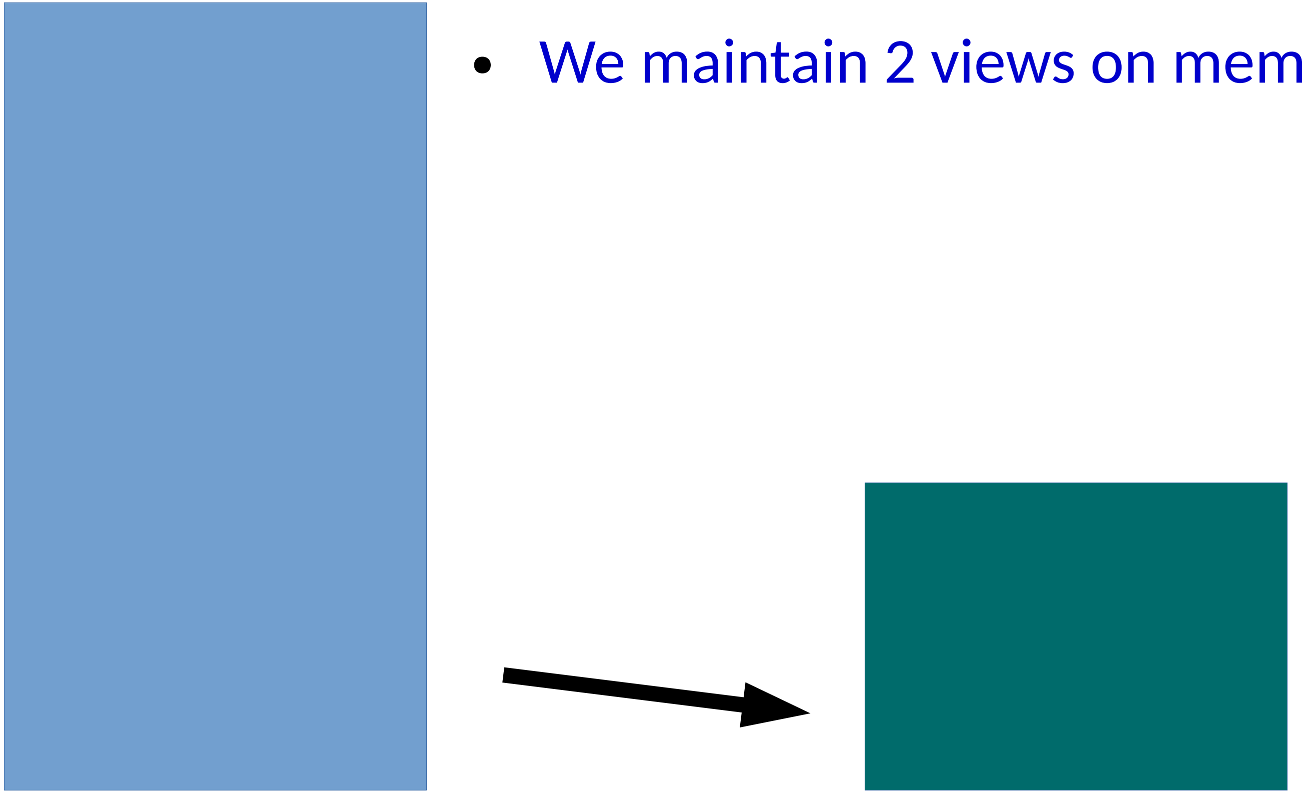


Application Memory

Need to know whether *any byte* of application memory is poisoned.

AddressSanitizer – Shadow Memory

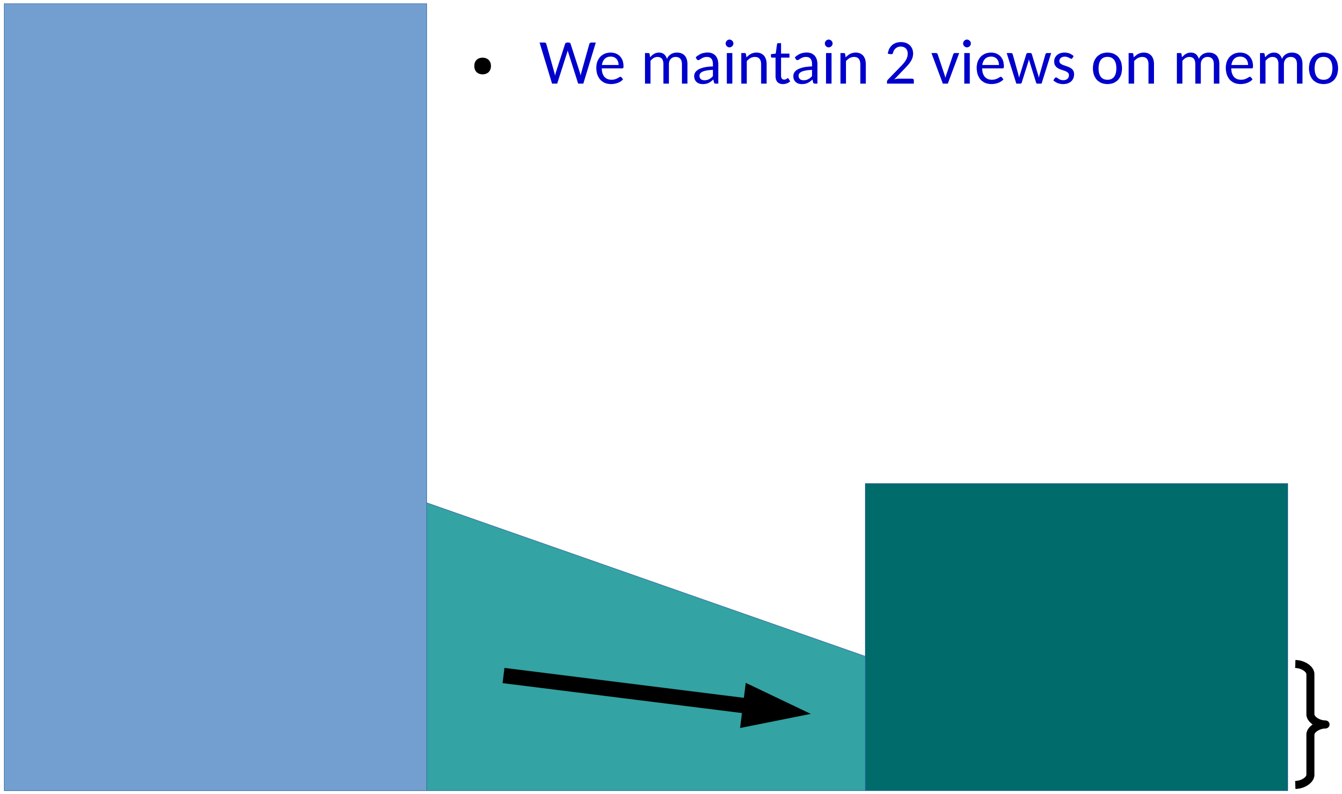
- We maintain 2 views on memory



Application Memory *Shadow* Memory

AddressSanitizer – Shadow Memory

- We maintain 2 views on memory

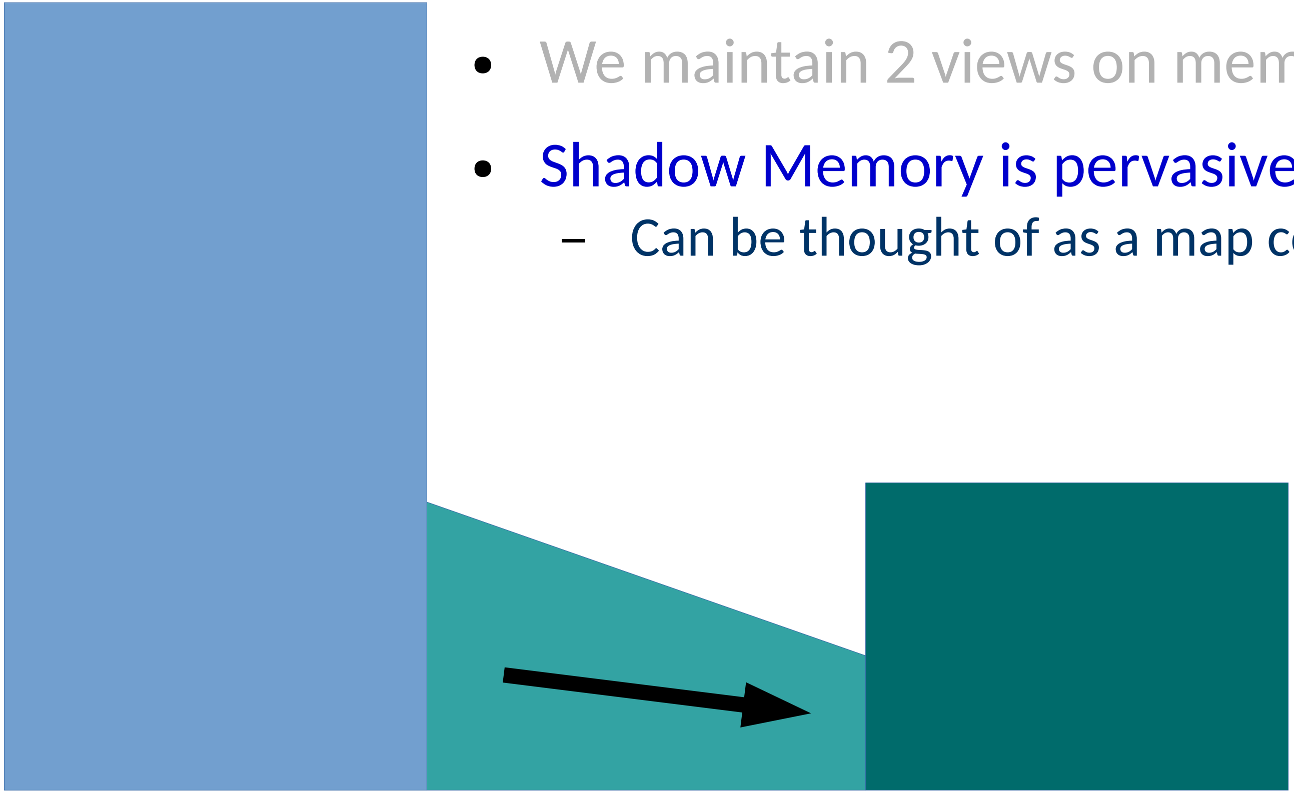


Shadow memory of the shadow memory!

Application Memory *Shadow* Memory

AddressSanitizer – Shadow Memory

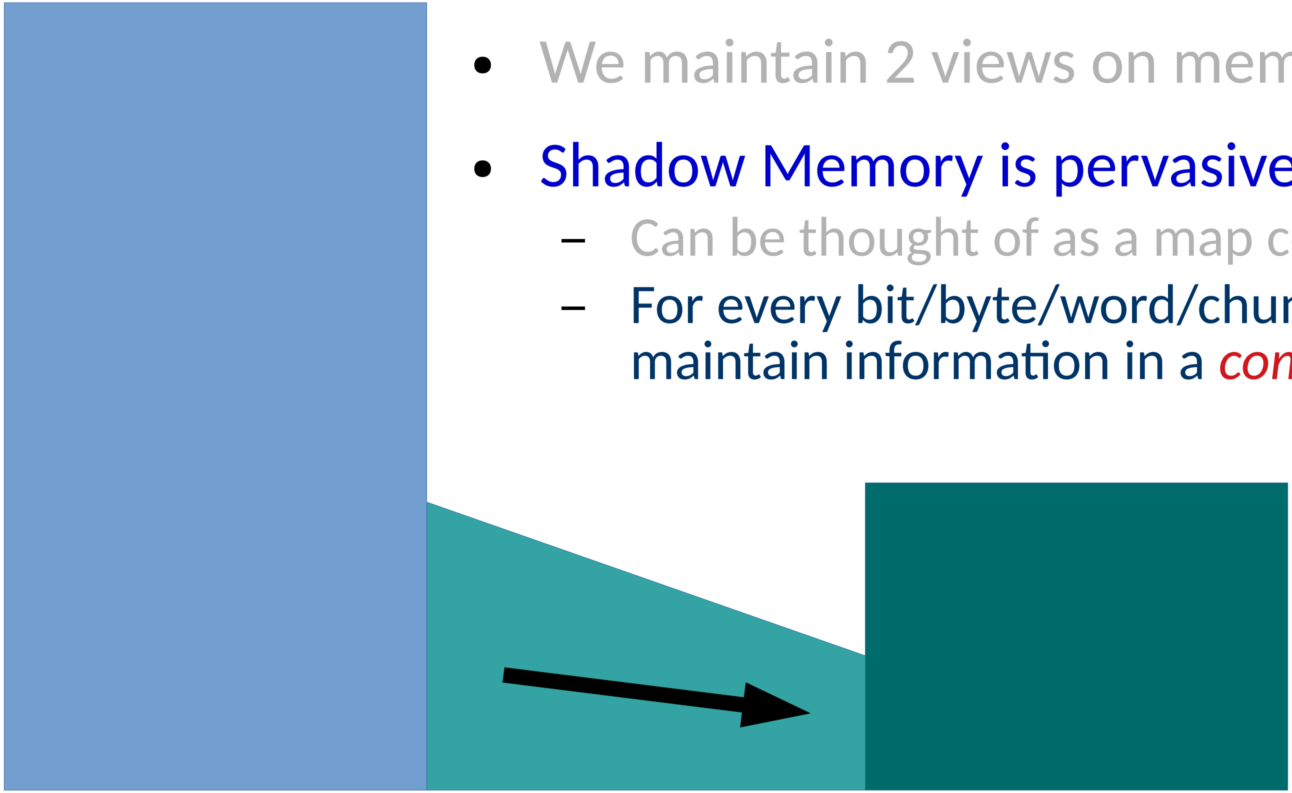
- We maintain 2 views on memory
- Shadow Memory is pervasive in dynamic analysis
  - Can be thought of as a map containing analysis data



Application Memory *Shadow* Memory

AddressSanitizer – Shadow Memory

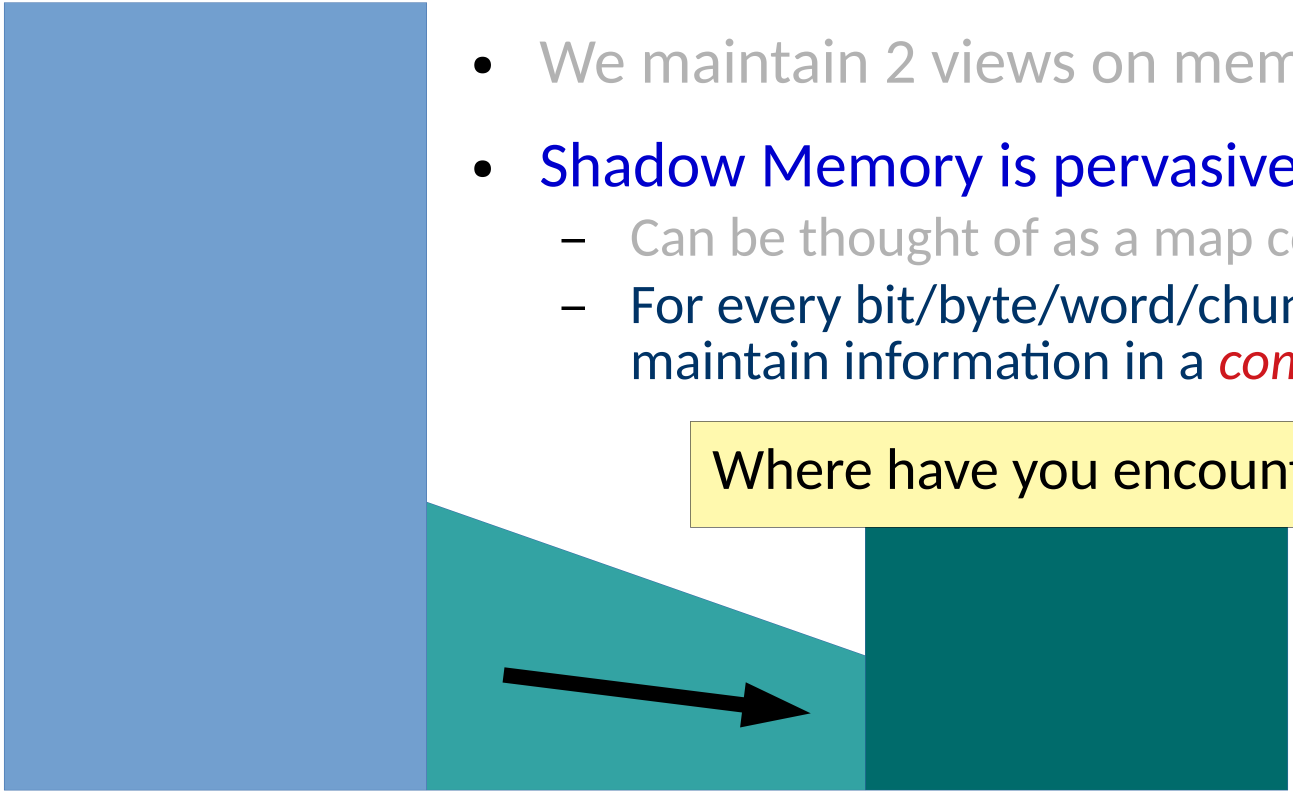
- We maintain 2 views on memory
- Shadow Memory is pervasive in dynamic analysis
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a *compact* table



AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- Shadow Memory is pervasive in dynamic analysis
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a *compact* table

Where have you encountered this before?

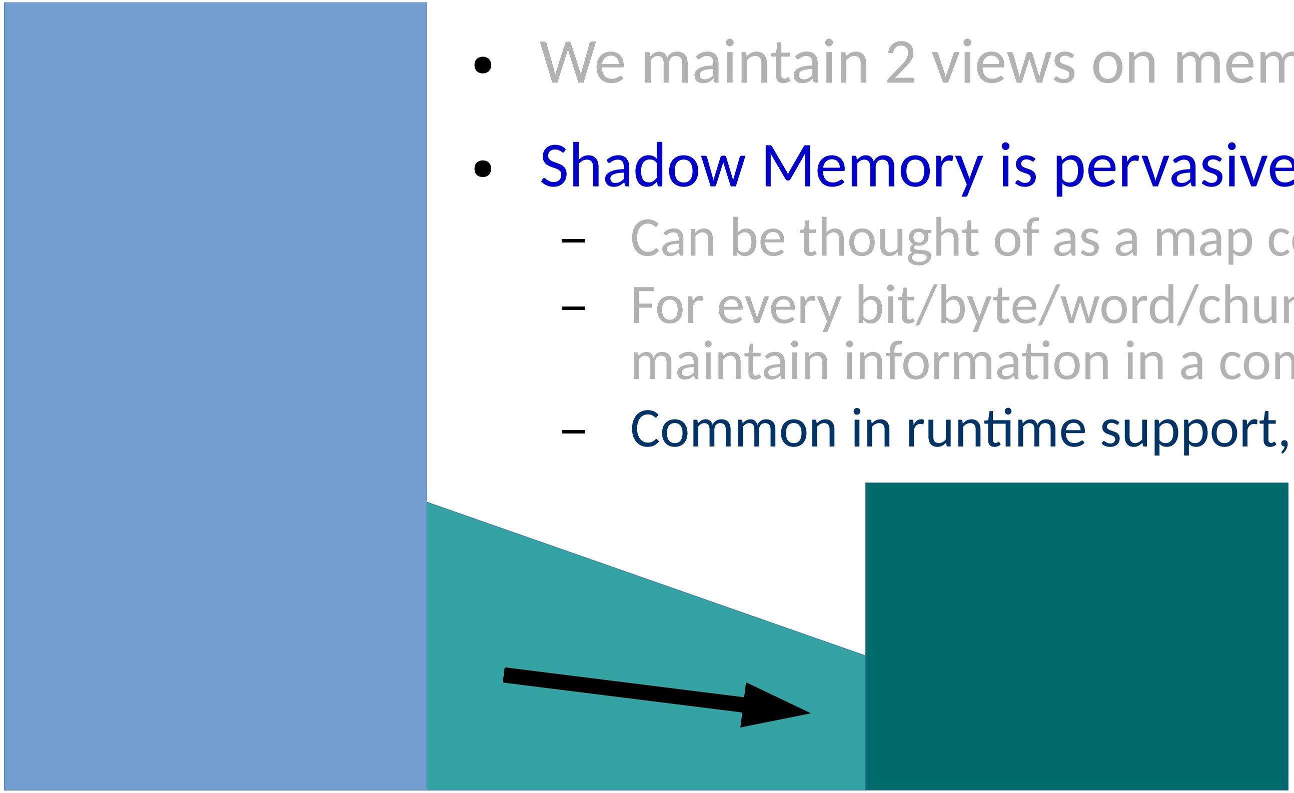


Application Memory

Shadow Memory

AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- Shadow Memory is pervasive in dynamic analysis
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support, e.g. page tables



Application Memory *Shadow* Memory

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

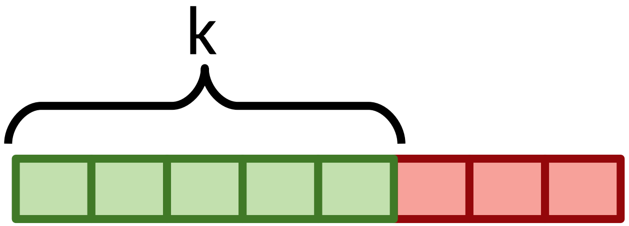
Encoding & abstraction
efficiency strategies

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)

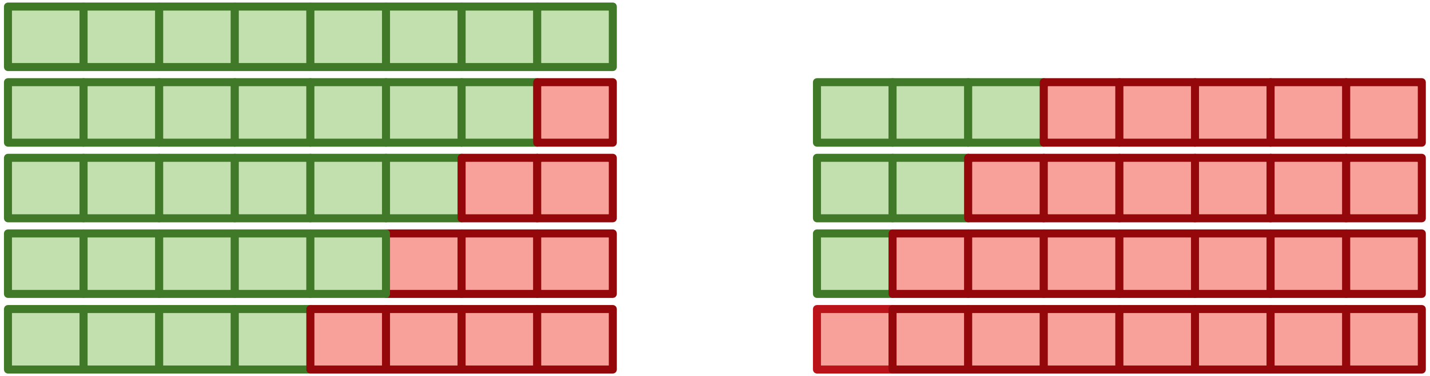
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable



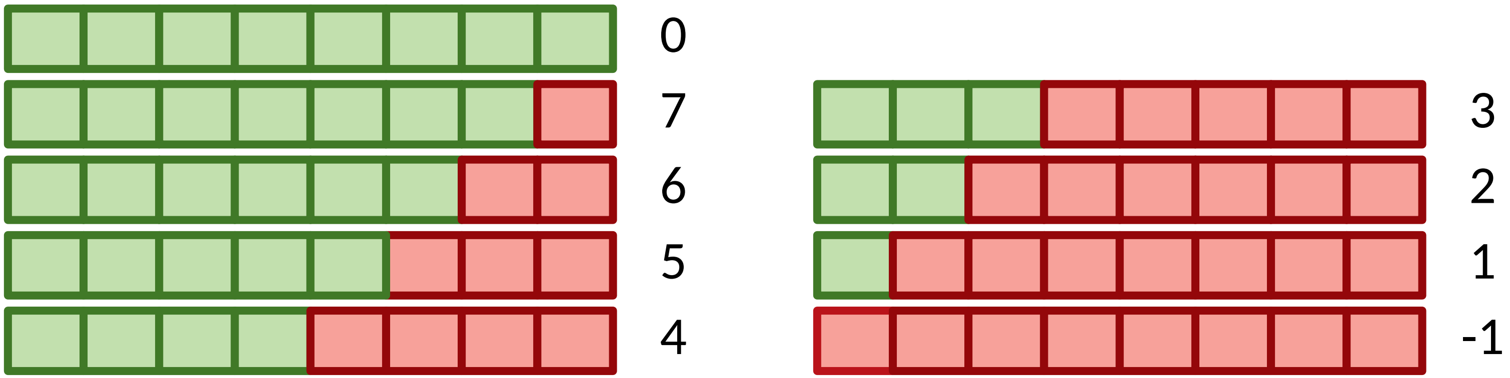
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states



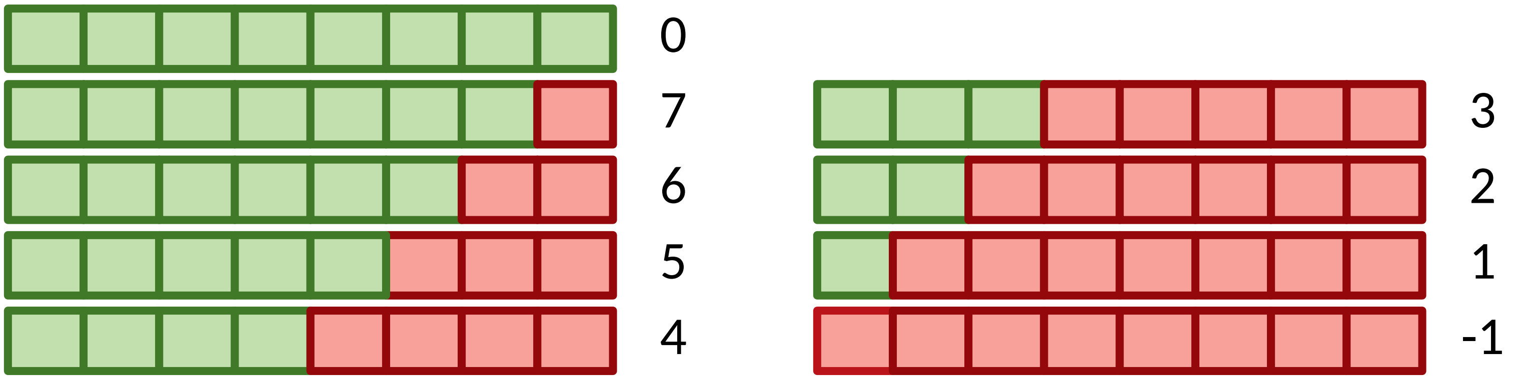
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number



AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
    All good = 0 All bad = -1 Partly good = # good



AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - `Shadow = (address >> 3) + Offset`

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - `Shadow = (address >> 3) + Offset`
  - With PIE, `Shadow = (address >> 3)`

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - `Shadow = (address >> 3) + Offset`
  - With PIE, `Shadow = (address >> 3)`

```
if (*(address>>3)) {
  ReportError(...);
}
*address = ...
```

AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - `Shadow = (address >> 3) + Offset`
  - With PIE, `Shadow = (address >> 3)`

Now you can also see the reason for the numerical encoding....

```
if (*(address>>3)) {
  ReportError(...);
}
*address = ...
```

AddressSanitizer – Shadow Memory

- Handling accesses of size < N (N=8)

```
shadow = address >> 3
state = *shadow
if (state != 0 && (state < (address & 7) + size)) {
  ReportError(...);
}
*address = ...
```

AddressSanitizer – Shadow Memory

- Handling accesses of size < N (N=8)

```
shadow = address >> 3
state = *shadow
if (state != 0 && (state < (address & 7) + size)) {
  ReportError(...);
}
*address = ...
```

Careful construction of states can make runtime checks efficient

AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality

AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- Overheads
  - Need to determine what resources are being consumed

AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- Overheads
  - Need to determine what resources are being consumed
  - Memory –
    Shadow memory *capacity* is cheap, but accessed shadows matter

AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- Overheads
  - Need to determine what resources are being consumed
  - Memory –
    Shadow memory *capacity* is cheap, but accessed shadows matter
  - Running time –
    Can effectively be free for I/O bound projects
    Up to 25x overheads on some benchmarks

AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- Overheads
  - Need to determine what resources are being consumed
  - Memory –
    Shadow memory *capacity* is cheap, but accessed shadows matter
  - Running time –
    Can effectively be free for I/O bound projects
    Up to 25x overheads on some benchmarks
- Quality
  - Precision & recall matter

Where will it miss bugs?
Where will it raise false alarms?

AddressSanitizer - Evaluating

- False negatives
  - Computed pointers that are accidentally in bounds

AddressSanitizer - Evaluating

- False negatives
  - Computed pointers that are accidentally in bounds
  - Unaligned accesses that are partially out of bounds

AddressSanitizer - Evaluating

- False negatives
  - Computed pointers that are accidentally in bounds
  - Unaligned accesses that are partially out of bounds
  - Use after frees with significant churn

Example: Comparing Executions

Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior

Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior
- Several tasks could be made simpler by trace comparison

Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior

- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new

Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior

- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new

Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior
- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new

Why compare traces or executions?

- Understanding the differences between two executions (& how some differences cause others) can help explain program behavior
- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - Debugging with concurrency – buggy vs nonbuggy schedules

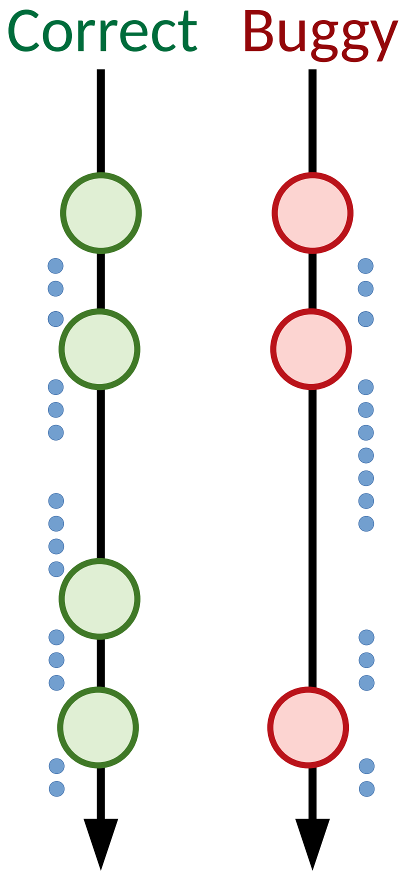
Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior
- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - Debugging with concurrency – buggy vs nonbuggy schedules
  - Malware analysis – malicious vs nonmalicious run

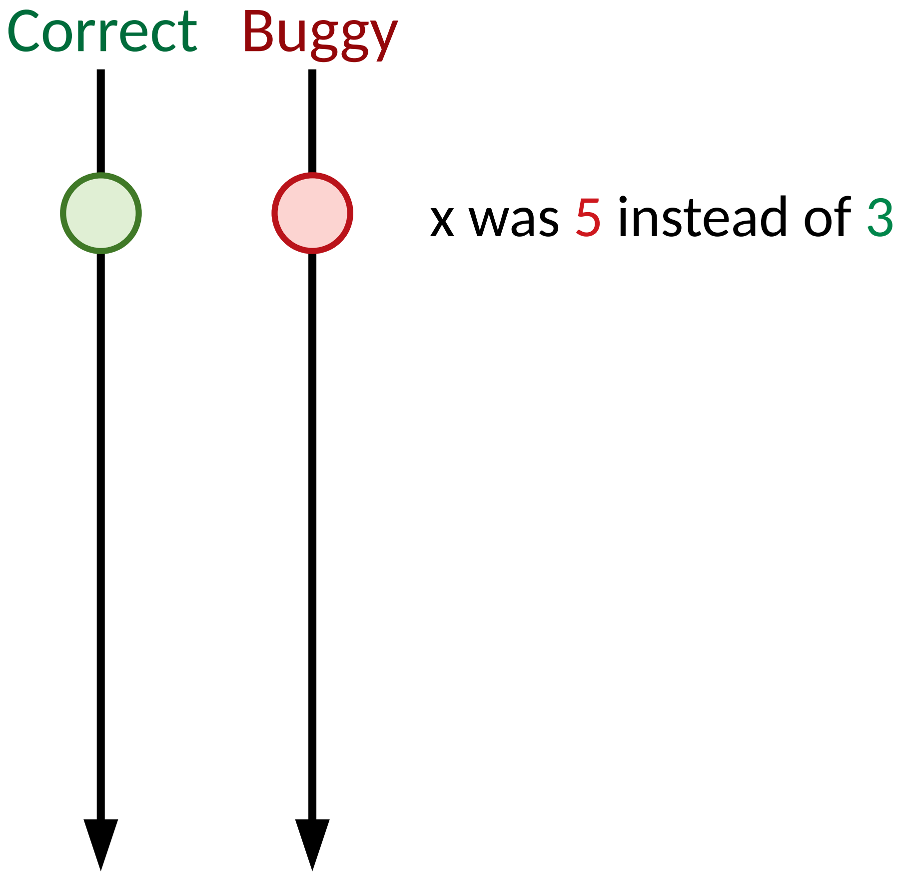
Why compare traces or executions?

- Understanding the differences between two executions
  (& how some differences cause others)
  can help explain program behavior
- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - Debugging with concurrency – buggy vs nonbuggy schedules
  - Malware analysis – malicious vs nonmalicious run
  - Reverse engineering – desired behavior vs undesirable

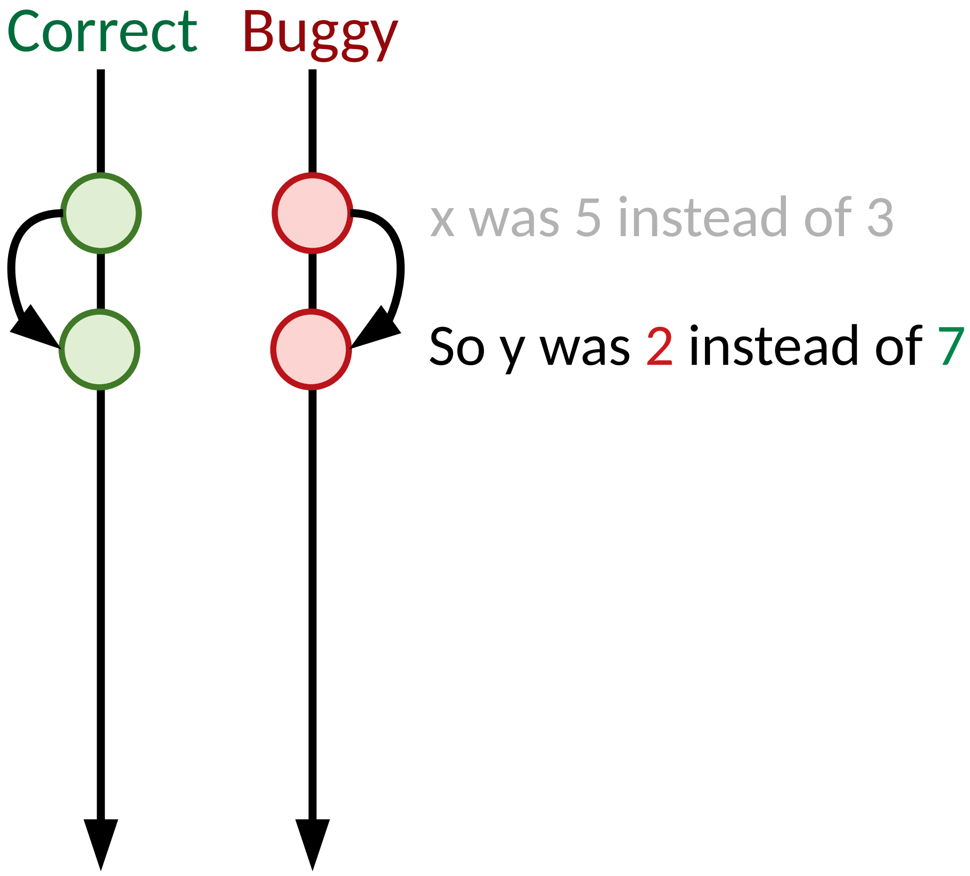
How it might look



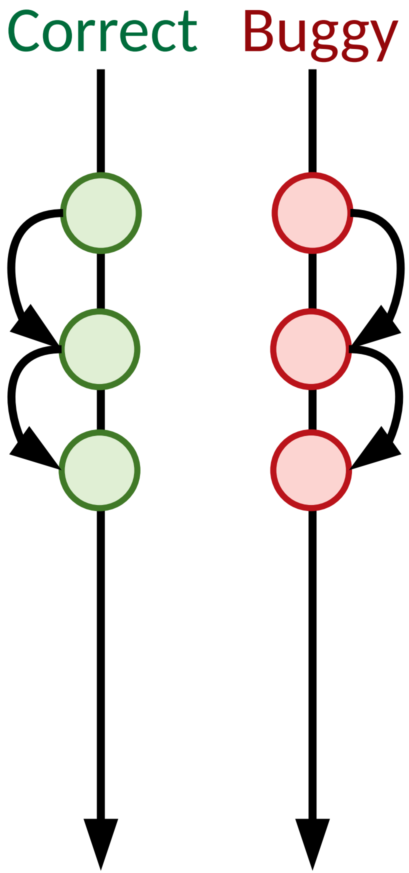
How it might look



How it might look



How it might look

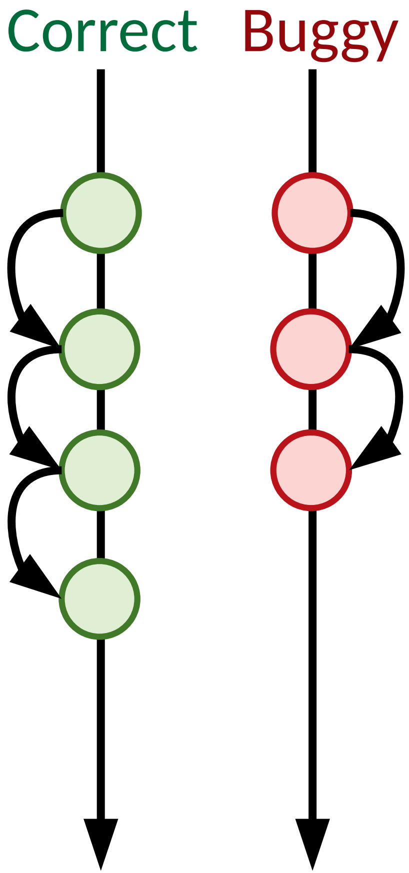


x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

How it might look



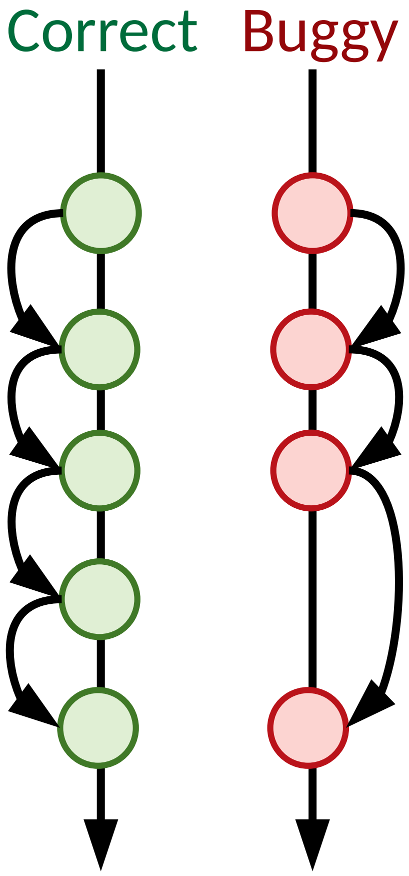
x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

So the update of z was skipped

How it might look



x was 5 instead of 3

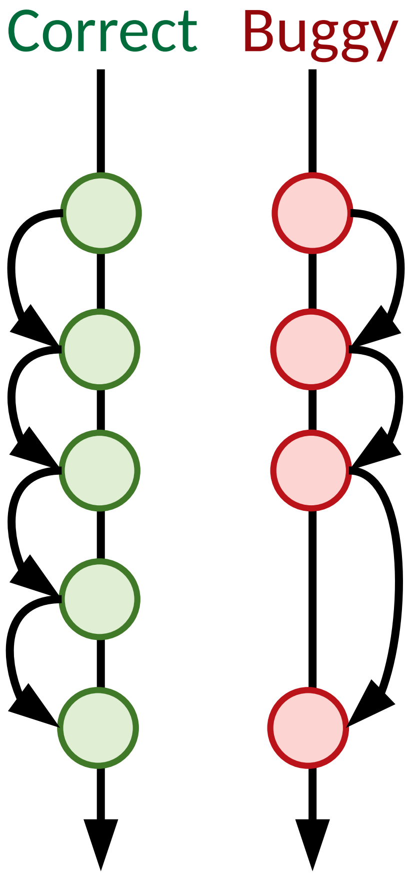
So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

So the update of z was skipped

So the incorrect value of z was printed

How it might look



x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

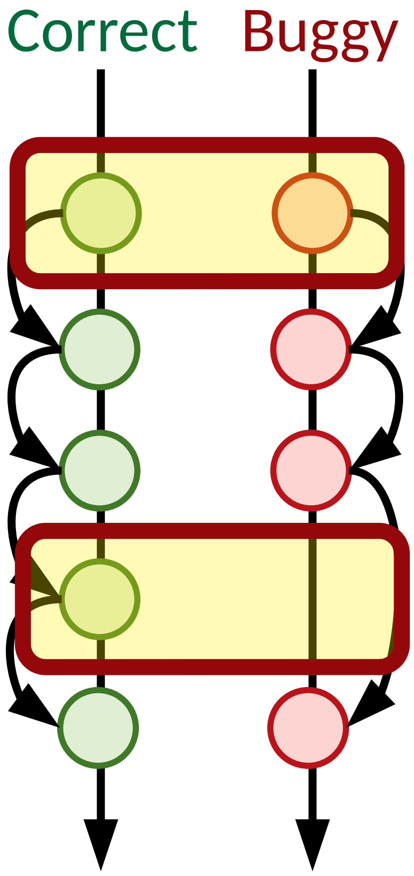
So the update of z was skipped

So the incorrect value of z was printed

What do we need?

- locations
- state
- flow
- causation

How it might look



x was 5 instead of 3

So y was 2 instead of 7

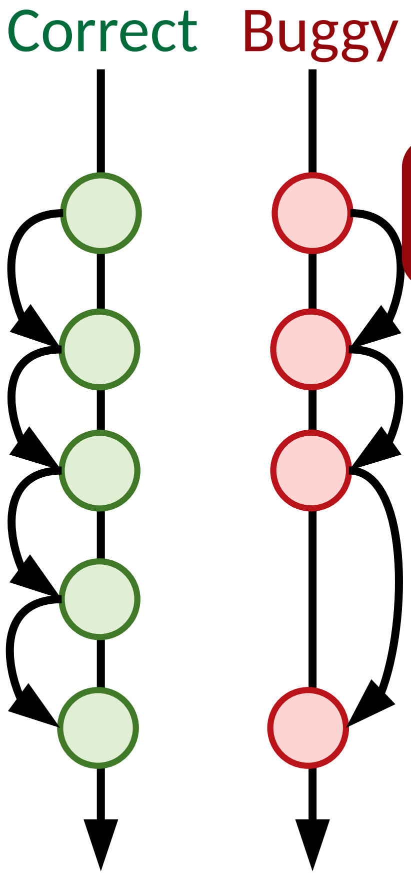
So the TRUE branch executed
instead of the FALSE branch

So the update of z was skipped

So the incorrect value of z was printed

What do we need?
- locations
- state
- flow
- causation

How it might look



x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

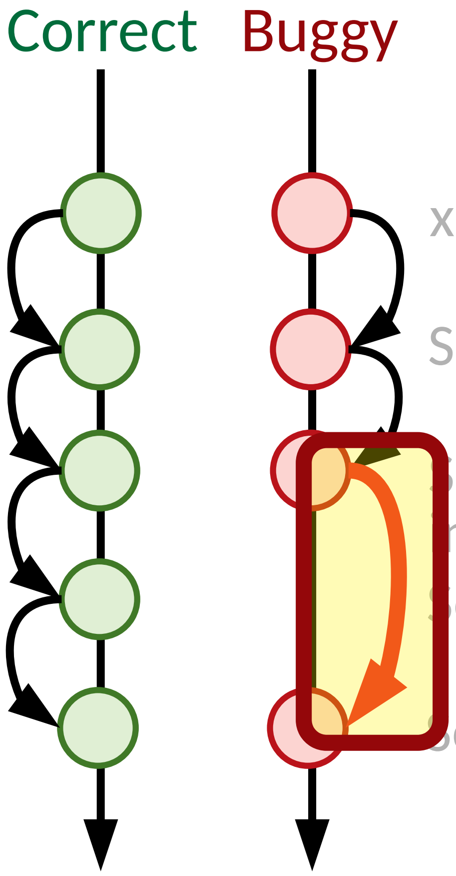
So the update of z was skipped

So the incorrect value of z was printed

What do we need?

- locations
- state
- flow
- causation

How it might look



x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed instead of the FALSE branch

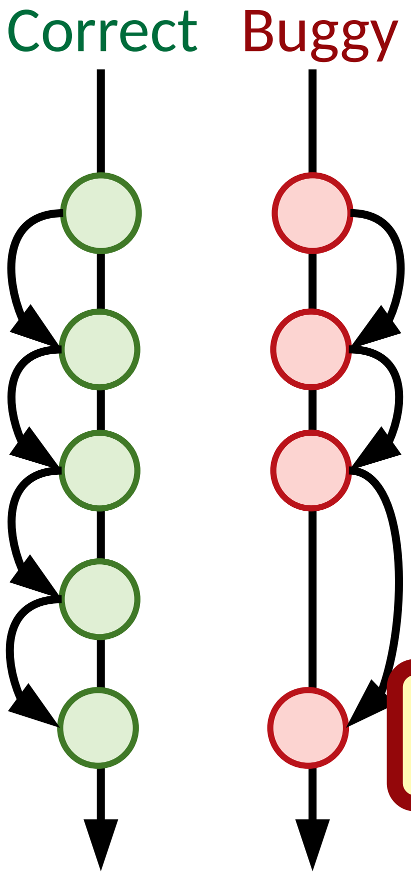
So the update of z was skipped

So the incorrect value of z was printed

What do we need?

- locations
- state
- flow
- causation

How it might look



x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of the FALSE branch

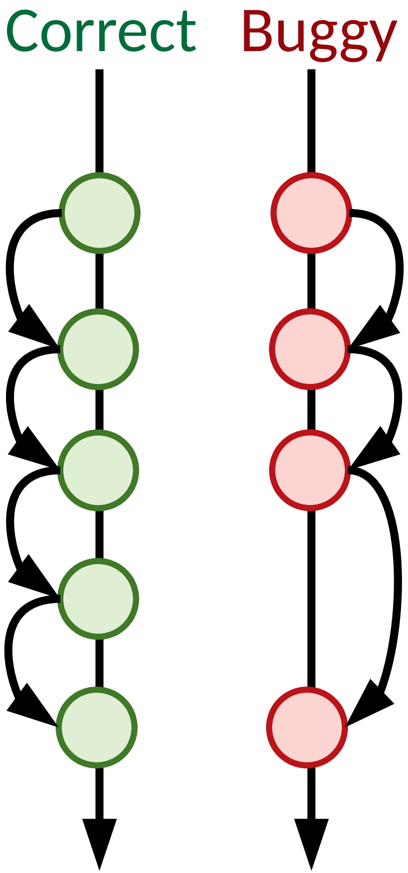
So the update of z was skipped

So the incorrect value of z was printed

What do we need?

- locations
- state
- flow
- causation

How it might look



x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed
instead of t

So the upda

So the incorrect value of z was printed

What do we need?
- locations
- state
- flow
- causation

We can construct this backward from a point of failure/difference

So why not just…

- Traces can be viewed as sequences….
  - Why not just do LCS based sequence alignment?

So why not just...

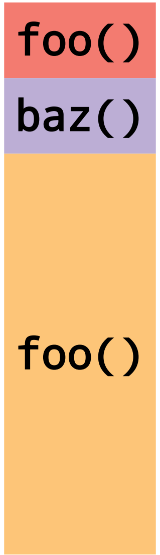
- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```

So why not just…

- Traces can be viewed as sequences….
  - Why not just do LCS based sequence alignment?

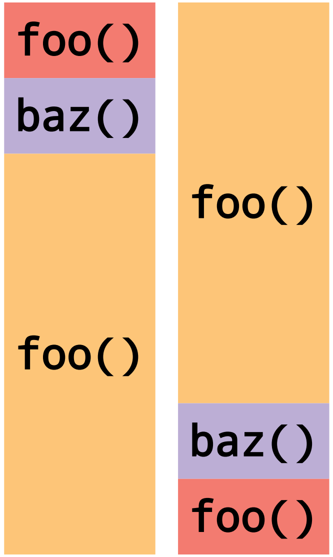
```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```



So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

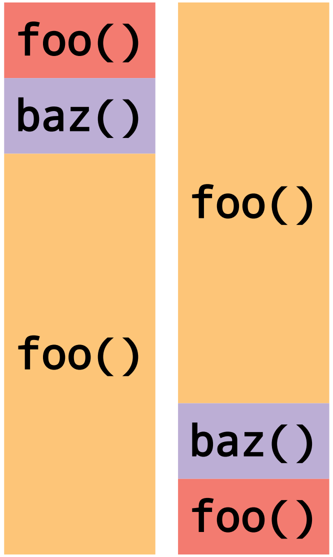
```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```



So why not just…

- Traces can be viewed as sequences….
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```

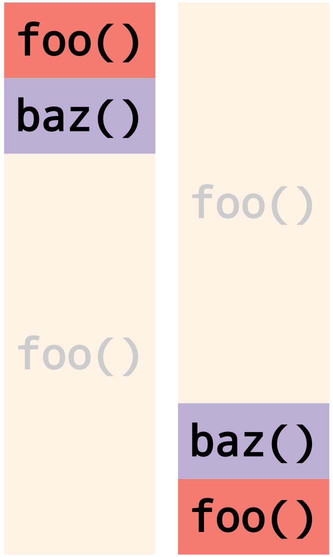


What is marked as *different*?

So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```

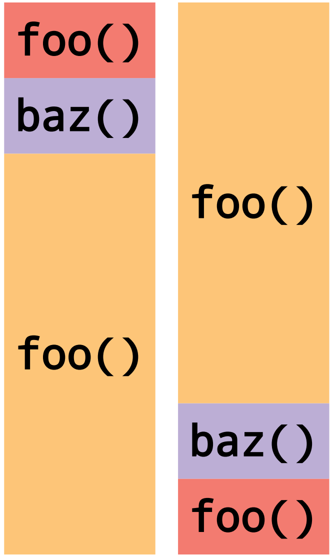


What is marked as *different*?

So why not just…

- Traces can be viewed as sequences….
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```



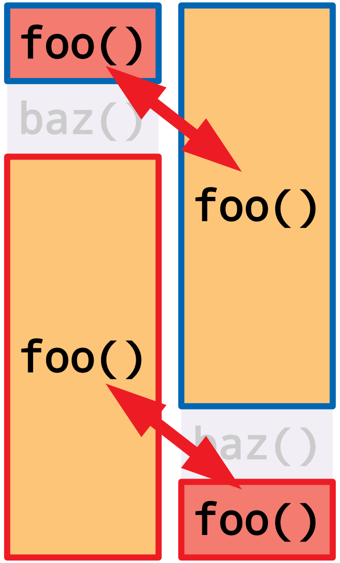
What is marked as *different*?

What is *intuitively* different?

So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```



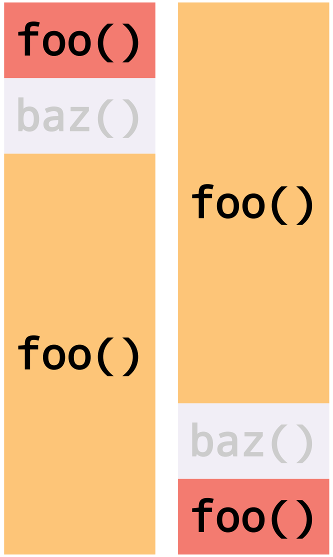
What is marked as *different*?

What is *intuitively* different?

So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```
def foo(int c):
  if c:
    while bar():
      ...

foo(...)
baz(...)
foo(...)
```



What is marked as *different*?

What is *intuitively* different?

Execution comparison must account for what a program *means* and *does*!

The big picture

- Fundamentally, execution comparison needs to account for

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure – How is a program organized?

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure – How is a program organized?
  - Value – What are the values in the different executions?

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure – How is a program organized?
  - Value – What are the values in the different executions?
  - Semantics – How is the meaning of the program affected by the differences?

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics
- We can attack these through

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics
- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics
- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?

The big picture

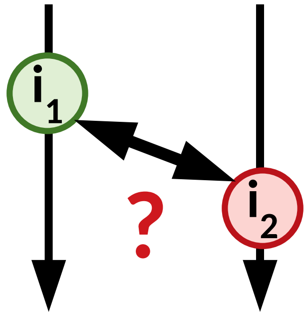
- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics
- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?
  - Slicing
    - How do differences transitively flow through a program?

The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics
- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?
  - Slicing
    - How do differences transitively flow through a program?
  - Causality testing
    - Which differences actually induce difference behavior?

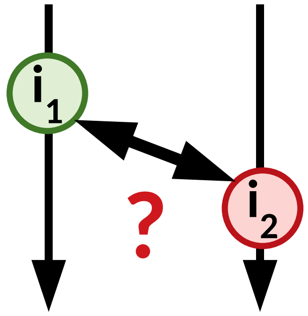
Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?



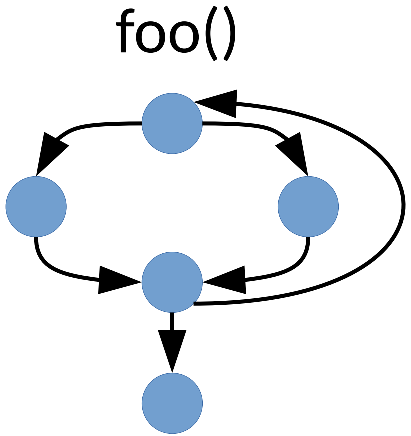
Temporal Alignment

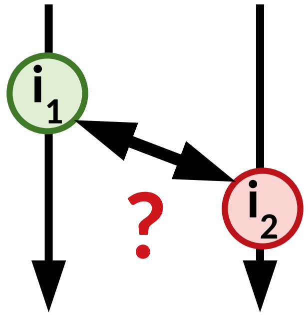
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]



Temporal Alignment

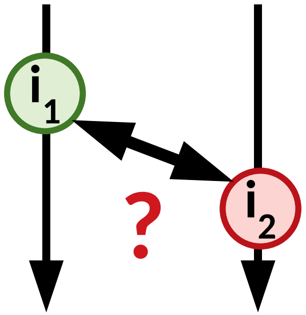
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

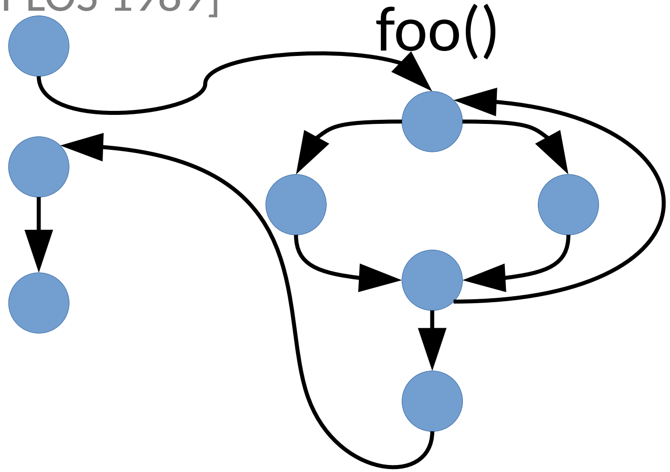




Temporal Alignment

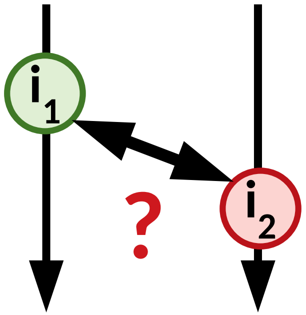
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

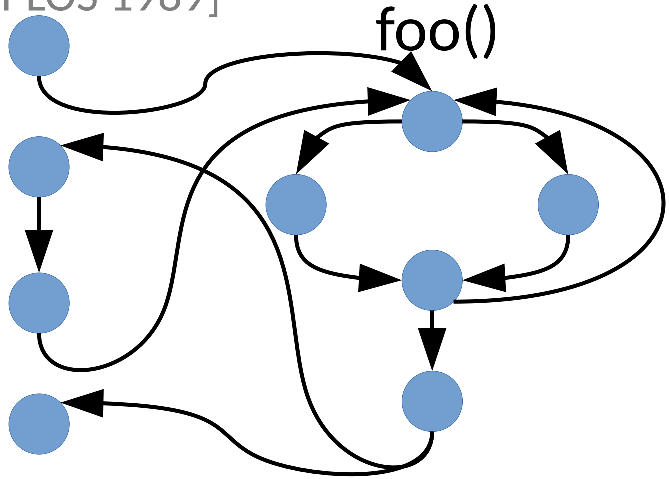




Temporal Alignment

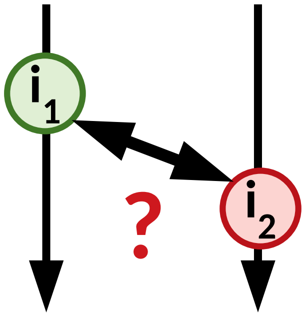
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

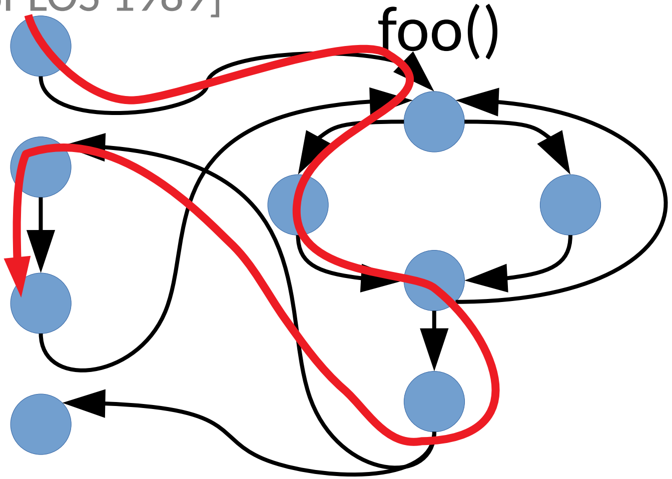




Temporal Alignment

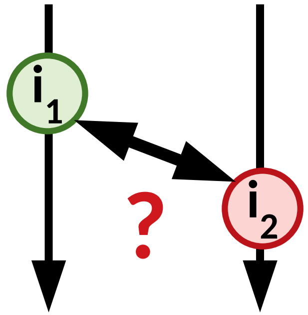
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

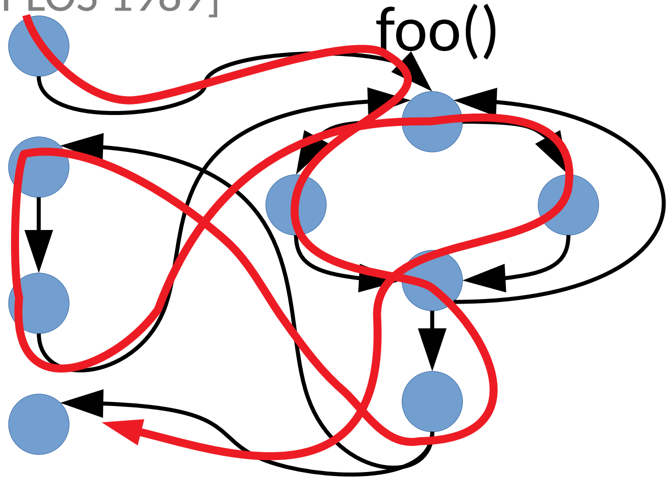




Temporal Alignment

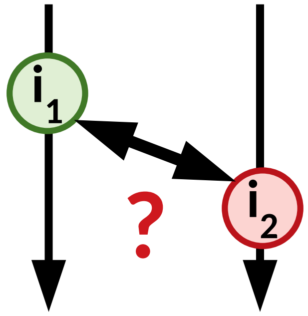
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]

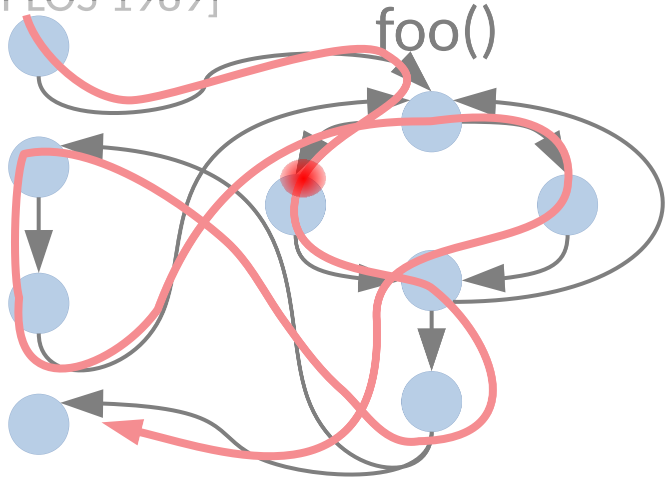




Temporal Alignment

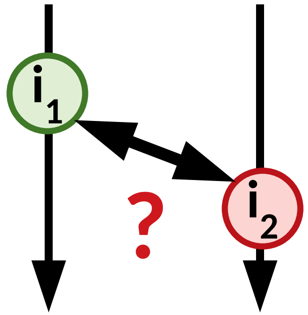
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]

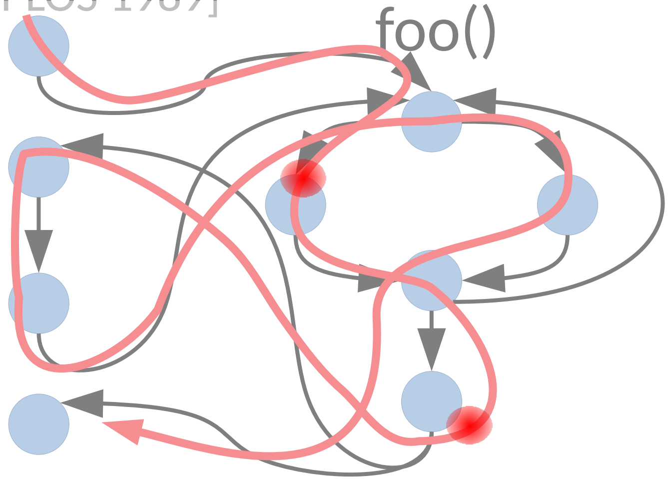




Temporal Alignment

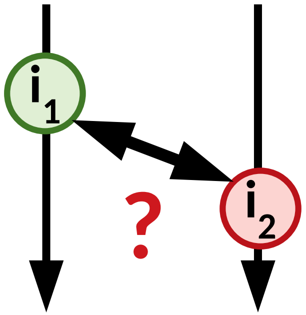
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

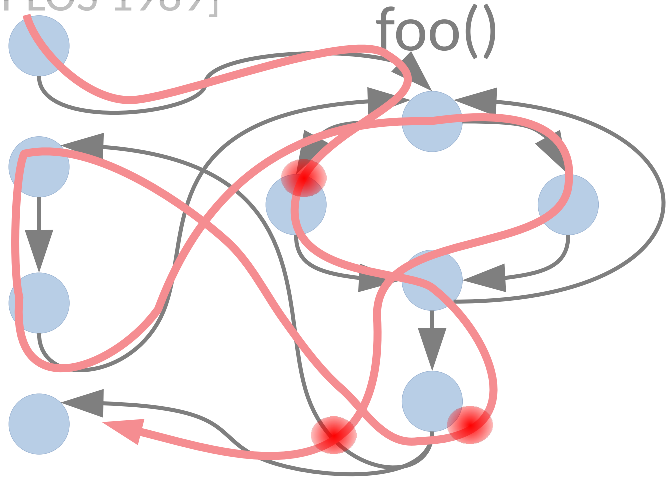




Temporal Alignment

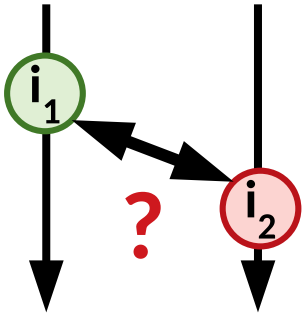
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]

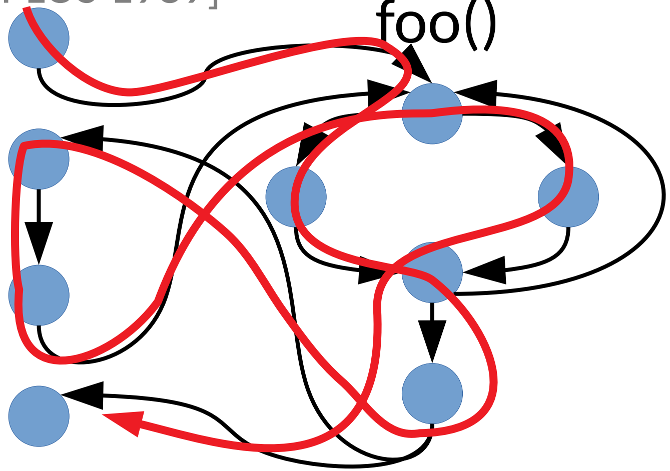




Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

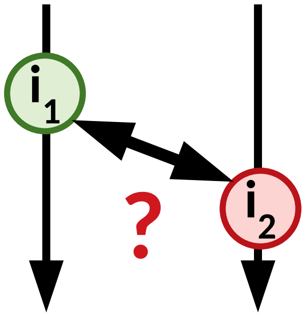


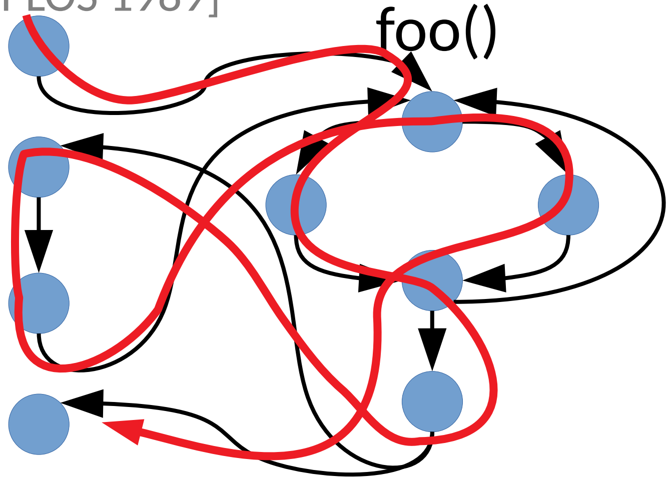


Position along a path can be maintained via a counter

Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path
  [Mellor-Crummey, ASPLOS 1989]



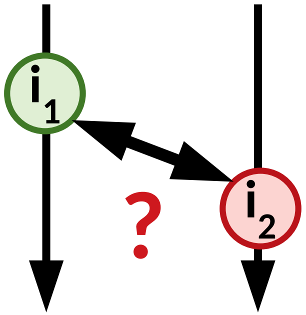


Position along a path can be maintained via a counter

Only need to increment along
1) back edges
2) function calls

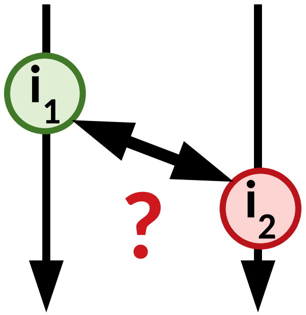
Temporal Alignment

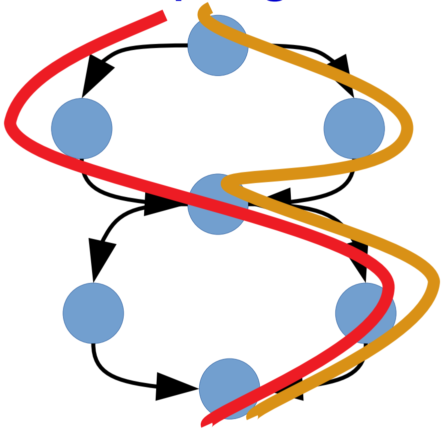
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?



Temporal Alignment

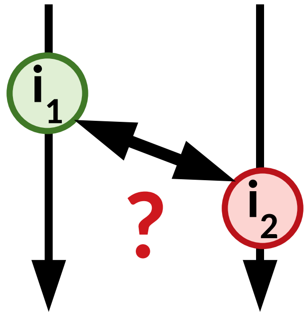
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?

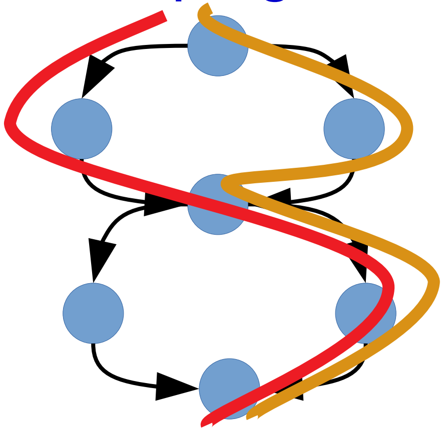




Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?

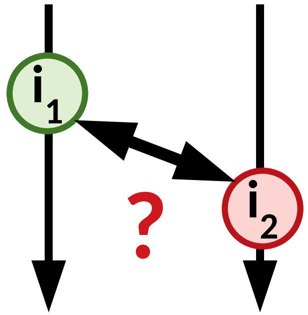




The *position* in the DAG relates the paths

Temporal Alignment

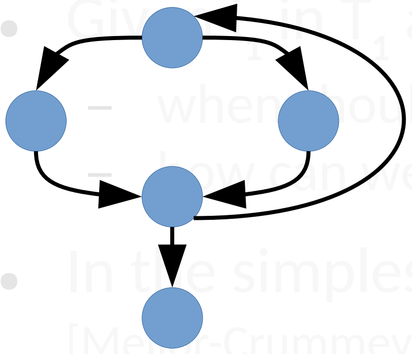
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?
- Now consider the case where cycles can occur… [Sumner, ASE 2013]



How can we extend the acyclic case?

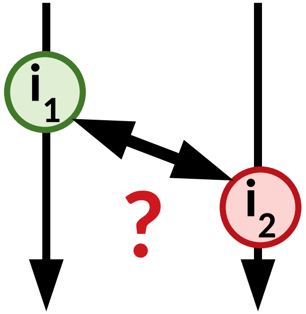
Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - How can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?



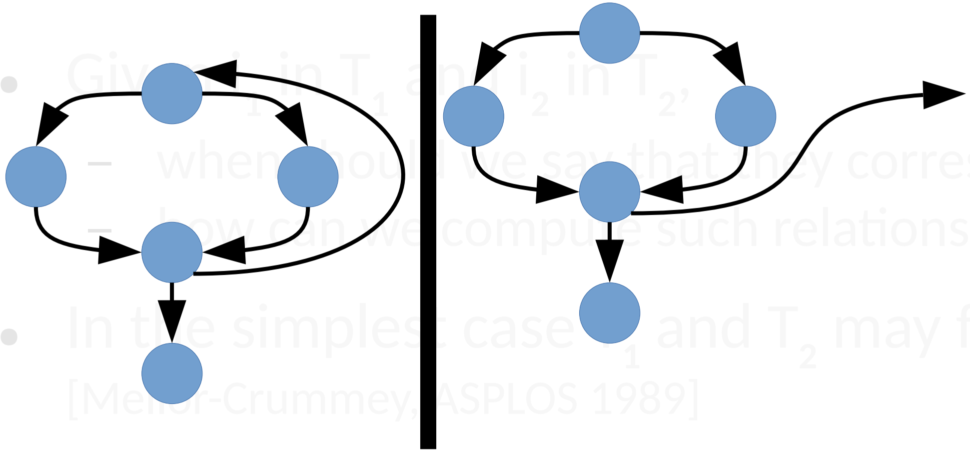
We can unwind the loop to make it logically acyclic

- Now consider the case where cycles can occur... [Sumner, ASE 2013]

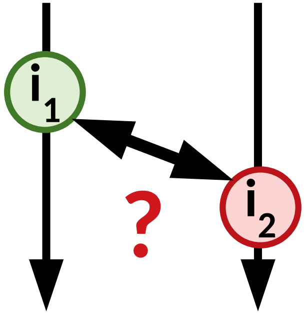


How can we extend the acyclic case?

Temporal Alignment

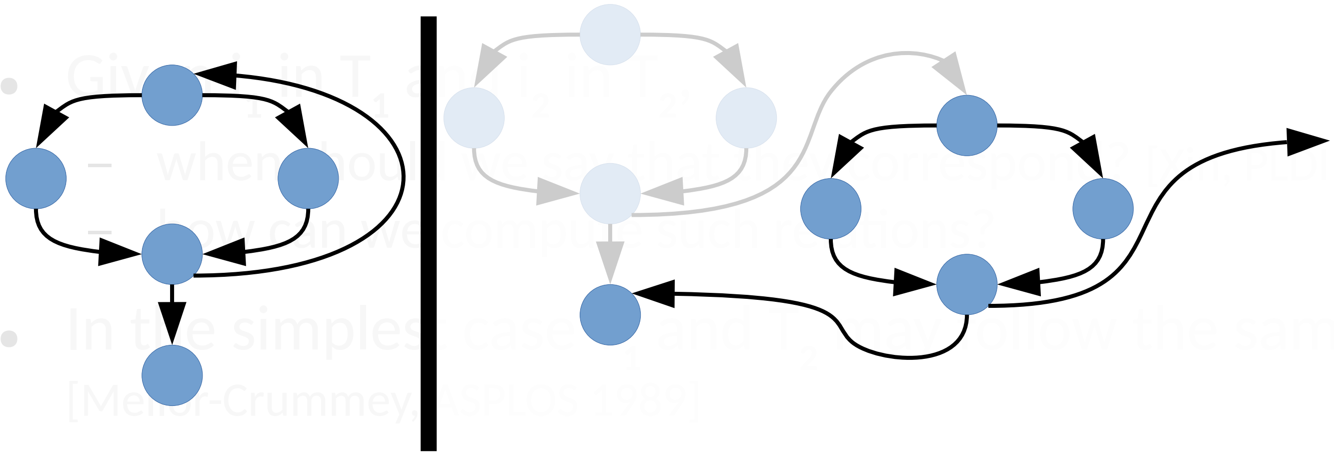


- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - How can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?
- Now consider the case where cycles can occur… [Sumner, ASE 2013]



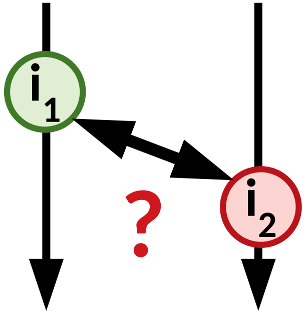
How can we extend the acyclic case?

Temporal Alignment



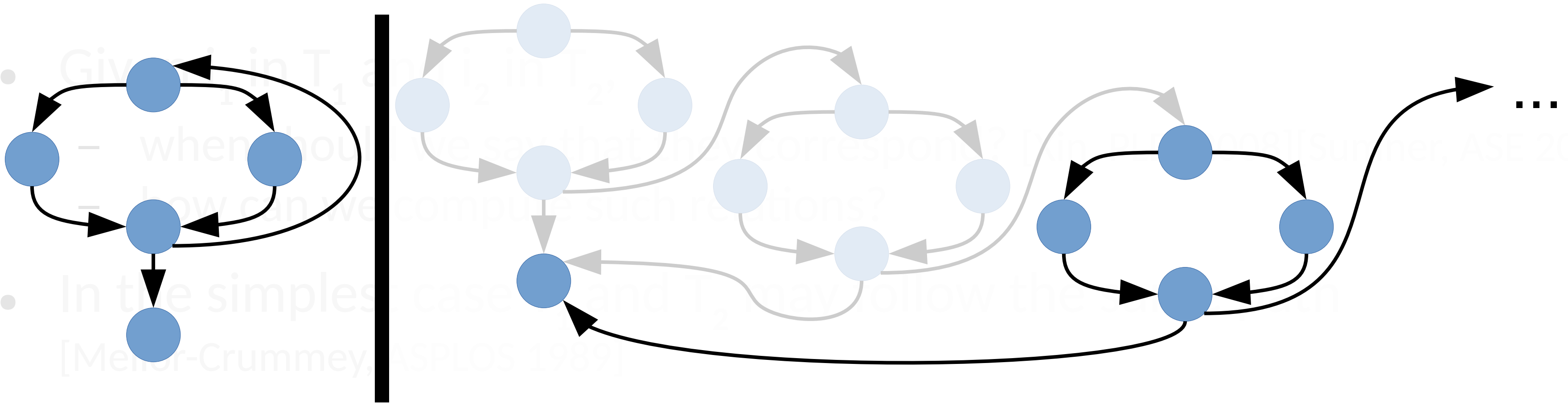
- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]



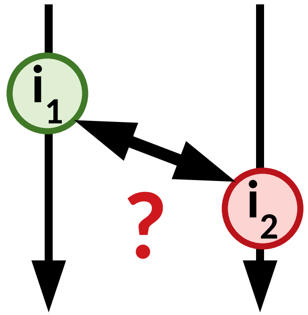
How can we extend the acyclic case?

Temporal Alignment



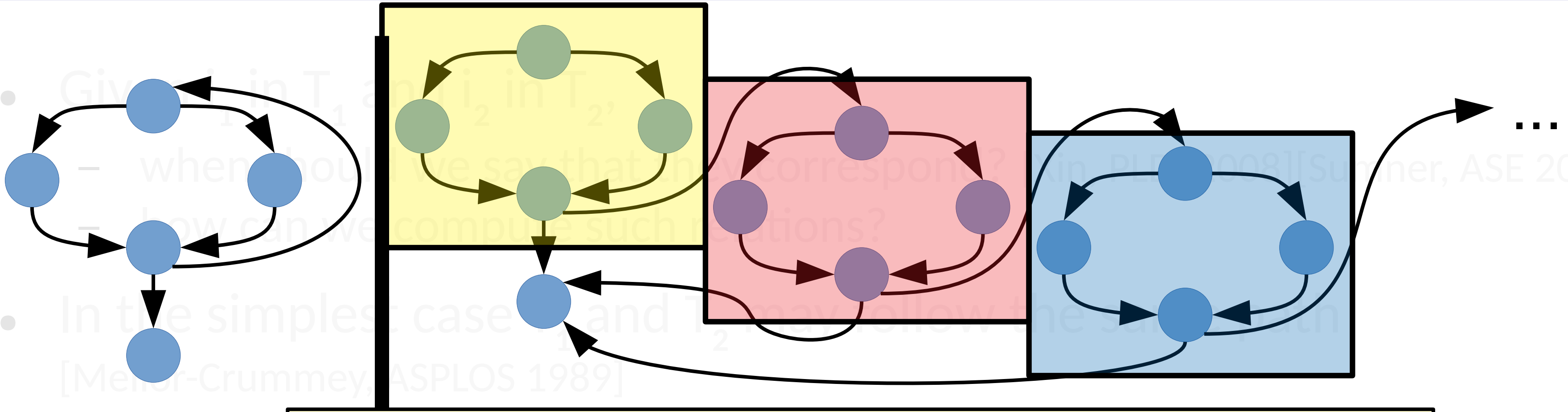
- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur… [Sumner, ASE 2013]



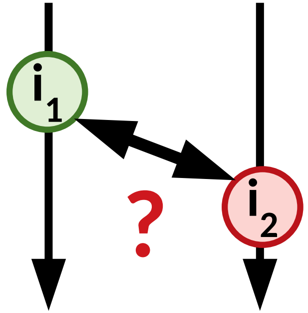
How can we extend the acyclic case?

Temporal Alignment



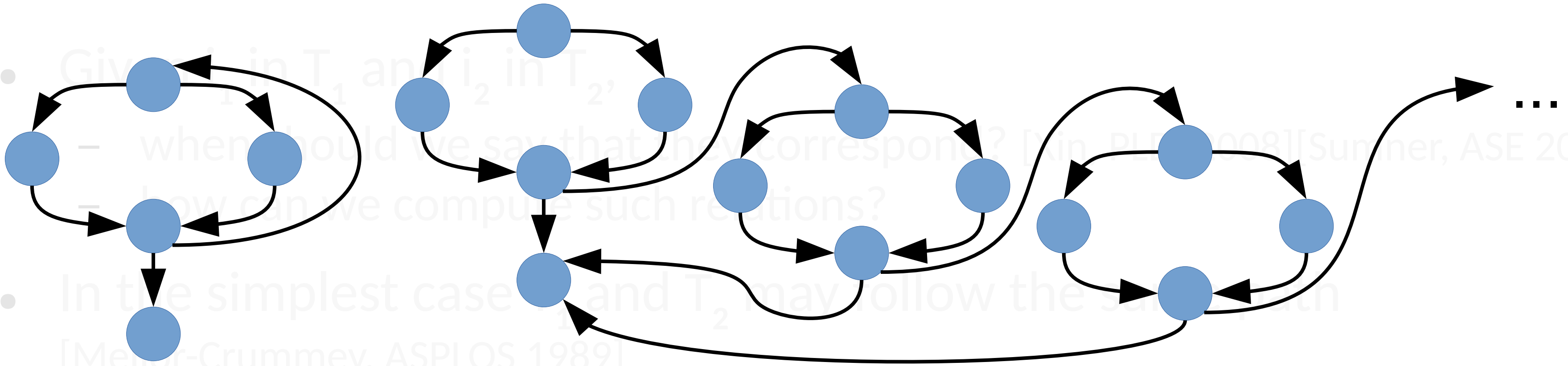
These are different iterations of one loop.
A counter for *each active loop* suffices (mostly).

- Now consider the case where cycles can occur… [Sumner, ASE 2013]

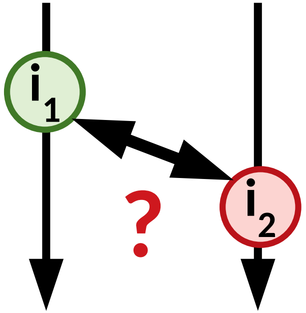


How can we extend the acyclic case?

Temporal Alignment



- Now consider the case where cycles can occur… [Sumner, ASE 2013]

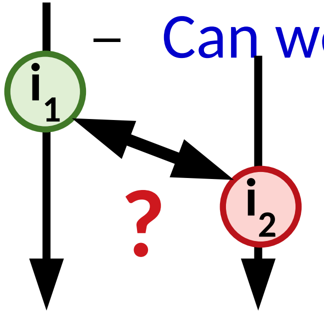


How can we extend the acyclic case?

1 counter per active loop
+ the call stack disambiguates!

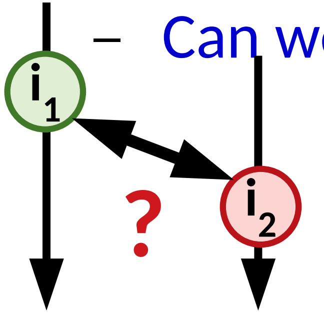
Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?
- Now consider the case where cycles can occur... [Sumner, ASE 2013]
  - Can we ***efficiently*** represent this?



Temporal Alignment

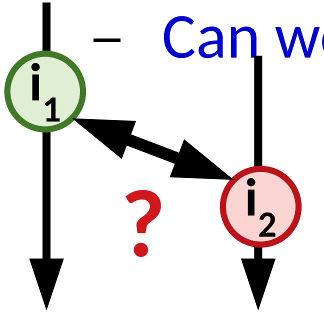
- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?
- Now consider the case where cycles can occur… [Sumner, ASE 2013]
  - Can we efficiently represent this?

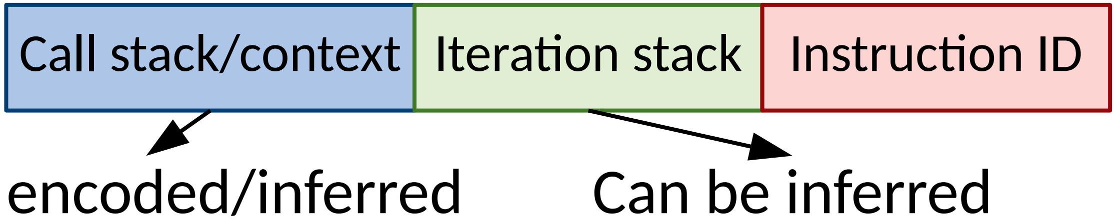


Call stack/context	Iteration stack	Instruction ID

Temporal Alignment

- Given i_1 in T_1 and i_2 in T_2,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
- In the simplest case T_1 and T_2 may follow the same path [Mellor-Crummey, ASPLOS 1989]
- Suppose that we know the programs are acyclic?
- Now consider the case where cycles can occur… [Sumner, ASE 2013]
  - Can we efficiently represent this?





Spatial Alignment

- We must also ask what it means to compare program *state* across executions

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?

3 != 5

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?

0xdeadbeef in T1 $\stackrel{?}{=}$ 0xcafef00d in T2

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?

0xdeadbeef in T1 $\stackrel{?}{=}$ 0xcafef00d in T2

If you allocated other stuff in only one run,
this can be true even without ASLR!

Spatial Alignment

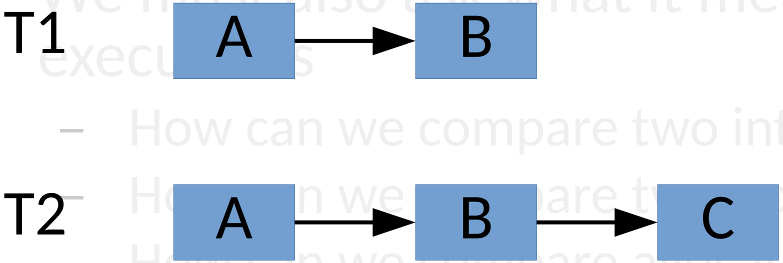
- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap?
    Should they even *be compared*?!?

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even *be compared*?!?
- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap?
    Should they even *be compared*?!?

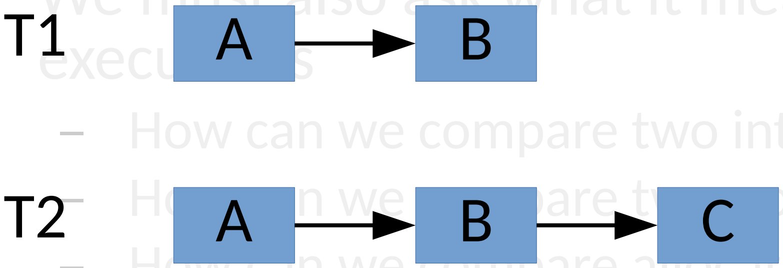


- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

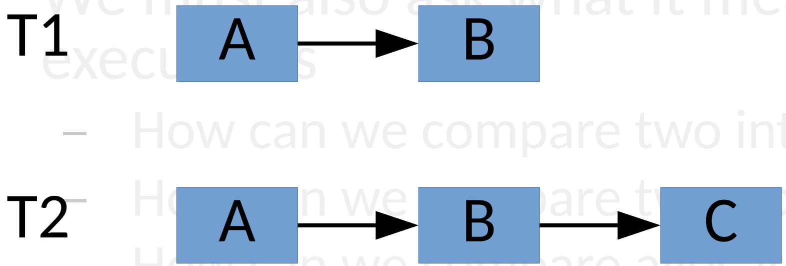
What are the differences?

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even *be compared*?!?



- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment



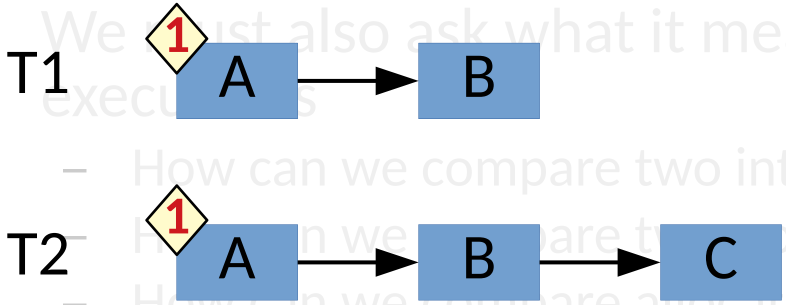
What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

- We ... also ask what it means to c... execu...s
  - How can we compare two integers X
  - H... n we ... are t... pointers A
  - How can we compare a...ed regions on the heap? Should they even *be compared*?!?

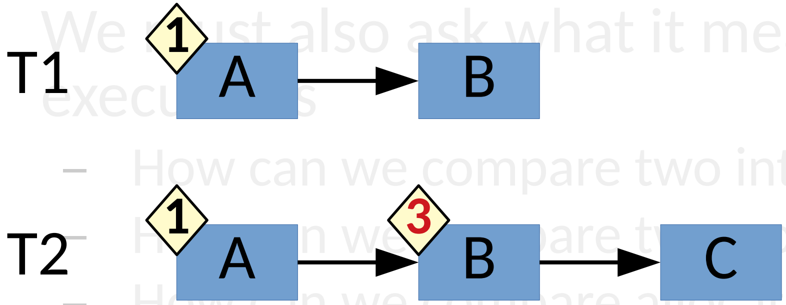


What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment



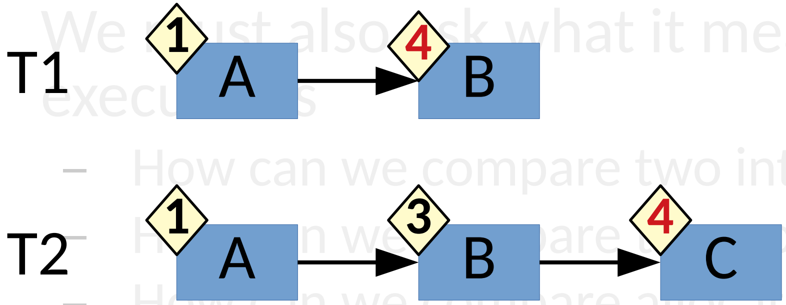
What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers X
  - How can we compare pointers A
  - How can we compare allocated regions on the heap? Should they even *be compared*?!?

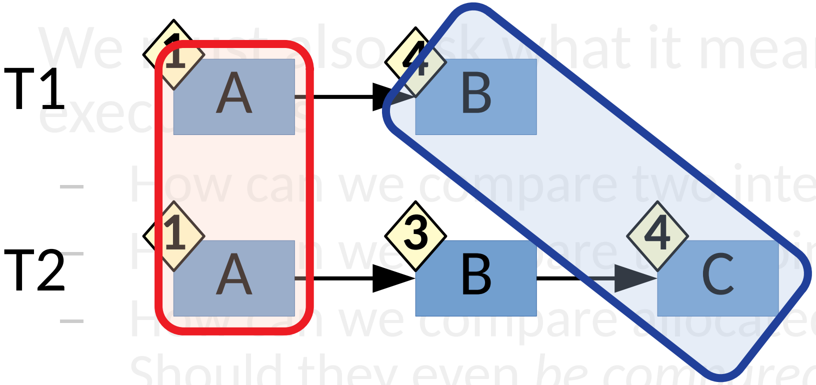


What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

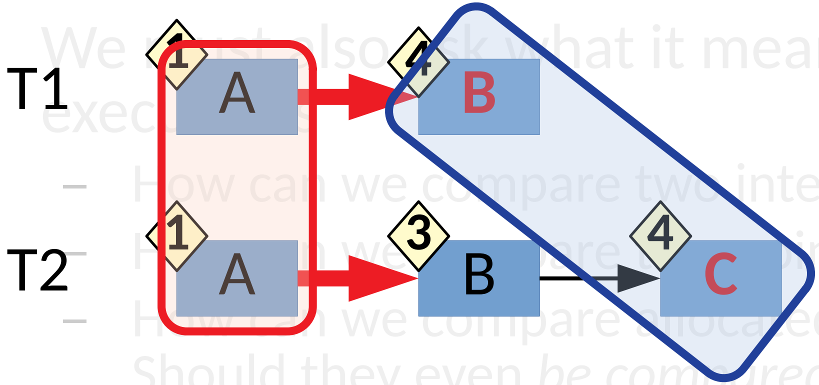


What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment



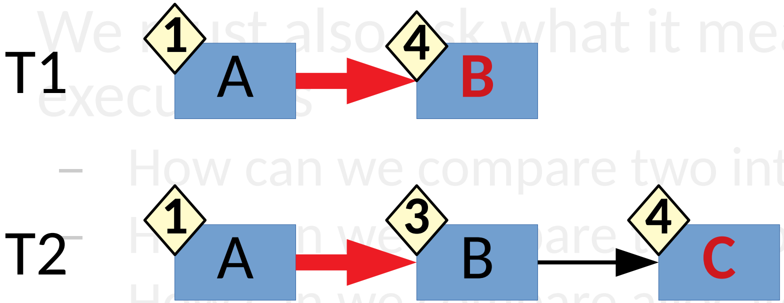
What are the differences?

```
1) list.append(value++)
2) if c:
3)     list.append(value++)
4) list.append(value++)
```

- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)

Spatial Alignment

- We must also ask what it means to compare program *state* across executions
  - How can we compare two integers X and Y?
  - How can we compare pointers A and B?
  - How can we compare allocated regions on the heap? Should they even *be compared*?!?



- In practice, comparing state across executions requires comparing *memory graphs*
  - We need a way to identify corresponding nodes (state elements)
- Again, the *semantics* of the program dictate the solution
  - Identify heap allocations by the *aligned time* of allocation

Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times

Dual Slicing

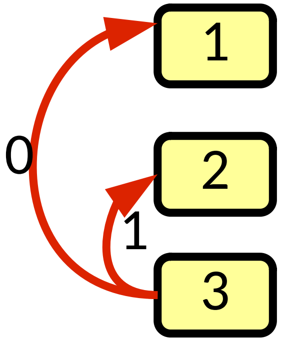
- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)

Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  - Based on classic dynamic slicing
  - Include transitive dependencies that differ or exist in only 1 execution

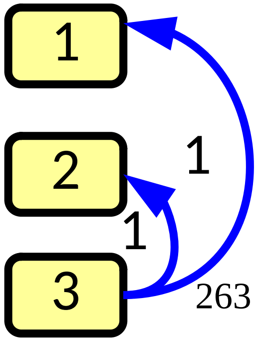
Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  - Based on classic dynamic slicing
  - Include transitive dependencies that differ or exist in only 1 execution



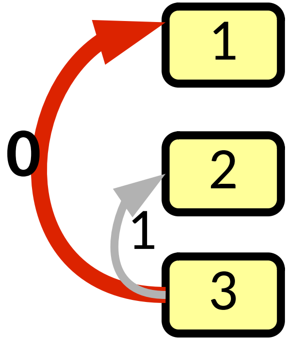

```
1)x ← 0
2)y ← 1
3)print(x+y)
```

```
1)x ← 1
2)y ← 1
3)print(x+y)
```



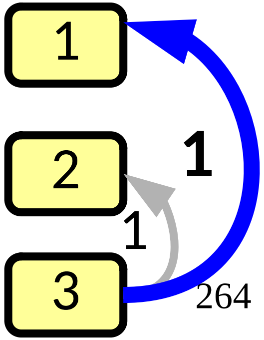
Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  - Based on classic dynamic slicing
  - Include transitive dependencies that differ or exist in only 1 execution



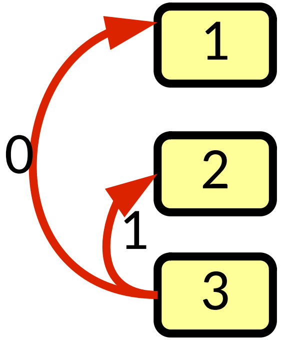

```
1)x ← 0
2)y ← 1
3)print(x+y)
```

```
1)x ← 1
2)y ← 1
3)print(x+y)
```

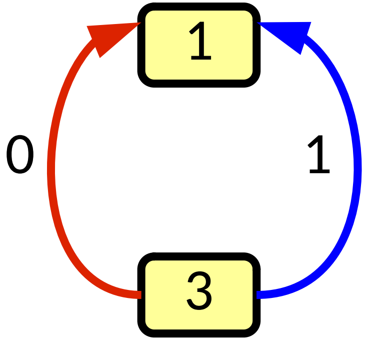


Dual Slicing

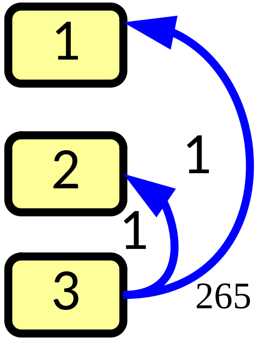
- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  - Based on classic dynamic slicing
  - Include transitive dependencies that differ or exist in only 1 execution




```
1)x ← 0
2)y ← 1
3)print(x+y)
```




```
1)x ← 1
2)y ← 1
3)print(x+y)
```

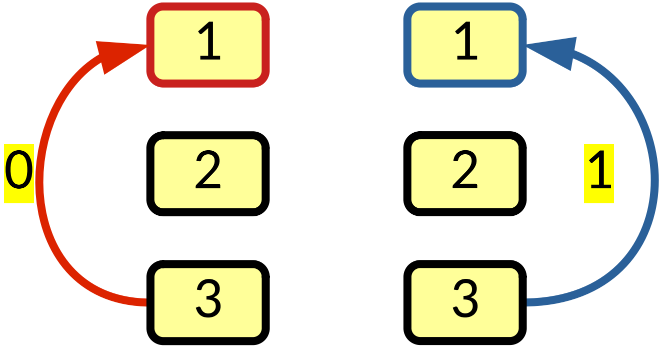


Dual Slicing

- The differences in dependencies capture multiple kinds of information

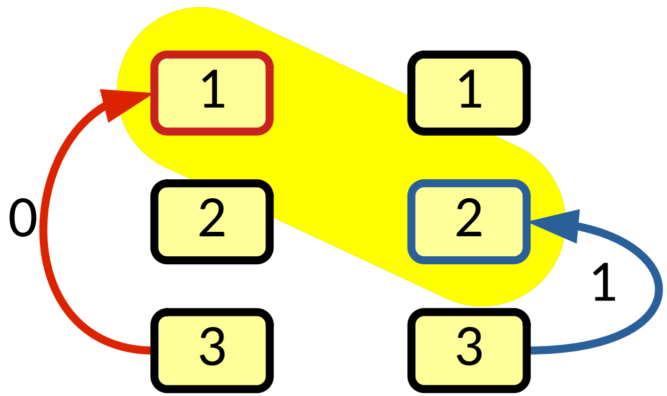
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences



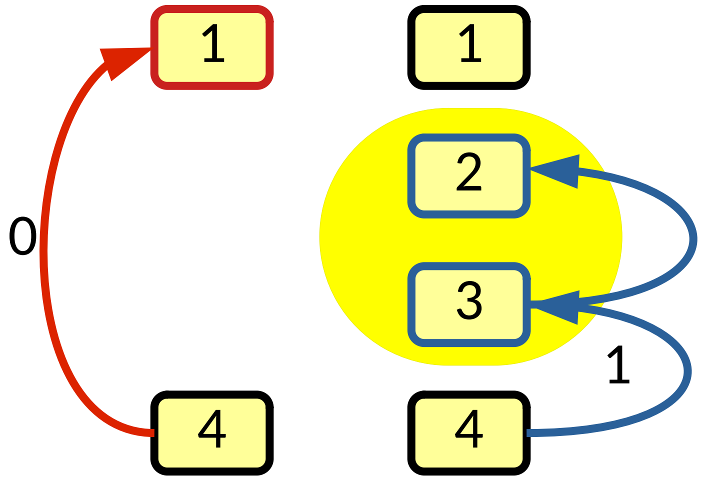
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences



Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior



Dual Slicing

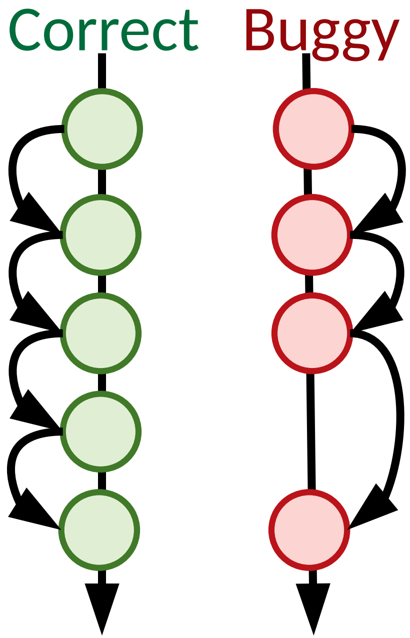
- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior
- Recall: Dynamic slicing could not handle execution omission, but dual slicing can!

Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior
- Recall: Dynamic slicing could not handle execution omission,
  but dual slicing can!
- Dual slices can be effective for concurrent debugging & exploit analysis
  [Weeratunge, ISSTA 2010][Johnson, S&P 2011]

Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]



Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)     z = 0
5) else:
6)     z = 1
7) print(z)
```

Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)     z = 0
5) else:
6)     z = 1
7) print(z)
```

Correct

```
x = 10
y = -1
True
z = 0


"0"
```

Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

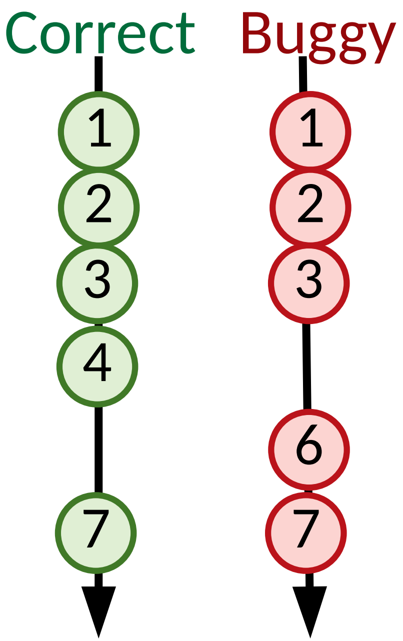
Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"

Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"

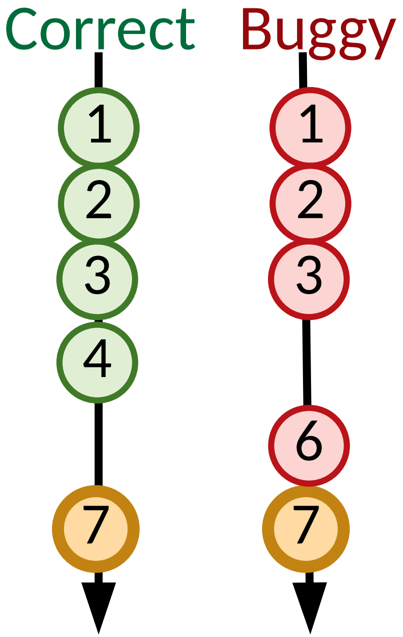


Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"

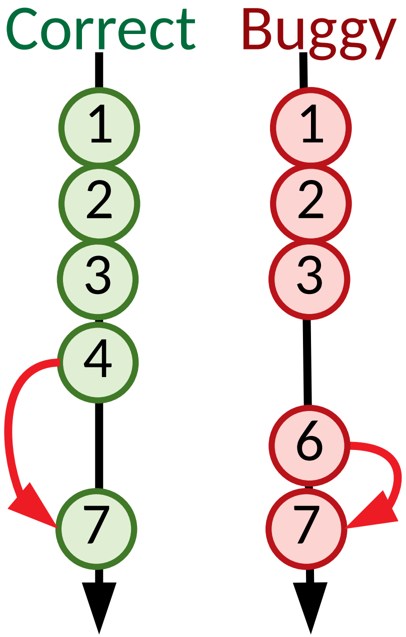


Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)     z = 0
5) else:
6)     z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"

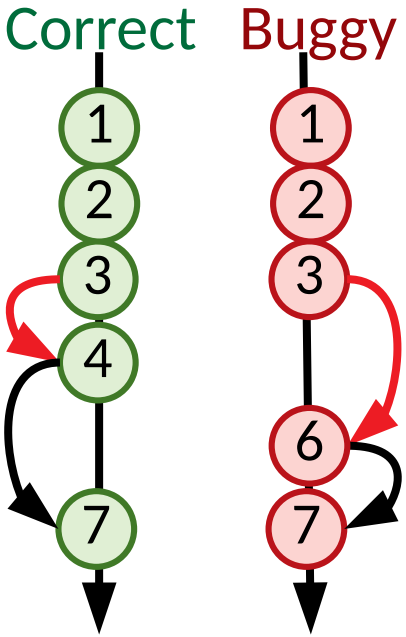


Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
“0”	“1”

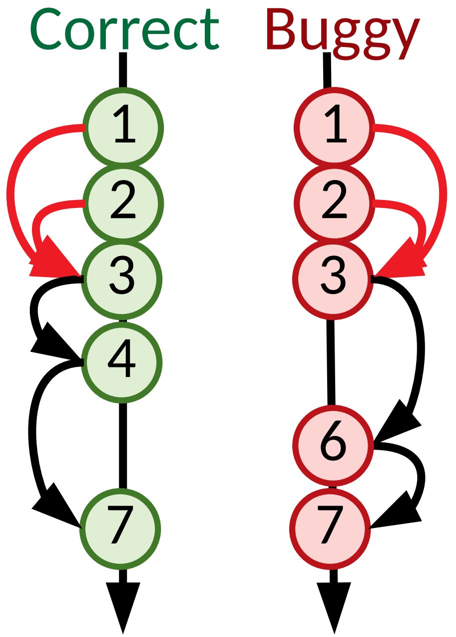


Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)       z = 0
5) else:
6)       z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
“0”	“1”



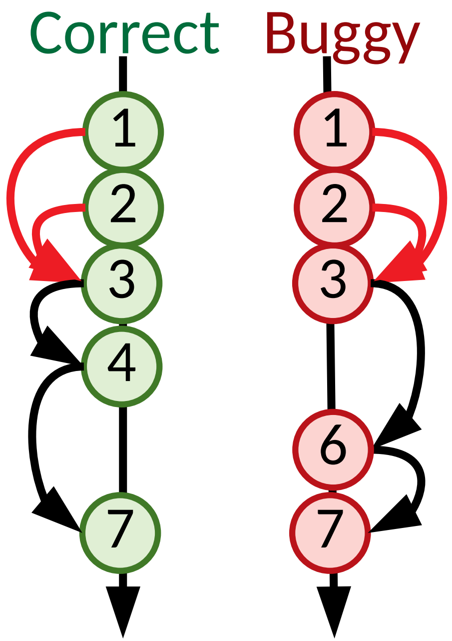
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise…

Dual slicing captures *differences*, not ***causes***.
What does that mean here?

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)       z = 0
5) else:
6)       z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"

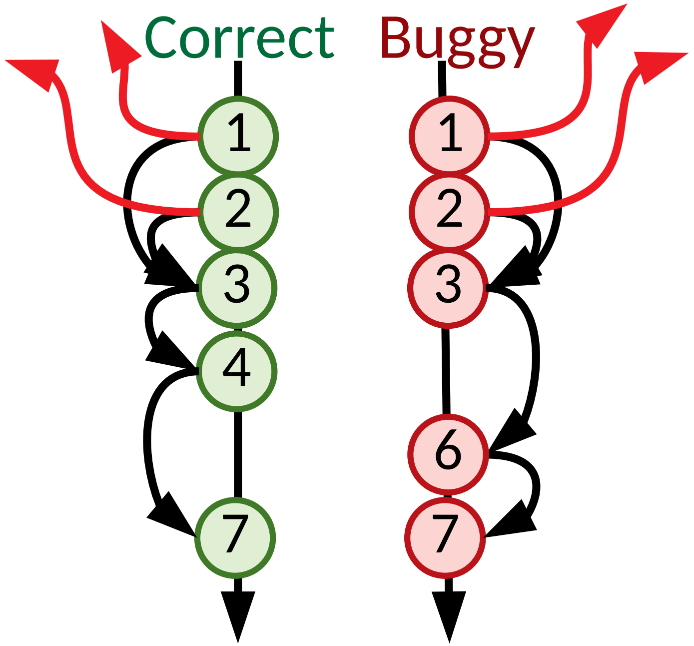


Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer "Why" and "Why not" questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
"0"	"1"



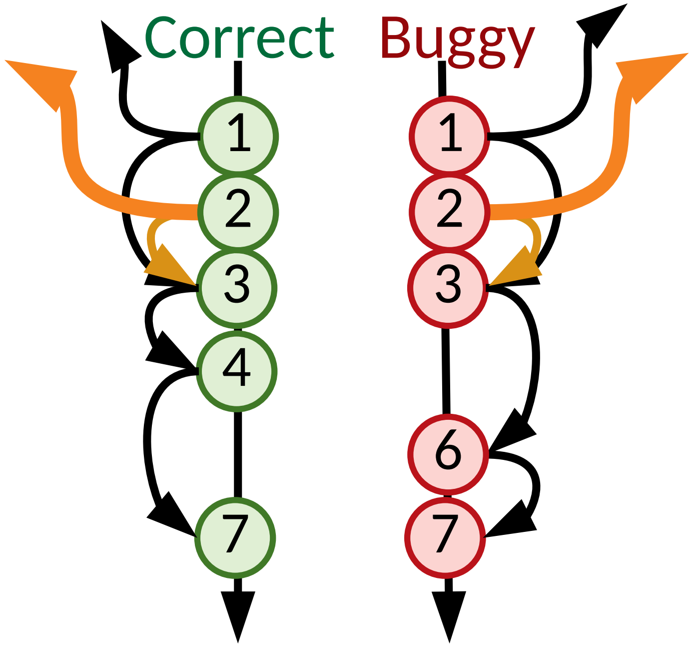
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and
    [Ko, ICSE 2008]
  - But they may still conta

The cost of these extra edges is high in practice!
All transitive dependencies…

```
1) x = ...
2) y = ...
3) if x + y > 0:
4)      z = 0
5) else:
6)      z = 1
7) print(z)
```

Correct	Buggy
x = 10	x = 0
y = -1	y = -2
True	False
z = 0	
	z = 1
“0”	“1”

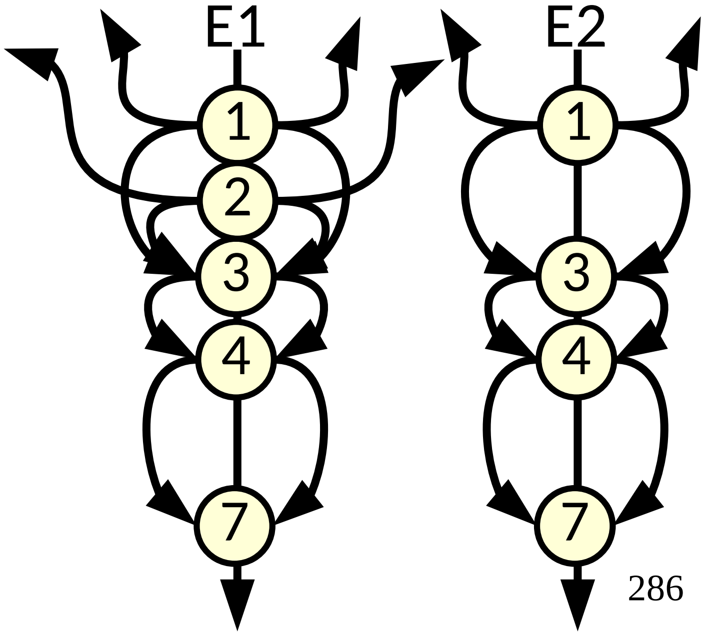


Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research

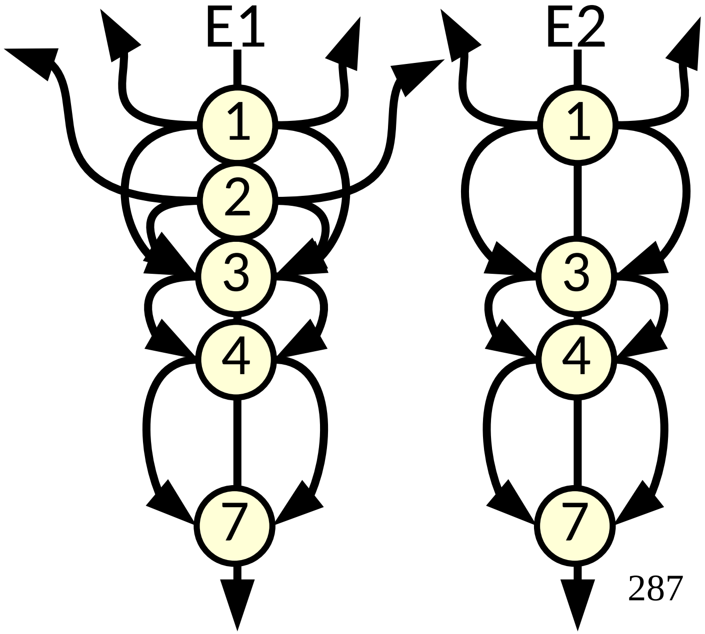
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?



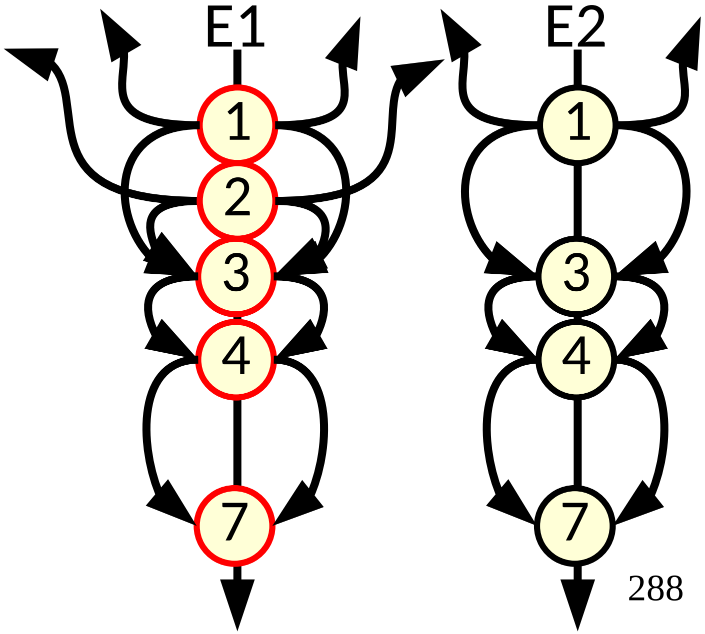
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred



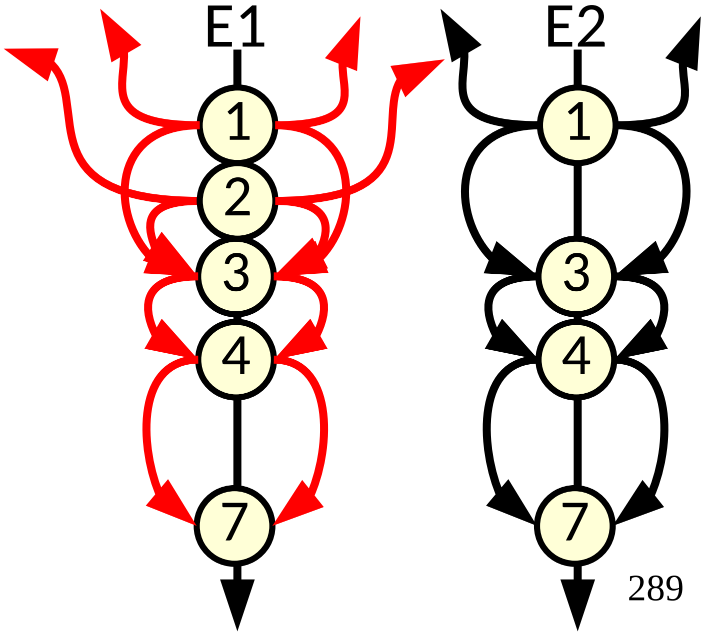
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?



Adding Causation

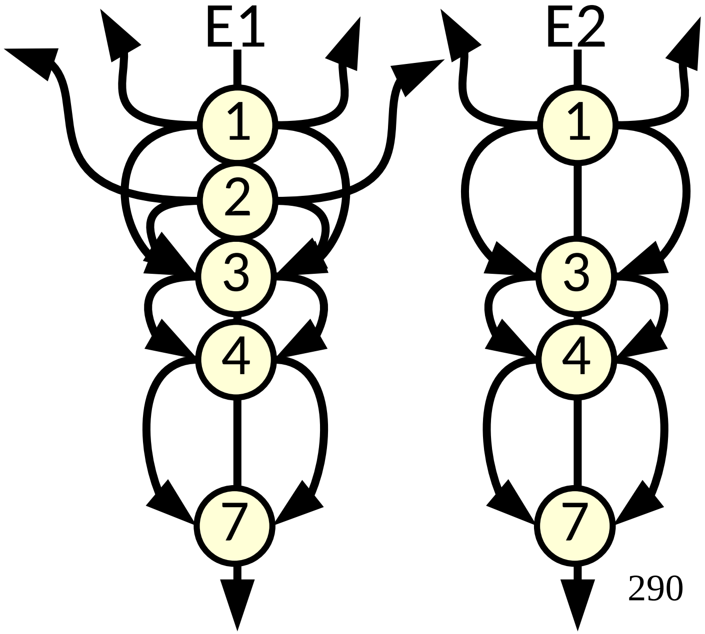
- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?



Adding Causation

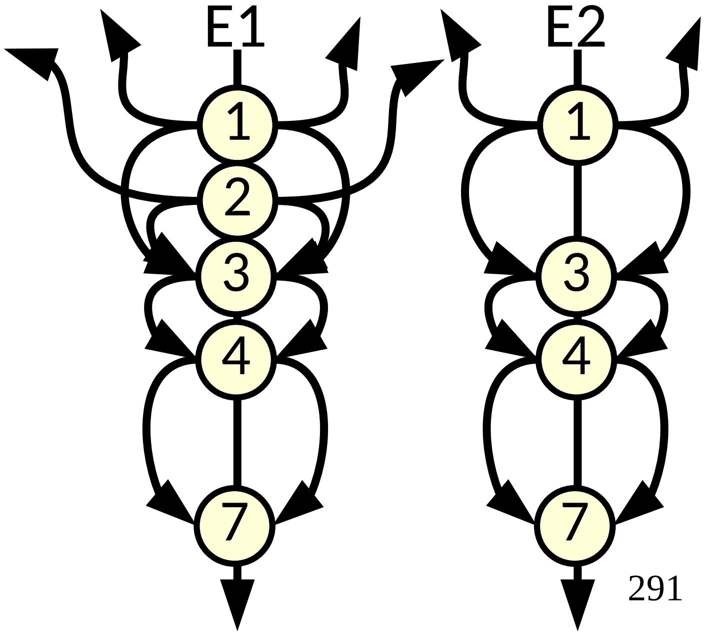
- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?

What big challenges do you see with these 2 approaches?



Adding Causation

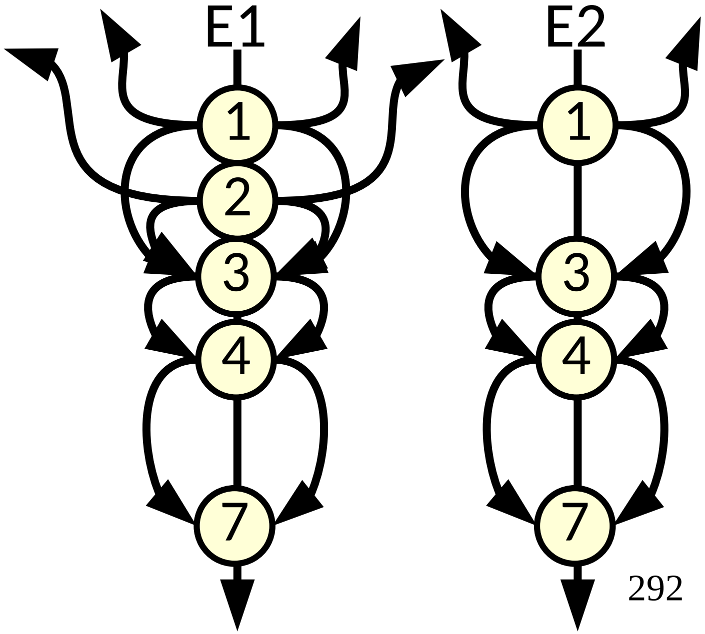
- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# *local* dependencies”?



Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# *local* dependencies”?

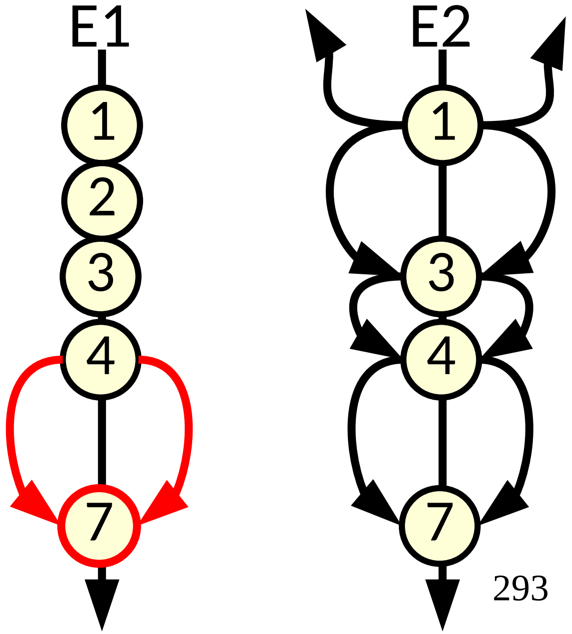
$$argmin_{s \in sd_i} |sd|$$
$$\wedge E1[sd(E2)_i] \rightarrow sd(E2)_{i+1}$$
$$\wedge E2[sd(E1)_i] \rightarrow sd(E1)_{i+1}$$



Adding Causation

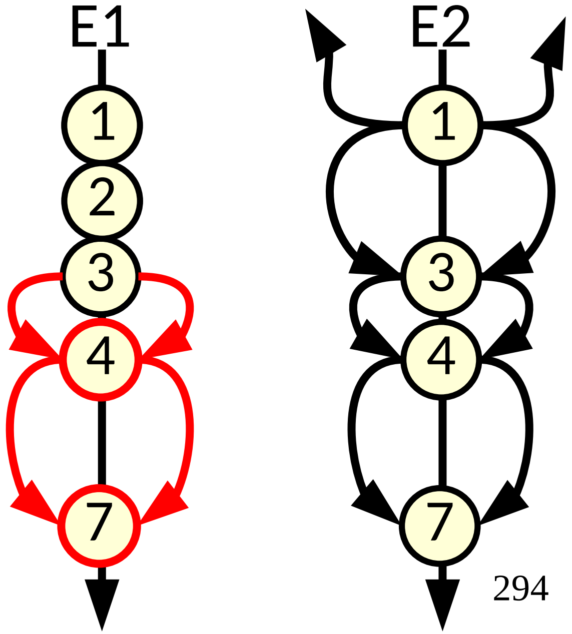
- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# *local* dependencies"?

$$argmin_{s \in sd_i} |sd| \wedge E1[sd(E2)_i] \rightarrow sd(E2)_{i+1} \wedge E2[sd(E1)_i] \rightarrow sd(E1)_{i+1}$$



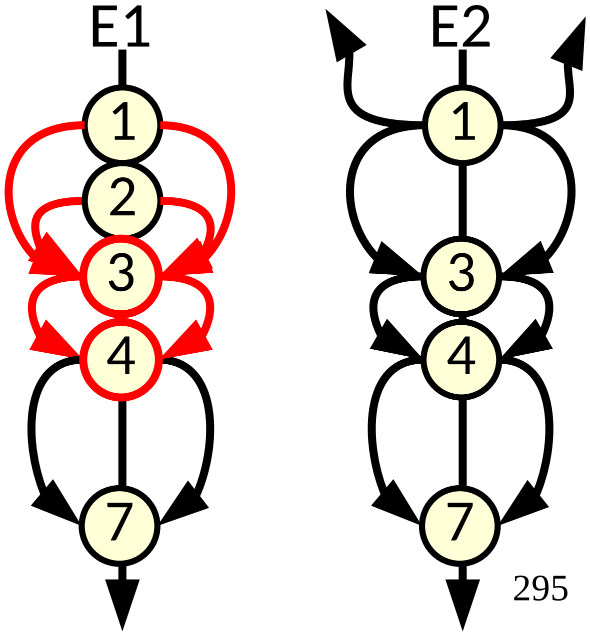
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?



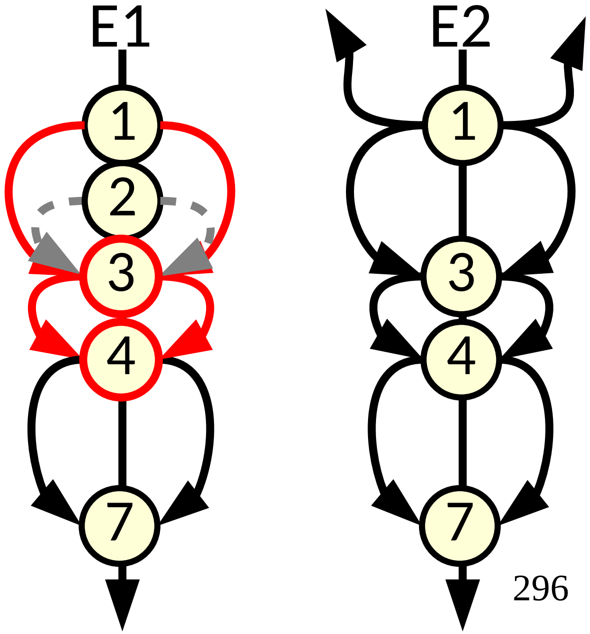
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?



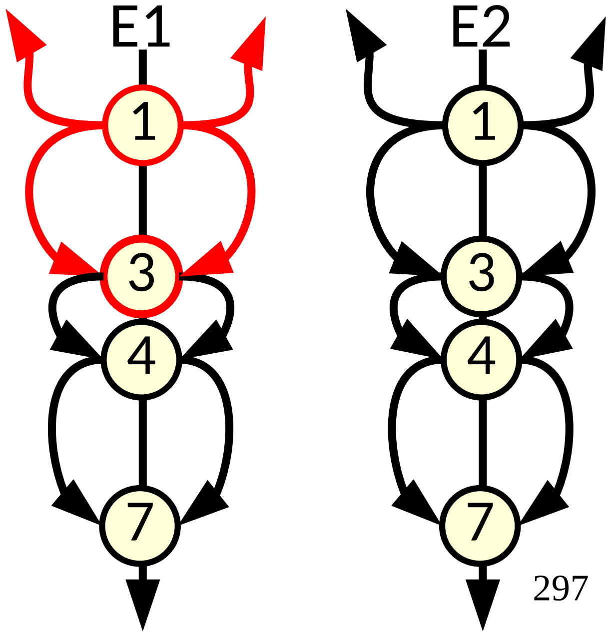
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?



Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?

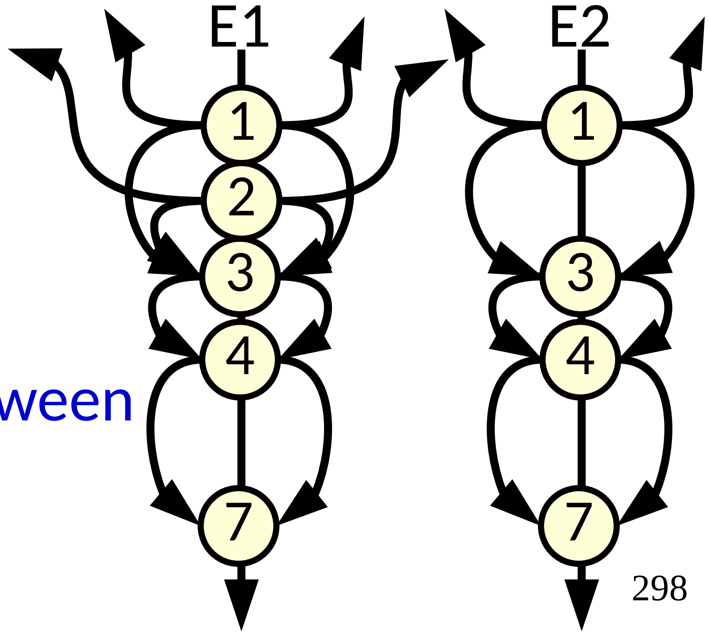


Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

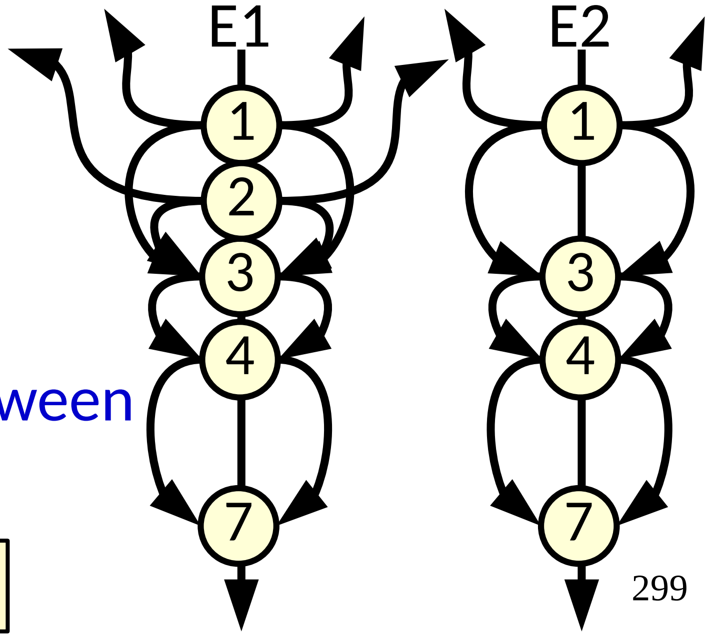
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?

- There are currently unknown trade offs between tractability, intuitiveness, and correctness



Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the "# steps"?
  - Minimize the "# dependencies"?
  - Minimize the "# local dependencies"?
- There are currently unknown trade offs between tractability, intuitiveness, and correctness

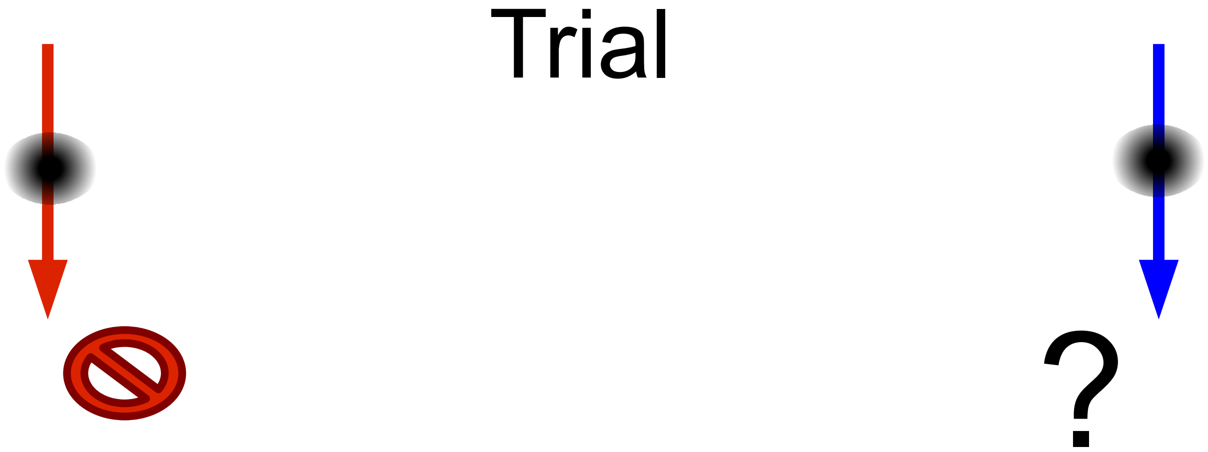


Even local blame is actually challenging

Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions…
  - We can answer these causality questions by running experiments!

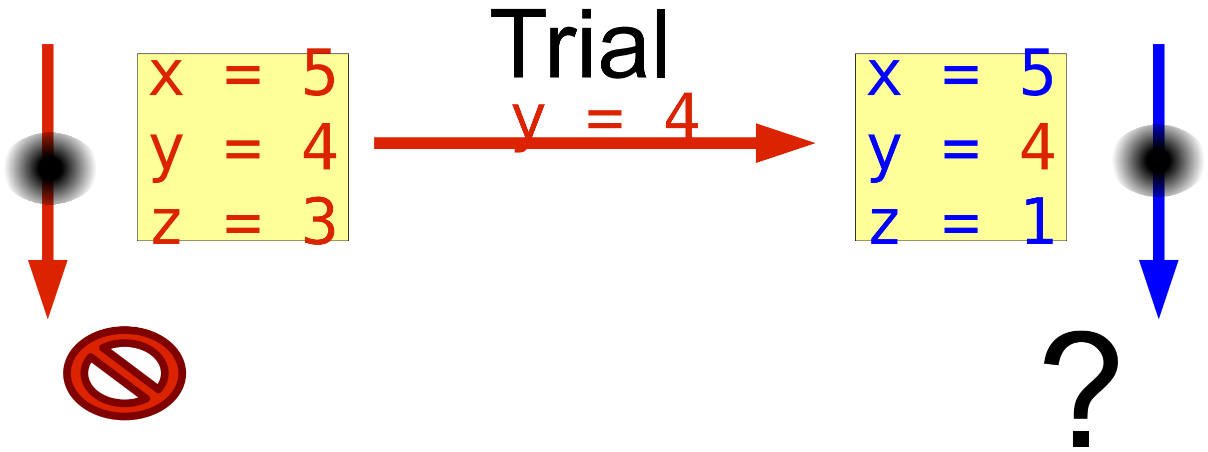
What Should We Blame?



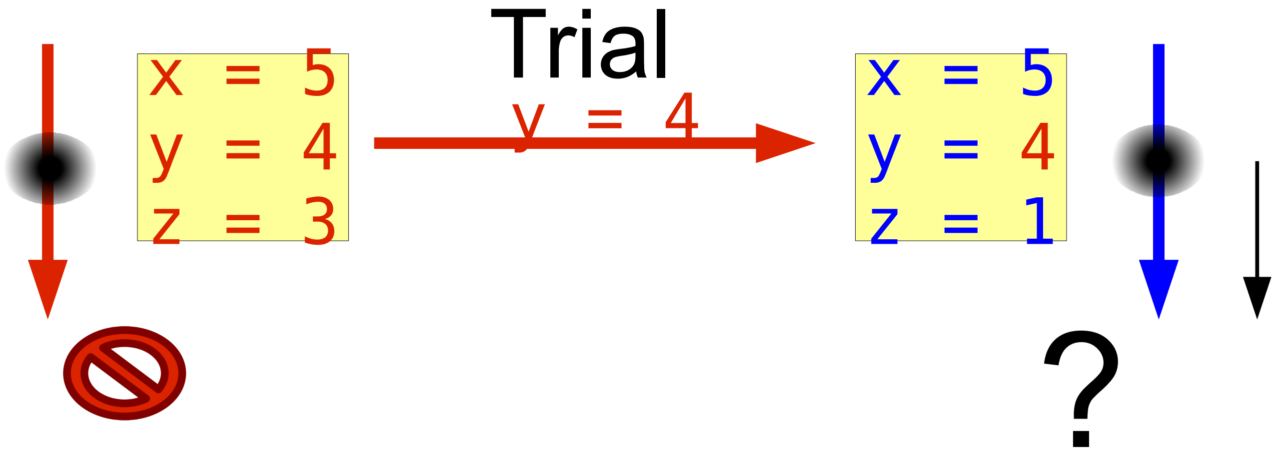
What Should We Blame?

Trial

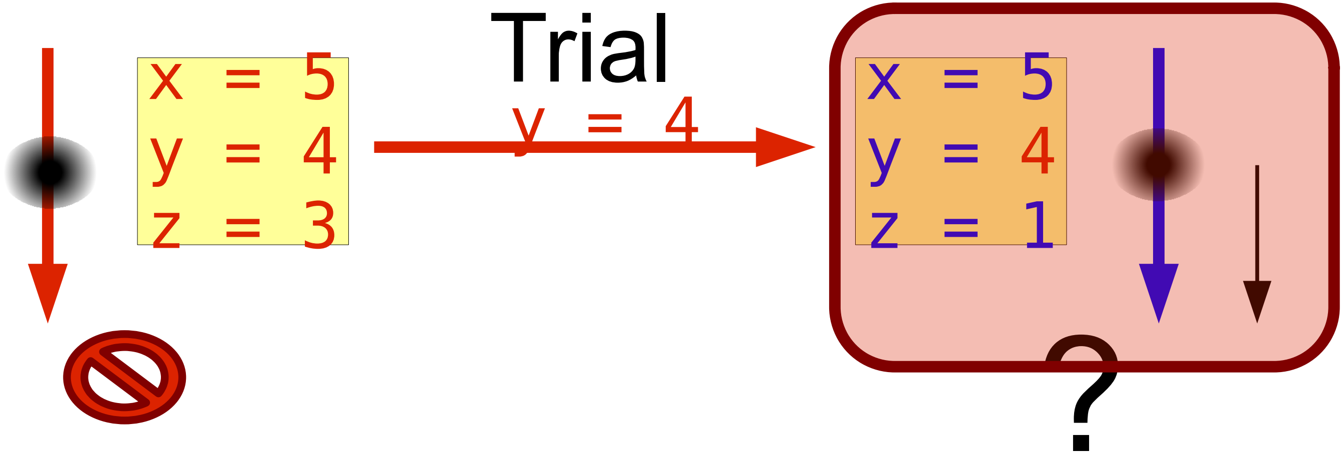
What Should We Blame?



What Should We Blame?



What Should We Blame?



What does this patched run even mean?

Example – Altered Meaning

```
x ← input()
y ← input()
z ← input()
if y+z > 10:
  y ← 5
else: y ← y+1
print(y)
```

Buggy

```
x ← 0
y ← 7
z ← 3
if False:

else: y ← 8
print(8)
```

Correct

```
x ← 1
y ← 3
z ← 6
if False:

else: y ← 4
print(4)
```

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy

```
x ← 0
y ← 7
z ← 3
if False:

else: y ← 8
print(8)
```

Correct

```
x ← 1
y ← 3
z ← 6
if False:

else: y ← 4
print(4)
```

What should we blame here?

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy

```
x ← 0
y ← 7
z ← 3
if False:

else: y ← 8
print(8)
```

Trial

Correct

```
x ← 1
y ← 3
z ← 6
if False:

else: y ← 4
print(4)
```

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy	Trial	Correct
x ← 0	x ← 0	x ← 1
y ← 7	y ← 7	y ← 3
z ← 3	z ← 3	z ← 6
if False:		if False:
else: y ← 8		else: y ← 4
print(8)		print(4)

Example – Altered Meaning

```
x ← input()
y ← input()
z ← input()
if y+z > 10:
    y ← 5
else: y ← y+1
print(y)
```

Buggy	Trial	Correct
x ← 0	x ← 0	x ← 1
y ← 7	y ← 7	y ← 3
z ← 3	z ← 3	z ← 6
if False:	if False:	if False:
else: y ← 8	else: y←8	else: y ← 4
print(8)	print(8)	print(4)

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy	Trial	Correct
x ← 0		x ← 1
y ← 7	y ← 7	y ← 3
z ← 3		z ← 6
if False:		if False:
else: y ← 8		else: y ← 4
print(8)		print(4)

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy	Trial	Correct
x ← 0	x ← 1	x ← 1
y ← 7	y ← 7	y ← 3
z ← 3	z ← 6	z ← 6
if False:		if False:
else: y ← 8		else: y ← 4
print(8)		print(4)

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy	Trial	Correct
x ← 0	x ← 1	x ← 1
y ← 7	y ← 7	y ← 3
z ← 3	z ← 6	z ← 6
if False:	if True:	if False:
	y ← 5	
else: y ← 8		else: y ← 4
print(8)	print(5)	print(4)

Example – Altered Meaning

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy	Trial	Correct
x ← 0	x ← 1	x ← 1
y ← 7	y ← 7	y ← 3
z ← 3	z ← 6	z ← 6
if False:	if True:	if False:
	y ← 5	
else: y ← 8		else: y ← 4
print(8)	print(5)	print(4)

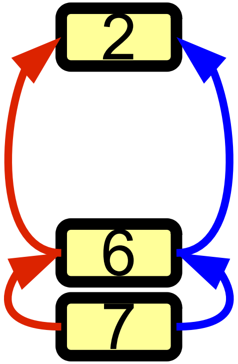
- New control flow unlike original runs
- Occurs in large portion of real bugs

Dual Slicing

```
1)x ← input()
2)y ← input()
3)z ← input()
4)if y+z > 10:
5)   y ← 5
6)else: y ← y+1
7)print(y)
```

Buggy

```
x ← 0
y ← 7
z ← 3
if False:

else: y ← 8
print(8)
```



Correct

```
x ← 1
y ← 3
z ← 6
if False:

else: y ← 4
print(4)
```

Extract

```
2)y ← input()
6)y ← y+1
7)print(y)
```

Example – Extracted Meaning

```
2)y ← input()
6)y ← y+1
7)print(y)
```

Buggy	Trial	Correct
y ← 7 y ← 8 print(8)	⟺	y ← 3 y ← 4 print(4)

Example – Extracted Meaning

```
2)y ← input()
6)y ← y+1
7)print(y)
```

Buggy	Trial	Correct
y ← 7	y ← 7	y ← 7
y ← 8		y ← 4
print(8)		print(4)

Example – Extracted Meaning

```
2)y ← input()
6)y ← y+1
7)print(y)
```

Buggy	Trial	Correct
y ← 7	y ← 7	y ← 7
y ← 8	y ← 8	y ← 4
print(8)	print(8)	print(4)

Trial can now correctly blame y

Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions…
  - We can answer these causality questions by running experiments!
- We perform these causality tests in both directions in order to collect symmetric information

Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions…
  - We can answer these causality questions by running experiments!
- We perform these causality tests in both directions in order to collect symmetric information
  - How did the buggy run behave differently than the correct one?
  - How did the correct run behave differently than the buggy one?
  - These questions do not have the same answer!

Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions…
  - We can answer these causality questions by running experiments!
- We perform these causality tests in both directions in order to collect symmetric information
  - How did the buggy run behave differently than the correct one?
  - How did the correct run behave differently than the buggy one?
  - These questions do not have the same answer!
- In practice, there are additional issues,
  and even defining causation in this context needs further research.

Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions…
  - We can answer these causality questions by running experiments!
- We perform these causality tests in both directions in order to collect symmetric information
  - How did the buggy run behave differently than the correct one?
  - How did the correct run behave differently than the buggy one?
  - These questions do not have the same answer!
- In practice, there are additional issues,
  and even defining causation in this context needs further research.
  - Did we want to blame only y in the example?
  - Pruning blame on y is necessary in many real cases, can they be refined?
  - Can it be done without execution? With a stronger statistical basis?

Summing Up

Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?

Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?
- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, …

Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?
- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, …
- Selecting which executions to analyze
  - Existing test suite? Always on runtime? Directed test generation?
  - How does *underapproximation* affect your conclusions?
  - Can you still achieve your objective in spite of it?

Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?
- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, …
- Selecting which executions to analyze
  - Existing test suite? Always on runtime? Directed test generation?
  - How does *underapproximation* affect your conclusions?
  - Can you still achieve your objective in spite of it?
- Doing some of the work ahead of time
  - What can you precompute to improve all of the above?

Summary

- Analyze the actual/observed behaviors of a program

Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program’s behavior to collect information

Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program's behavior to collect information
- Analyze the information online or offline

Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program's behavior to collect information
- Analyze the information online or offline
- The precise configuration must be tailored to the objectives & insights
  - Compiled vs DBI
  - Online vs Postmortem
  - Compressed, Encoded, Samples, …
  - …