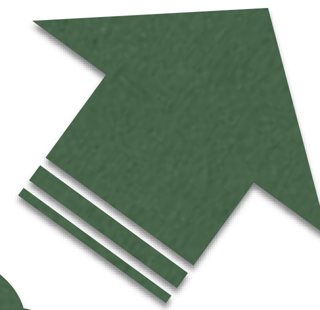
Taming Undefined Behavior in LLVM

SIGPL 2017 Summer

Juneyoung Lee
Software Foundations Lab (Advisor: Chung-Kil Hur)

Undefined Behavior (UB)?

UB in LLVM IR

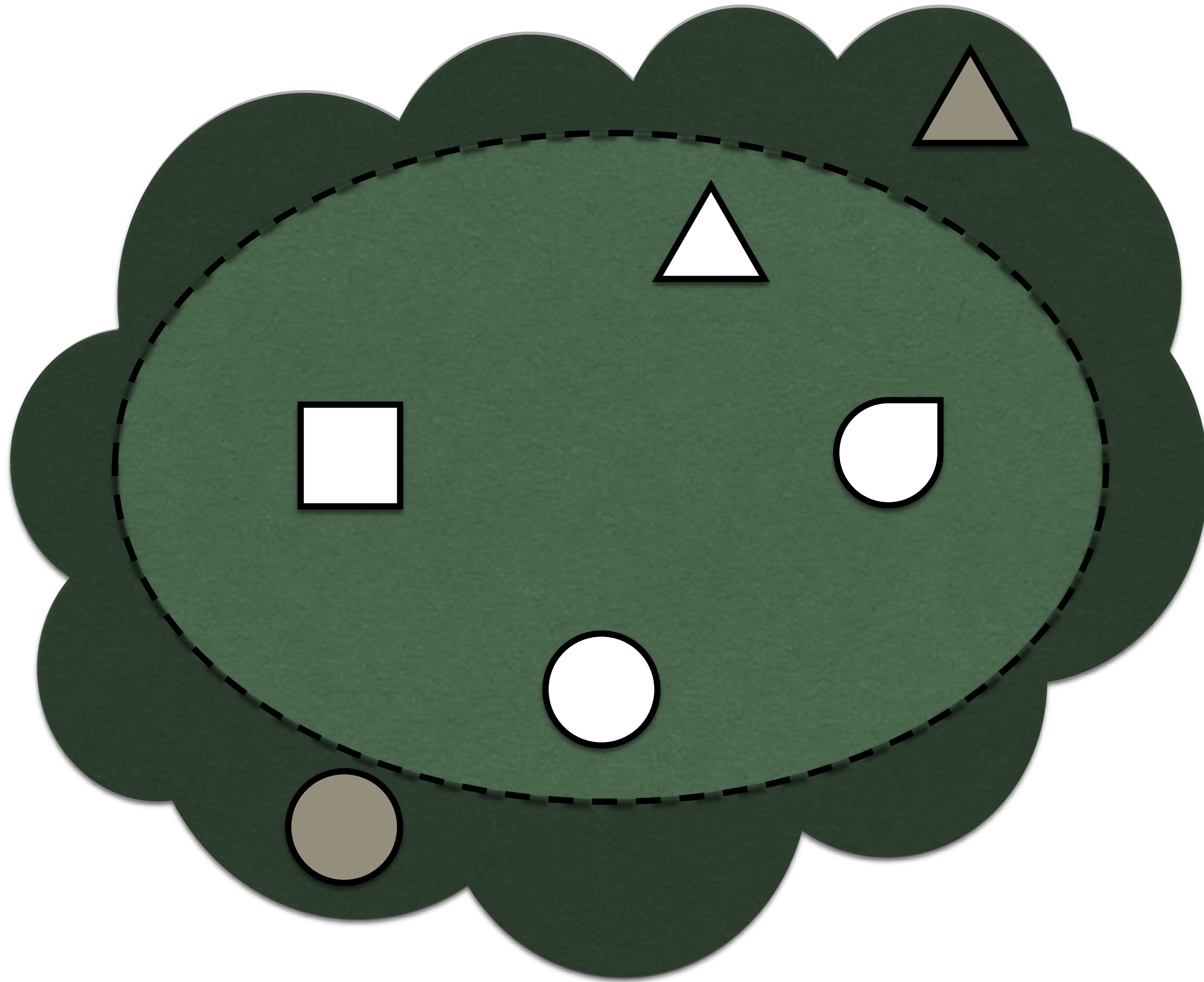


Problem of UB in IR & Solution

What is Undefined Behavior?

Undefined
Behavior?

Undefined Behavior?



Undefined
Behavior?

ISO/IEC 9899:2011
Programming languages – C

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

ISO/IEC 9899:2011
Programming languages – C

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

```
int a[4];
*(a + 4) = 123;
```

ISO/IEC 9899:2011
Programming languages – C

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

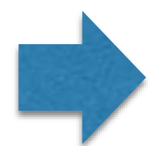
— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

```
int a[4];
*(a + 4) = 123;
```

Undefined Behavior!

C ≠ Assembly

- C abstract machine!



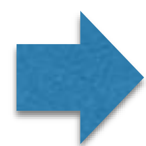
```
int a[4];
*(a + 4) = 123;
```

Memory

-

$C \neq$ Assembly

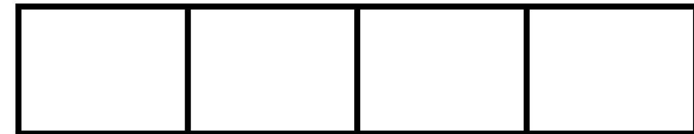
- C abstract machine!



```
int a[4];
*(a + 4) = 123;
```

Memory

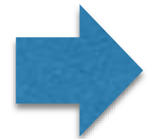
l



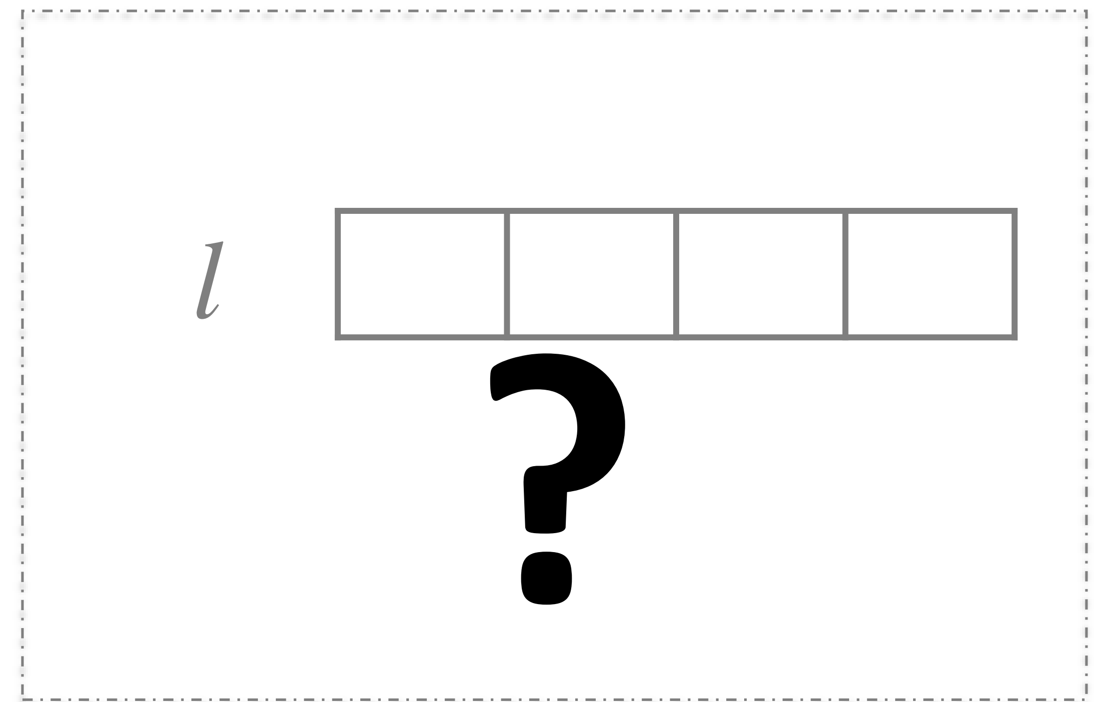
$a \rightarrow (l, 0)$

$C \neq$ Assembly

- C abstract machine!

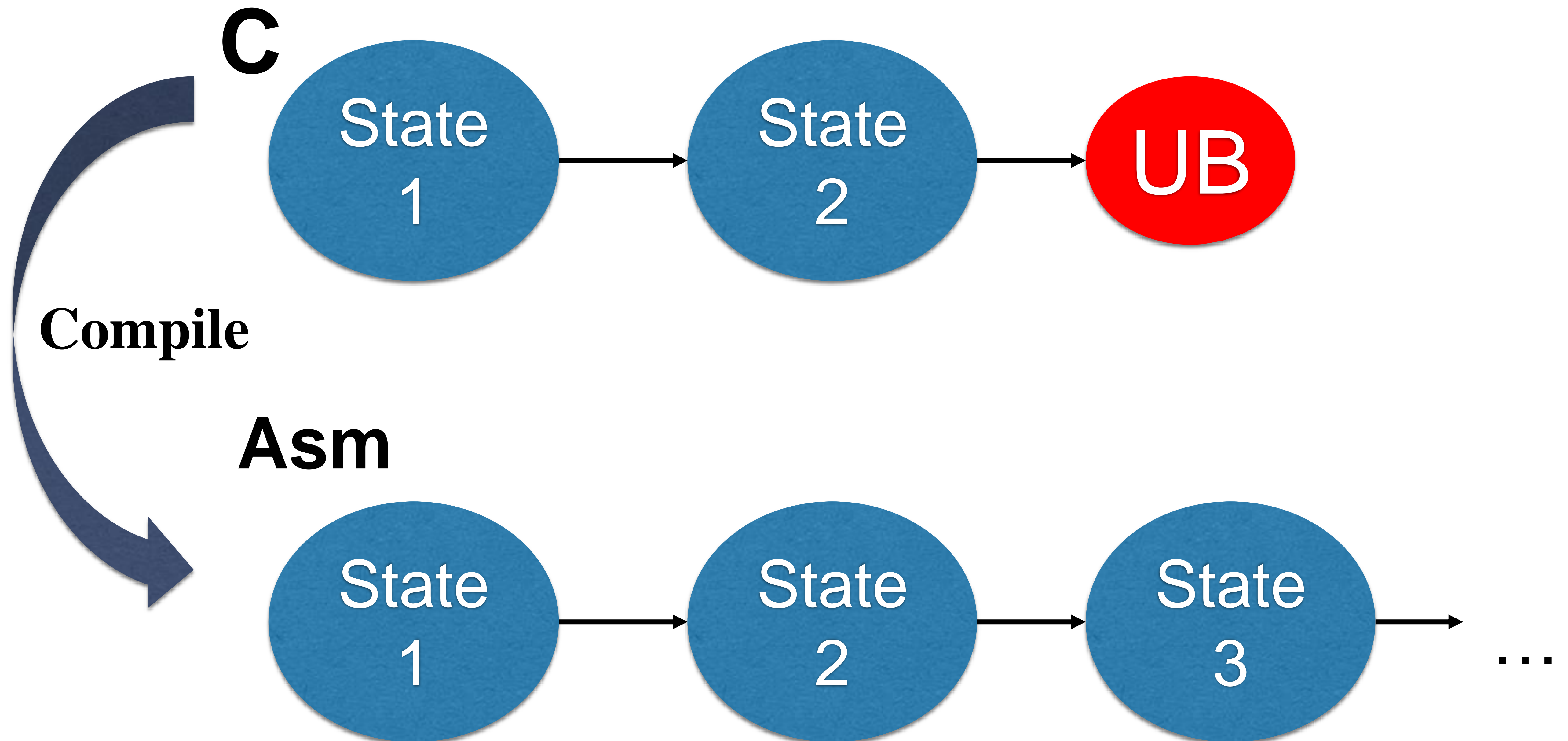


```
int a[4];
*(a + 4) = 123;
```

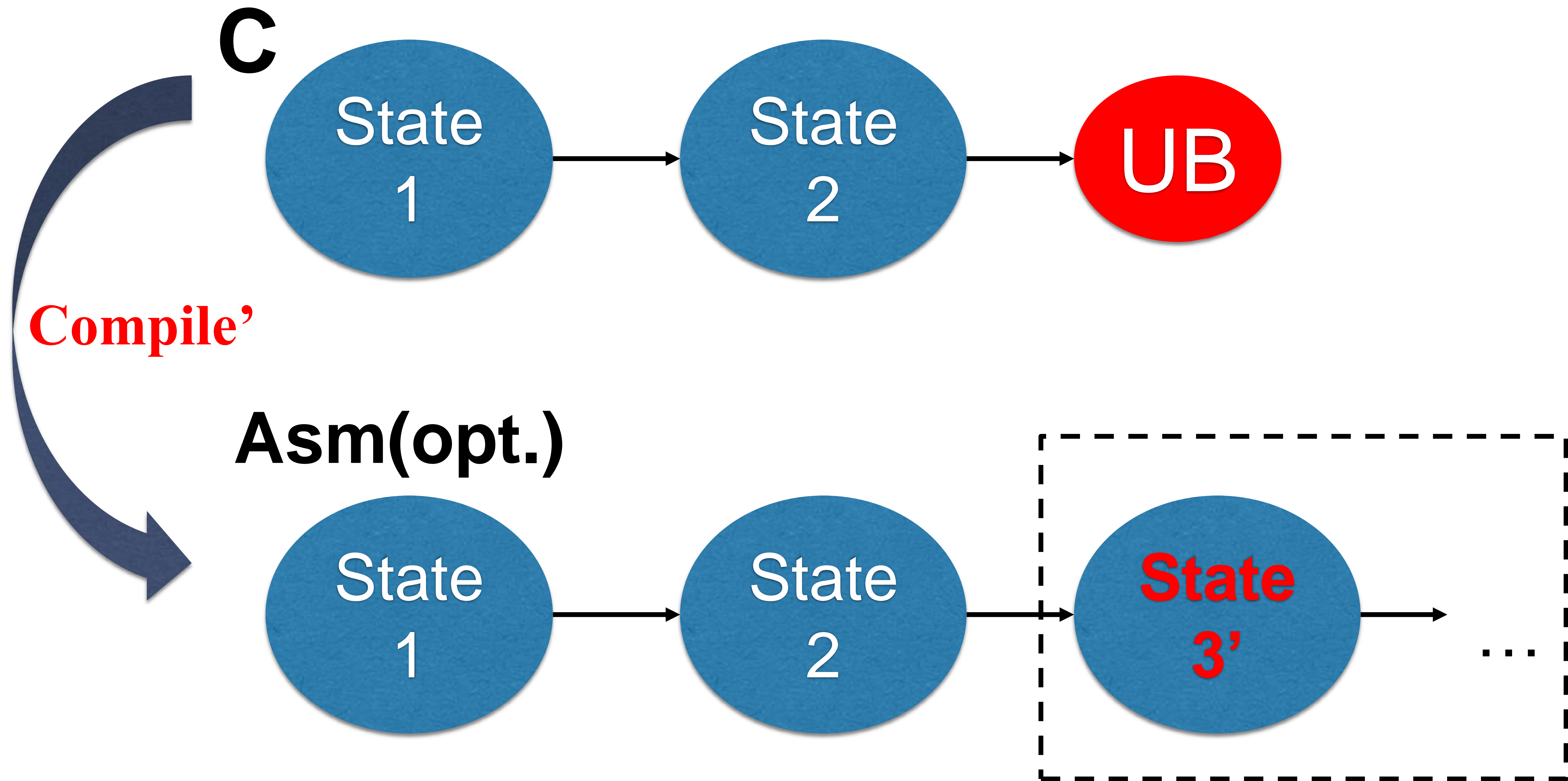


$a \rightarrow (l, 0)$

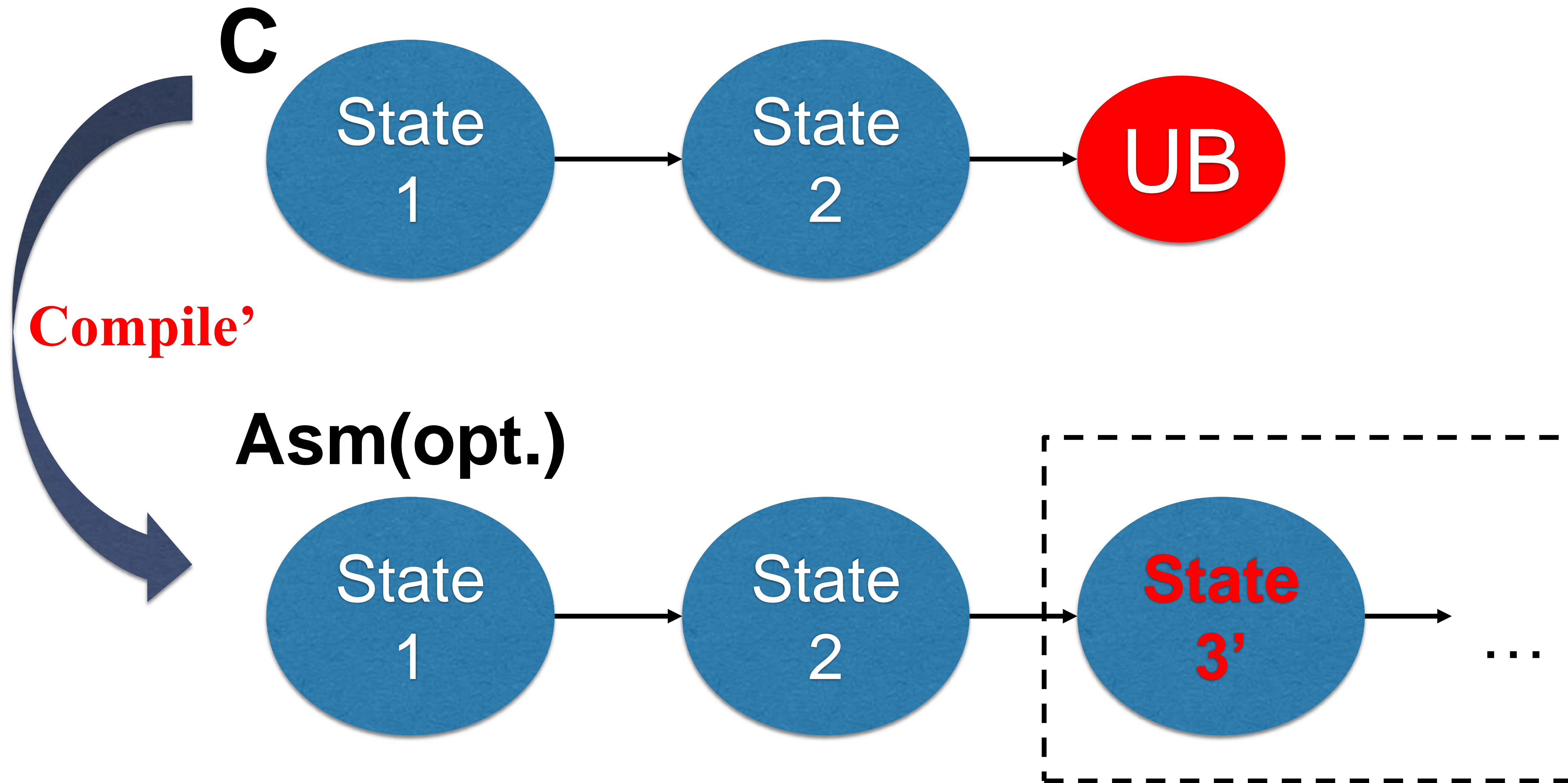
UB & Compiler



UB & Compiler



UB & Compiler



Optimizer can change UB into anything!

Eliminating Redundant Load

```
int a[4];
int b[4];
int i;
```

C

```
a[0] = 10;
b[i] = 20;
output(a[0]);
```

Asm

```
a[0] = 10;
b[i] = 20;
output(10);
```

Eliminating Redundant Load

```
int a[4];
int b[4];
int i;
```

3

C

```
a[0] = 10;
b[i] = 20;
output(a[0]);
```

Asm

```
a[0] = 10;
b[i] = 20;
output(10);
```

Eliminating Redundant Load

```
int a[4];
int b[4];
int i;
```

← 4

C

```
a[0] = 10;
b[i] = 20;
output(a[0]);
```

Asm

```
a[0] = 10;
b[i] = 20;
output(10);
```

Eliminating Redundant Load

```
int a[4];
int b[4];
int i;
```

4

UB

C

```
a[0] = 10;
b[i] = 20;
output(a[0]);
```

Asm

```
a[0] = 10;
b[i] = 20;
output(10);
```

UB and Optimization

Register Promotion

```
int a;
int b[4];
int i;
```

C

```
a    = t;
b[i] = 20;
output(a);
```

Asm

```
eax  = t;
b[i] = 20;
output(eax);
```

Register Promotion

```
int a;
int b[4];
int i;
```

4

C

```
a    = t;
b[i] = 20;
output(a);
```

Asm

```
eax  = t;
b[i] = 20;
output(eax);
```

Register Promotion

```
int a;
int b[4];
int i;
```

4

UB

C

```
a    = t;
b[i] = 20;
output(a);
```

Asm

```
eax  = t;
b[i] = 20;
output(eax);
```

Undefined Behavior in C

- Many operations are defined to produce UB
  - To support optimizations
  - **> 200 cases in total**!

- Let me introduce two famous cases:

1. Pointer overflow
2. Signed integer overflow

UB and Optimization

Pointer Overflow

J.2 Undefined behavior

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

UB and Optimization

Pointer Overflow

J.2 Undefined behavior

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

```
int a[4];
.. a + 4 ..
```

OK

```
int a[4];
.. a + 5 ..
```

UB

UB and Optimization

Pointer Overflow

J.2 Undefined behavior

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

```
int a[4];
.. a + 4 ..
```

OK

```
int a[4];
.. a + 5 ..
```

UB

This guarantees that "p + i" never overflows 2^32!

Pointer Overflow

```
int* p;
int  a;
int  b;
```

C

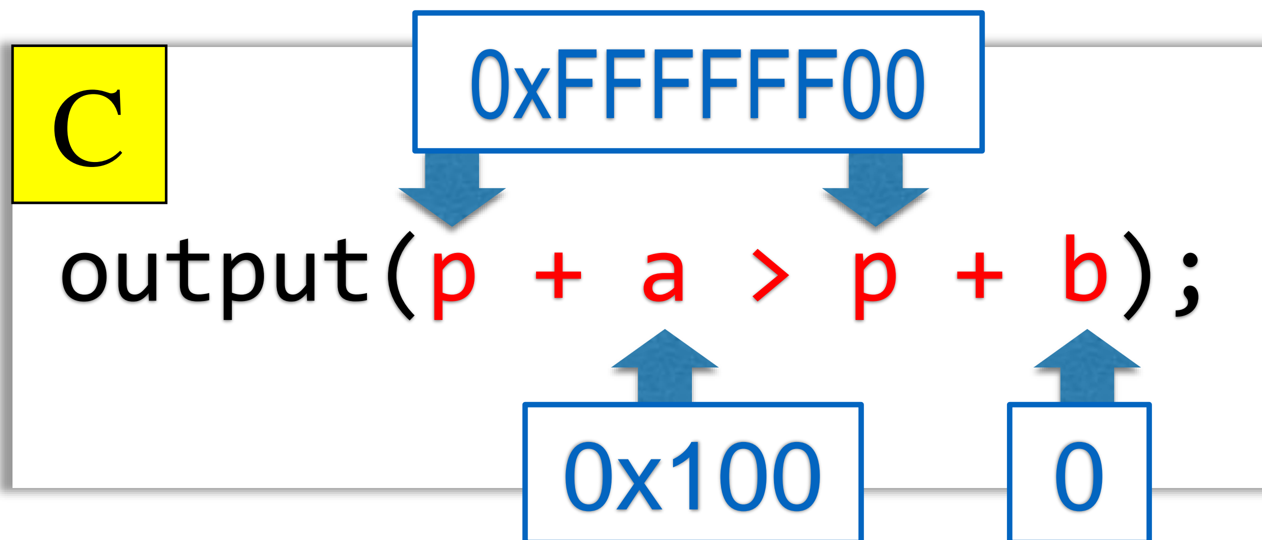
```
output(p + a > p + b);
```

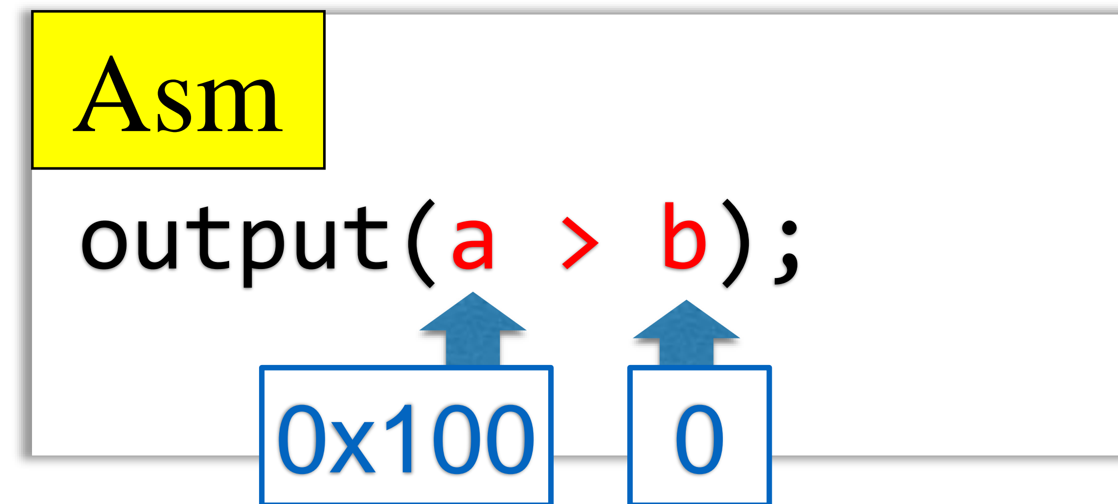
Asm

```
output(a > b);
```

Pointer Overflow

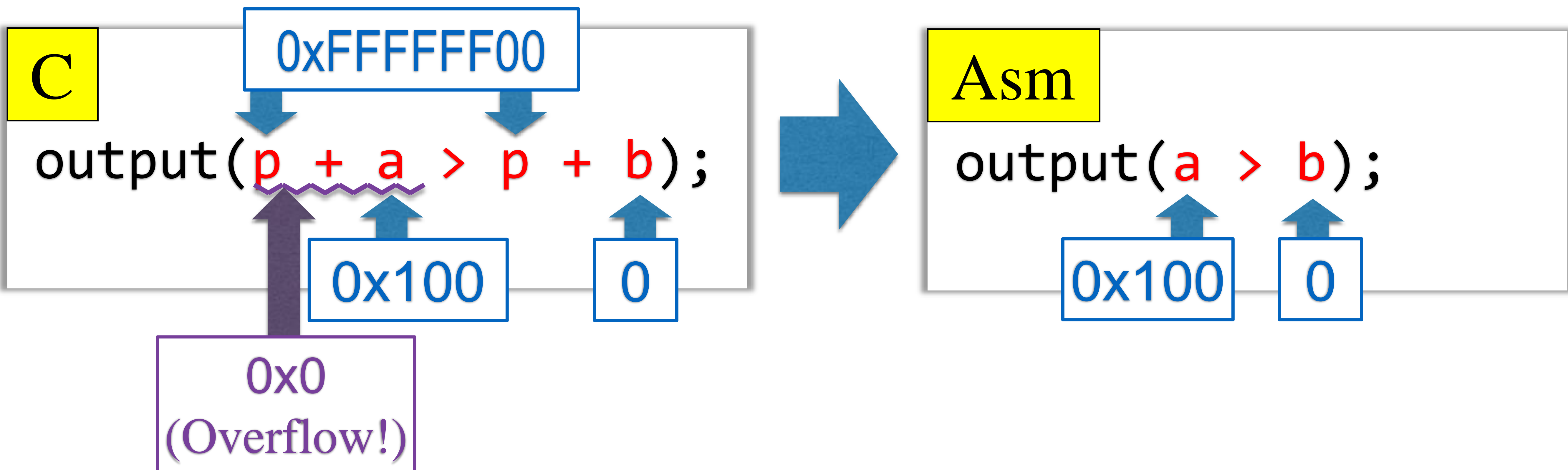
```
int* p;
int  a;
int  b;
```





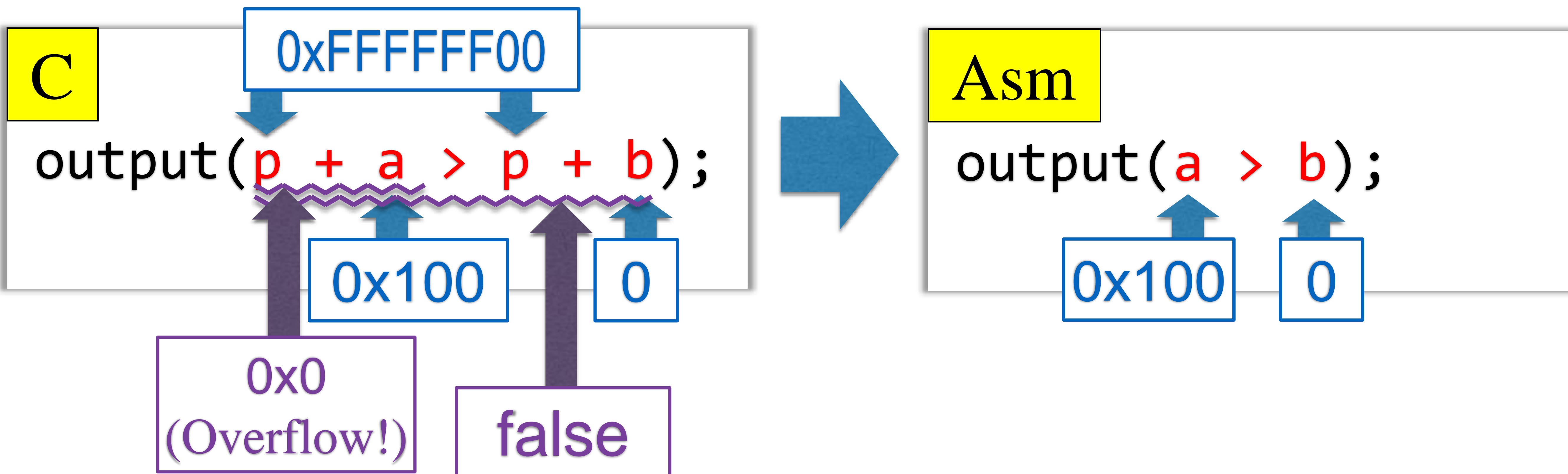
Pointer Overflow

```
int* p;
int  a;
int  b;
```



Pointer Overflow

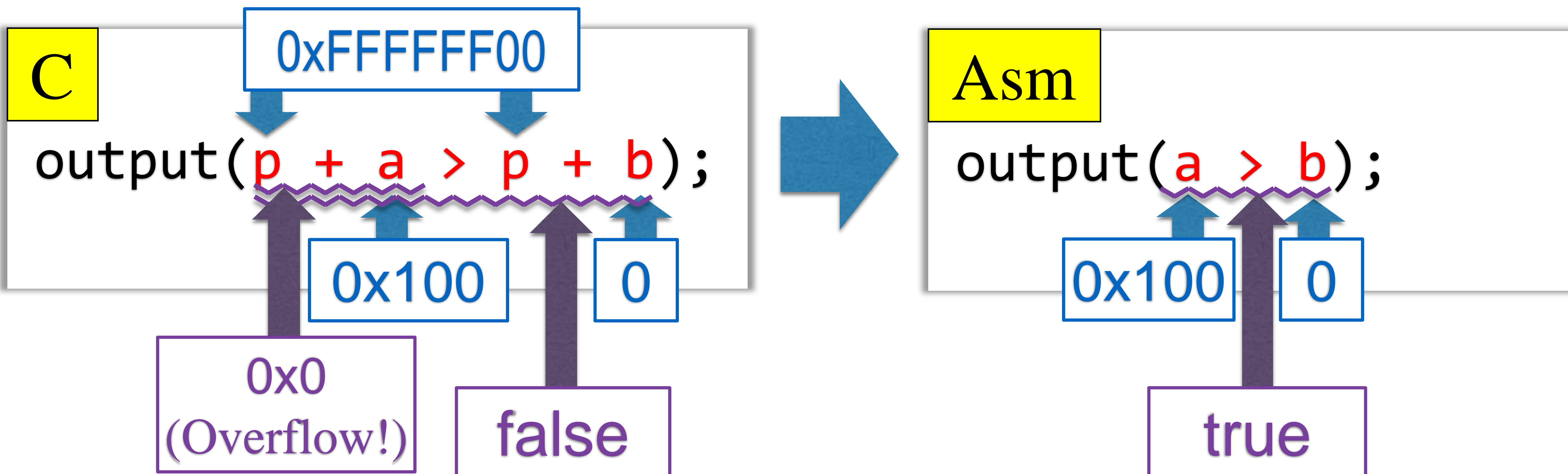
```
int* p;
int  a;
int  b;
```



UB and Optimization

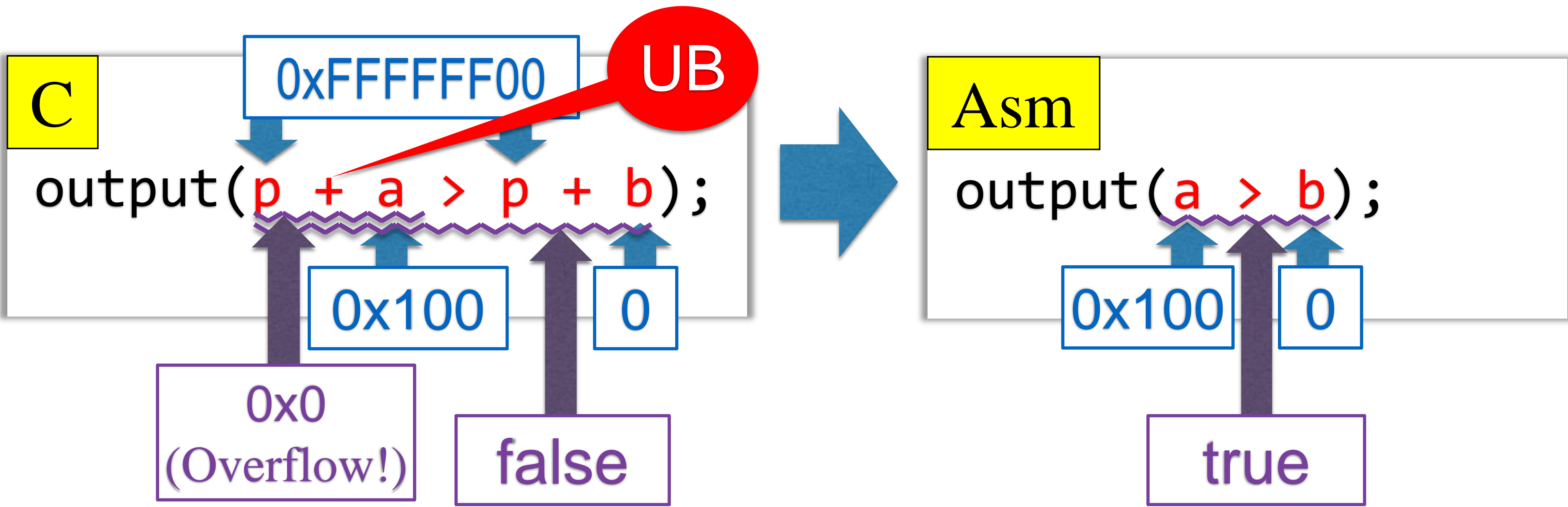
Pointer Overflow

```
int* p;
int  a;
int  b;
```



UB and Optimization
Pointer Overflow

```
int* p;
int  a;
int  b;
```



UB and Optimization
Signed Integer Overflow

6.5 Expressions

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

UB and Optimization

Signed Integer Overflow

6.5 Expressions

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

```
int t = 2147..47
.. t + 1 ..
```

UB

```
int t = -2147..48
.. (-t) ..
```

UB

Signed Integer Overflow

> **6.5 Expressions**
>
> 5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

```
int t = 2147..47
.. t + 1 ..
```

UB

```
int t = -2147..48
.. (-t) ..
```

UB

1. It is known that SIO is ‘dangerous’.
2. It gives much more optimization opportunities!

Signed Integer Overflow is Dangerous!

- It's on "CWE/SANs Top 25 Software Errors"[1].

> The 2011 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical errors that can lead to serious vulnerabilities in software.

- Sanitize your application with..
  - Runtime Test: IOC[2] / Static Analysis Tool: cppcheck
- Use `unsigned int`

[1] http://cwe.mitre.org/top25/

[2] Will, Peng, John, and Vikram Adve., Understanding Integer Overflow in C/C++, ICSE'12

Signed Integer Overflow

```
int* p;
```

C

```
int i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Asm

```
int64 i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Signed Integer Overflow

```
int* p;
```

C

INT32_MAX

```
int i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Asm

INT32_MAX

```
int64 i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Signed Integer Overflow

```
int* p;
```

C

INT32_MAX

INT32_MIN

```
int i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Asm

INT32_MAX

```
int64 i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Signed Integer Overflow

```
int* p;
```

C

INT32_MAX

INT32_MIN

```
int i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

Asm

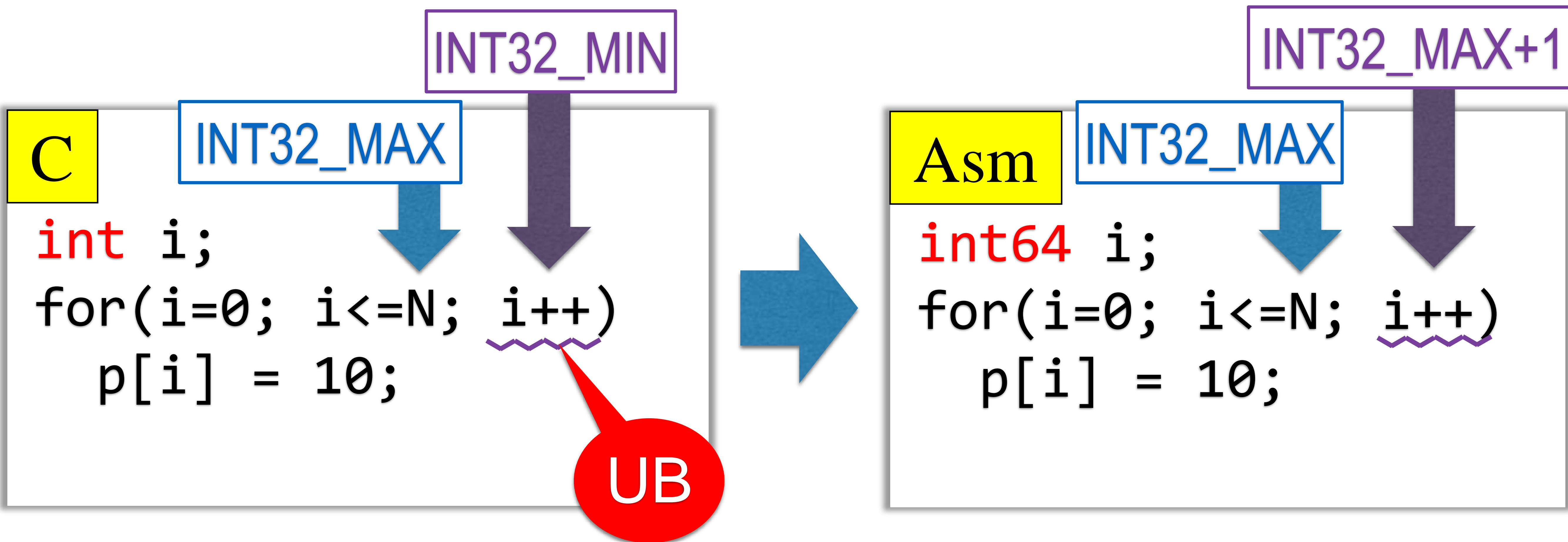
INT32_MAX

INT32_MAX+1

```
int64 i;
for(i=0; i<=N; i++)
  p[i] = 10;
```

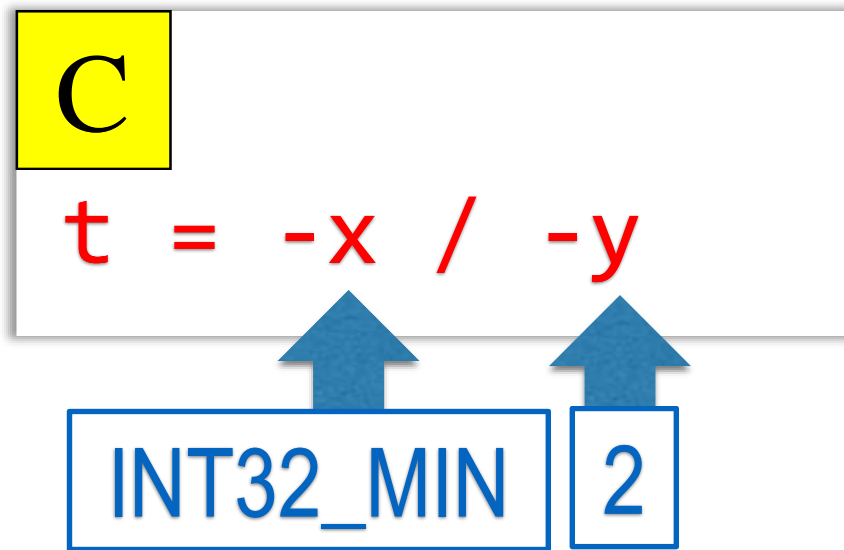
Signed Integer Overflow

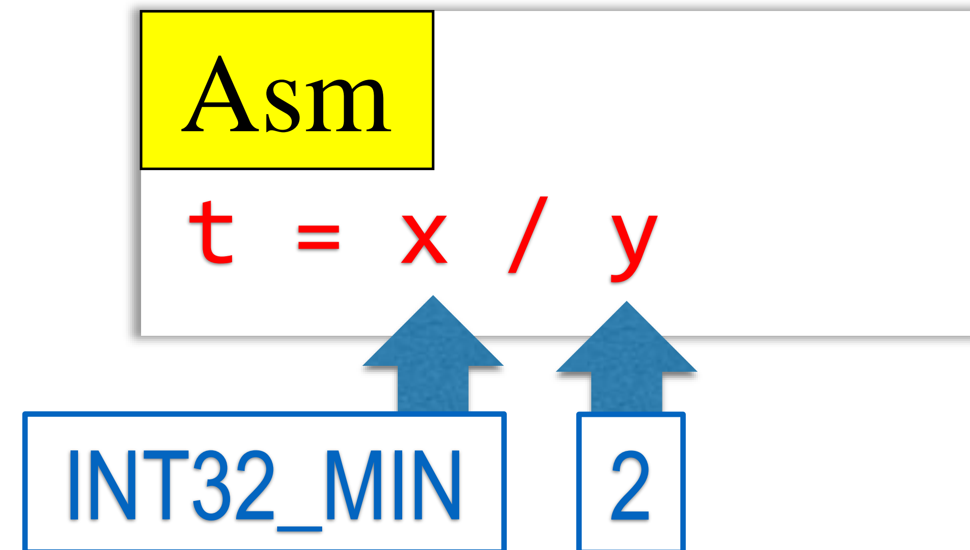
```
int* p;
```



Signed Integer Overflow (cont.)

Signed Integer Overflow (cont.)





Signed Integer Overflow (cont.)

C

```
t = -x / -y
```

INT32_MIN 2

Asm

```
t = x / y
```

INT32_MIN 2

C

```
t = (x*c) / c'
```

-1 INT32_MIN

Asm

```
t = x * (c/c')
```

-1 INT32_MIN

UB and Optimization

Signed Integer Overflow (cont.)

C

```
t = -x / -y
```

INT32_MIN 2

Asm

```
t = x / y
```

INT32_MIN 2

C

```
t = (x*c) / c'
```

-1 INT32_MIN

Asm

```
t = x * (c/c')
```

-1 INT32_MIN

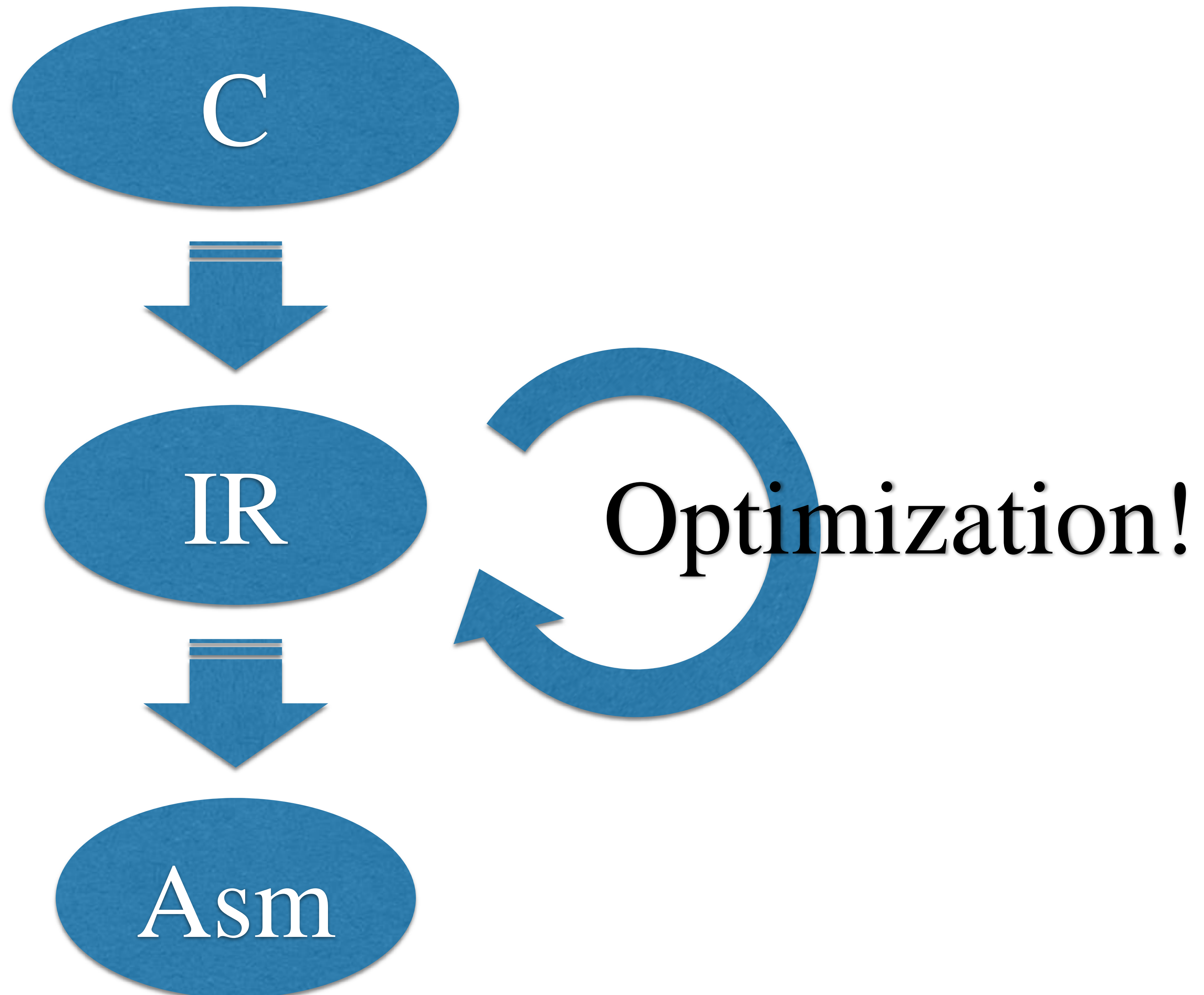
.. And many other arithmetic optimizations

Summary

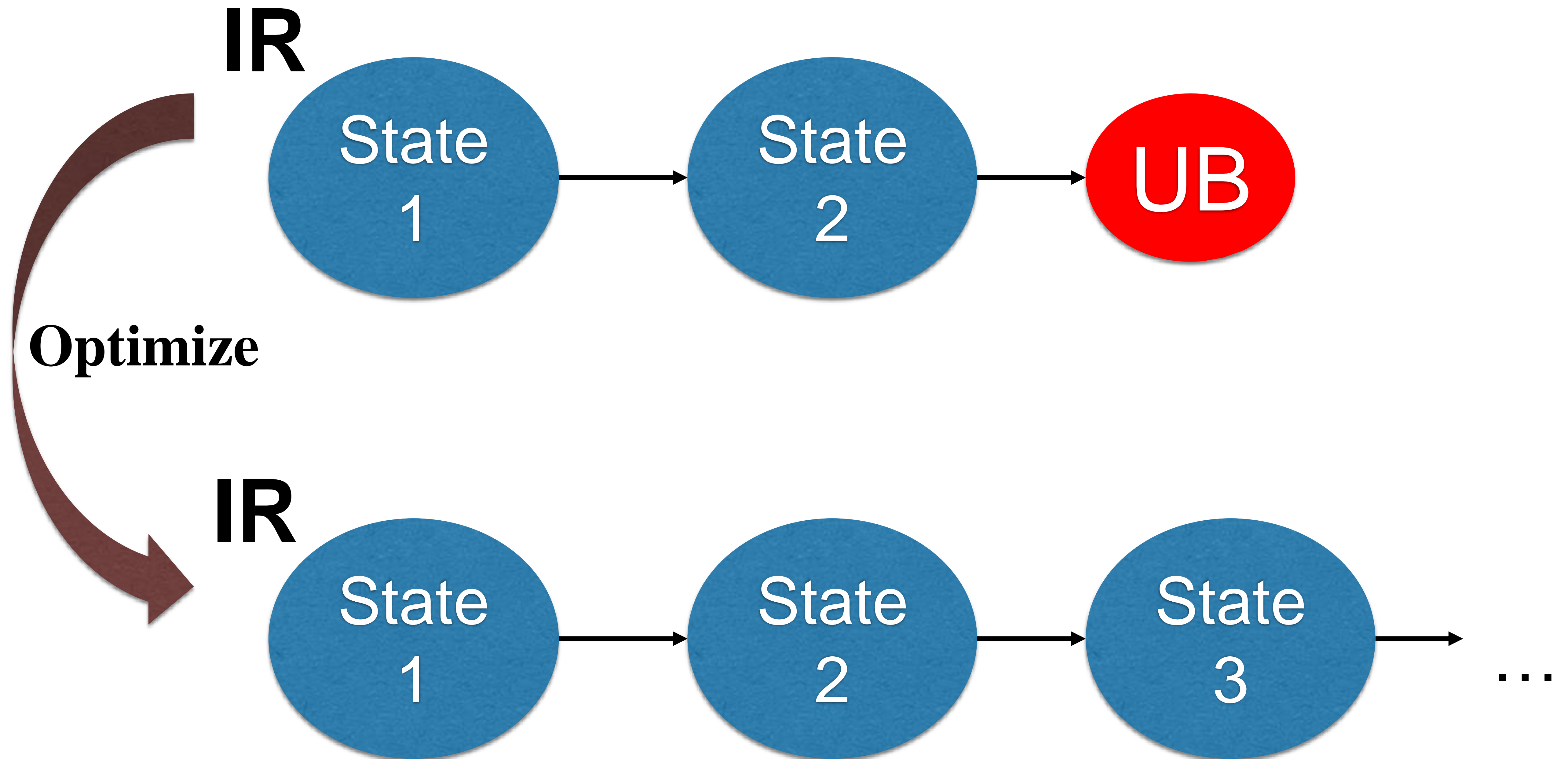
- UB is the result of erroneous operation.
- A well-written program should not have UB.
- UB helps compiler to do more optimization.

Undefined Behavior in LLVM IR

IR in Compiler

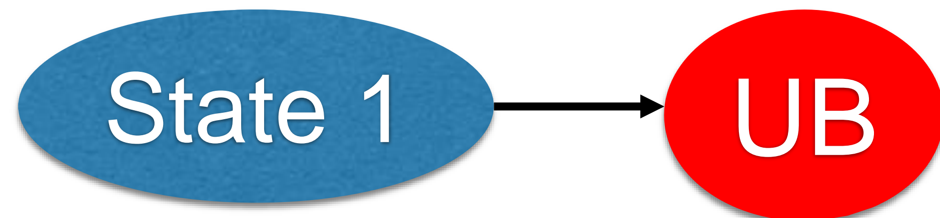


UB & Optimization



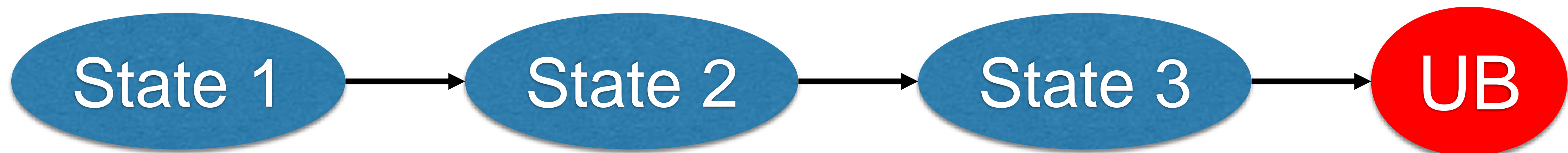
UB & Optimization

C

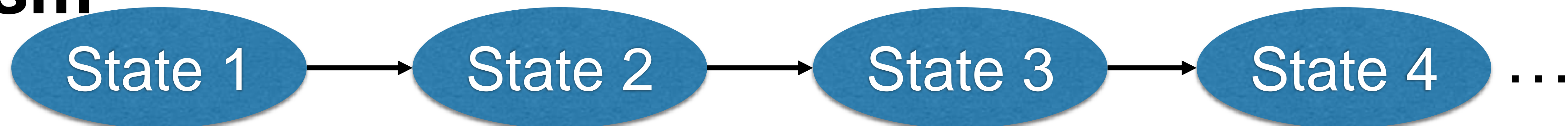


IR

IR



Asm



…

Undefined Behavior in LLVM IR?

Undefined Behavior in LLVM IR?

UB in C

≠

UB in IR!

UB in C ≠ UB in IR

Peephole Optimization

```
int* p
int  a
int  b
```

IR

```
output(p + a > p + b)
```

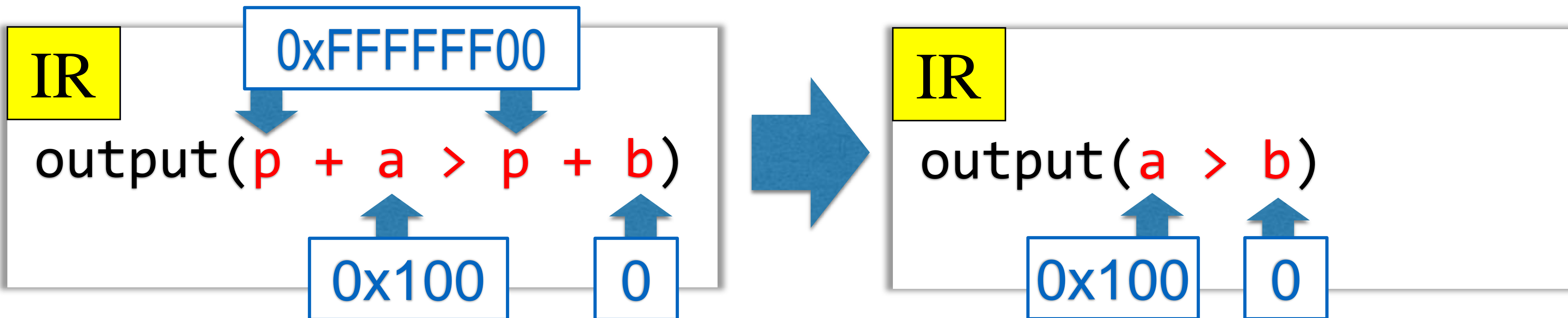
IR

```
output(a > b)
```

UB in C ≠ UB in IR

Peephole Optimization

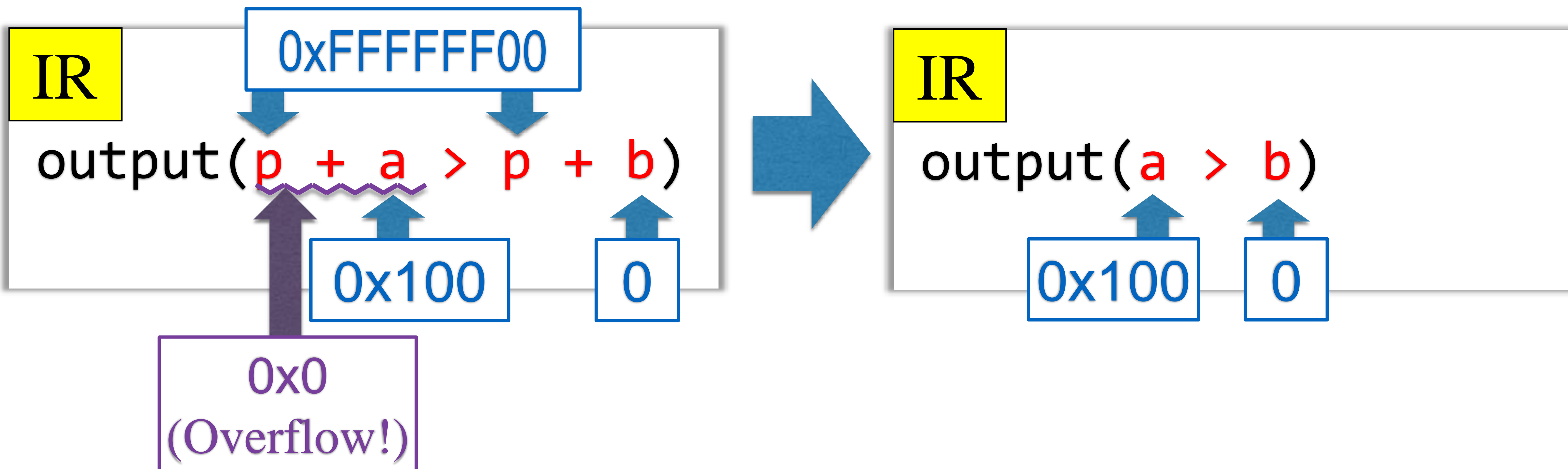
```
int* p
int  a
int  b
```



UB in C ≠ UB in IR

Peephole Optimization

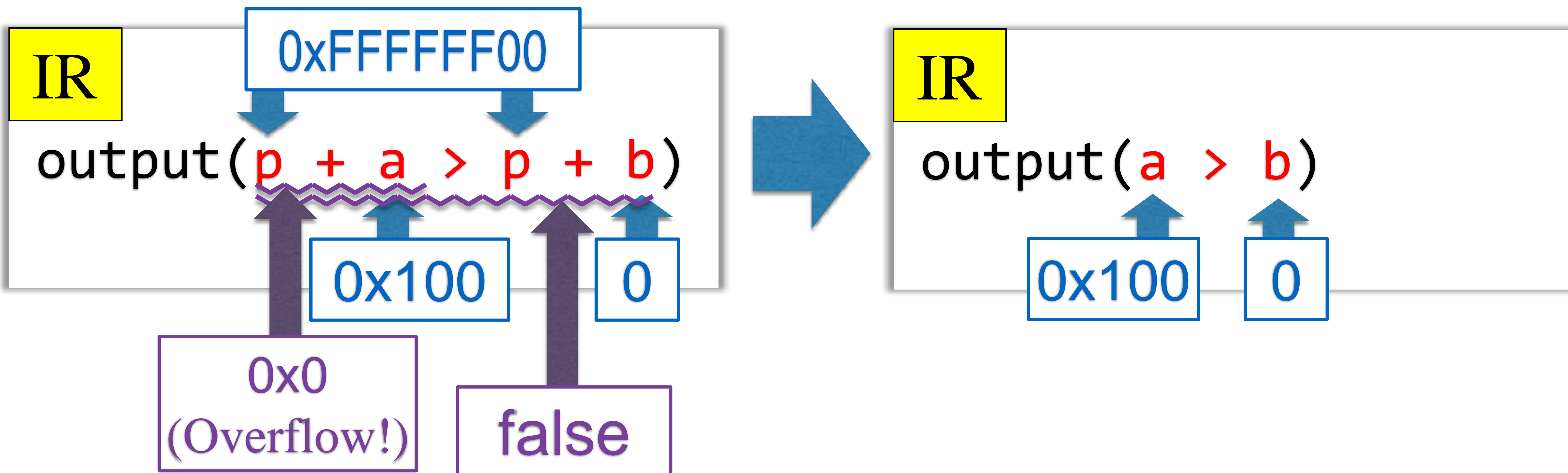
```
int* p
int  a
int  b
```



UB in C ≠ UB in IR

Peephole Optimization

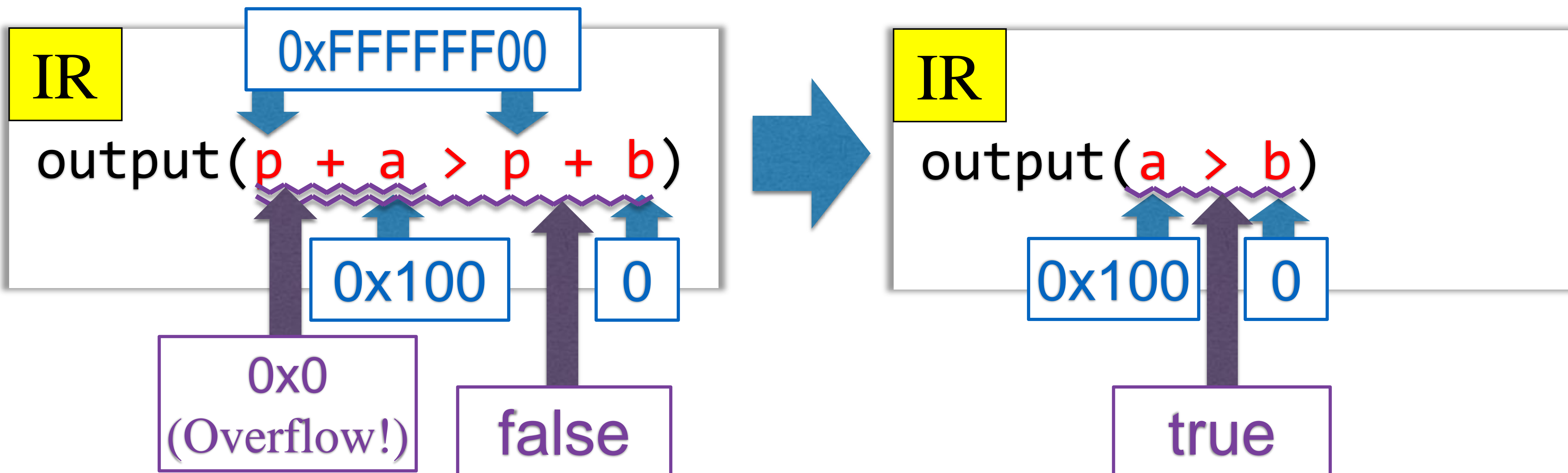
```
int* p
int  a
int  b
```



UB in C ≠ UB in IR

Peephole Optimization

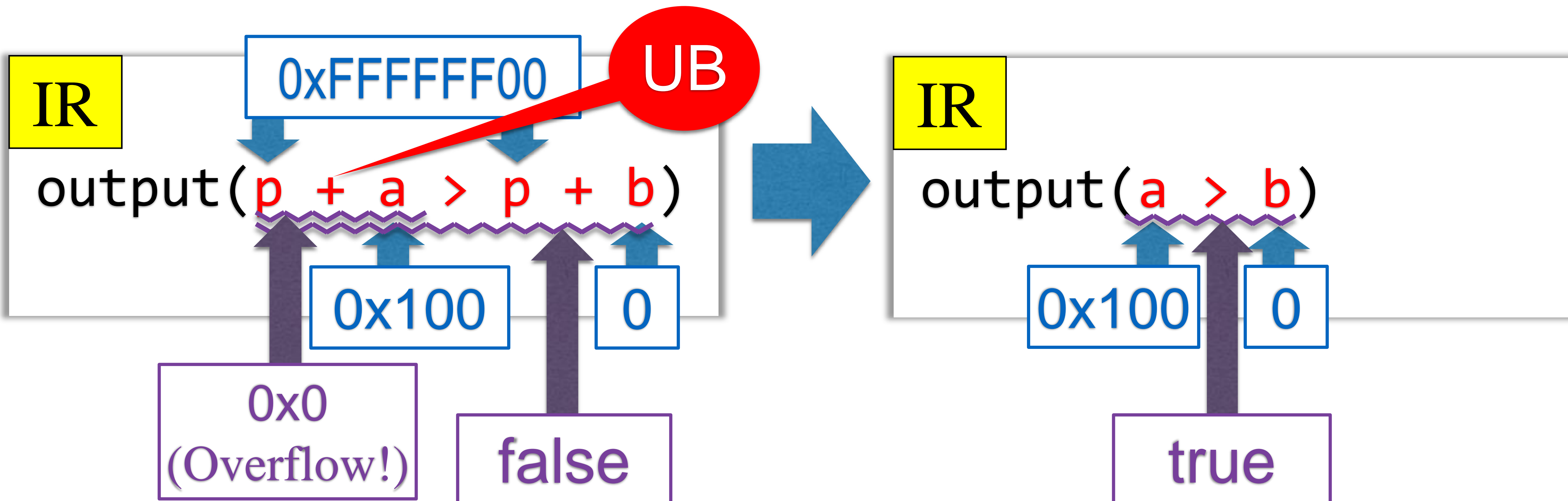
```
int* p
int  a
int  b
```



UB in C ≠ UB in IR

Peephole Optimization

```
int* p
int  a
int  b
```



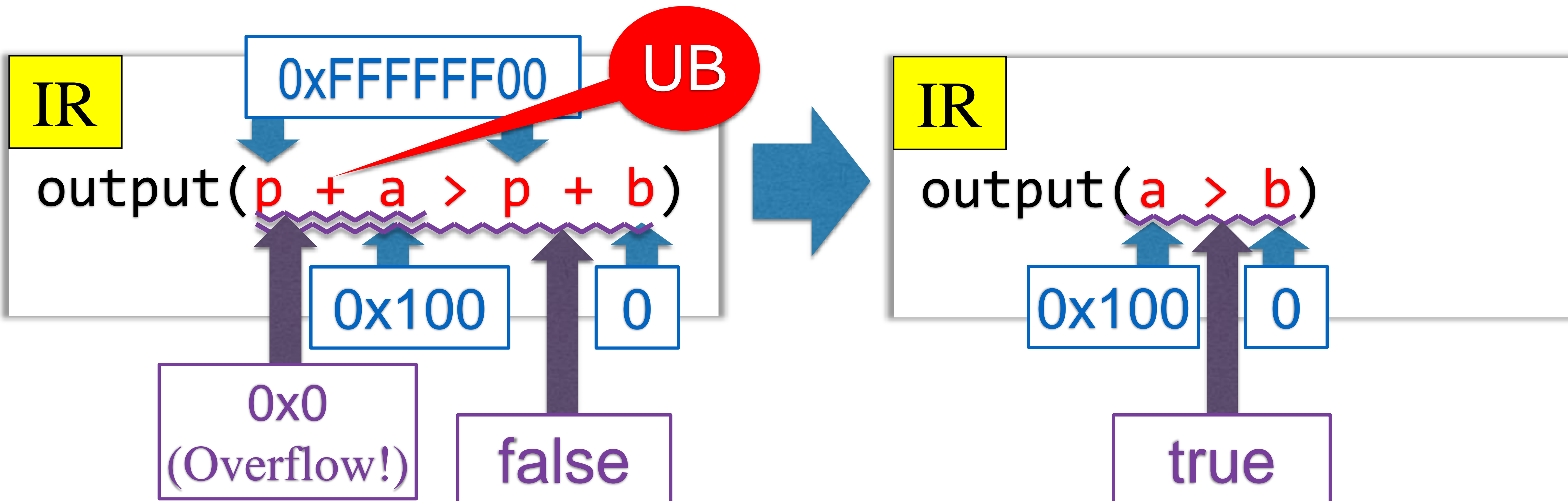
UB in C ≠ UB in IR

Peephole Optimization

C's UB Model:

Pointer Arithmetic Overflow is

Undefined Behavior



UB in C ≠ UB in IR

Loop Invariant Code Motion

C's UB Model:

Pointer Arithmetic Overflow is

Undefined Behavior

IR

```
...
for(i=0; i<n; ++i)
{
  a[i] = p + 0x100
}
```

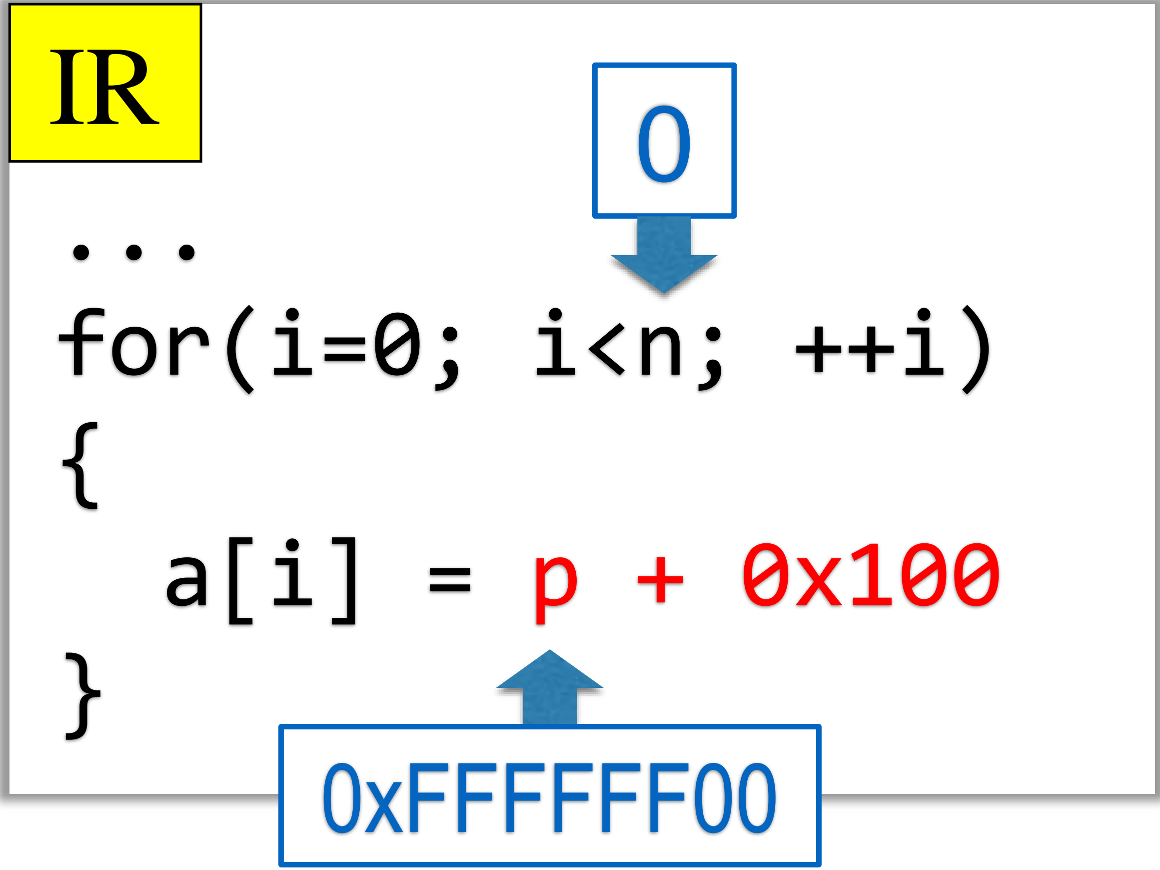
IR

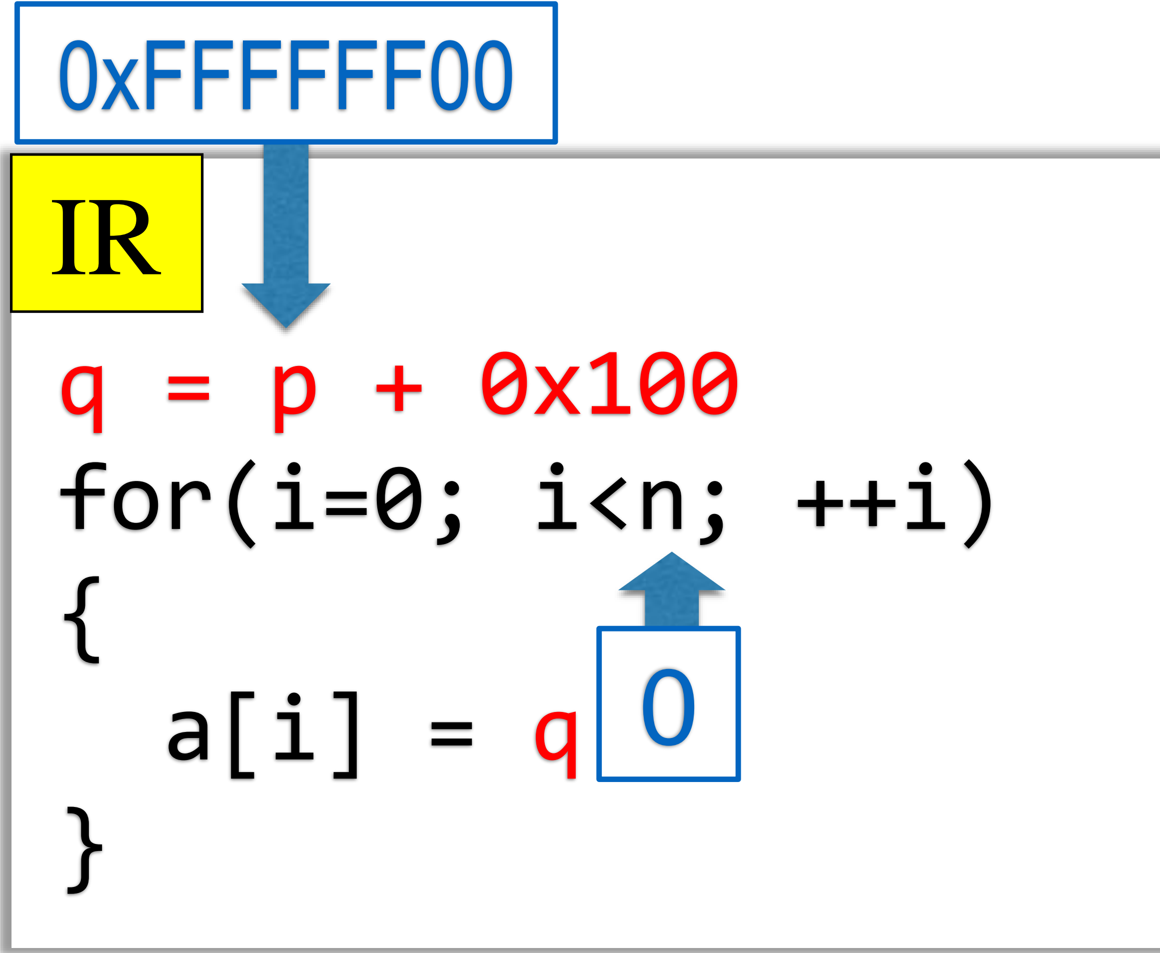
```
q = p + 0x100
for(i=0; i<n; ++i)
{
  a[i] = q
}
```

UB in C ≠ UB in IR

Loop Invariant Code Motion

C's UB Model:
Pointer Arithmetic Overflow is
Undefined Behavior





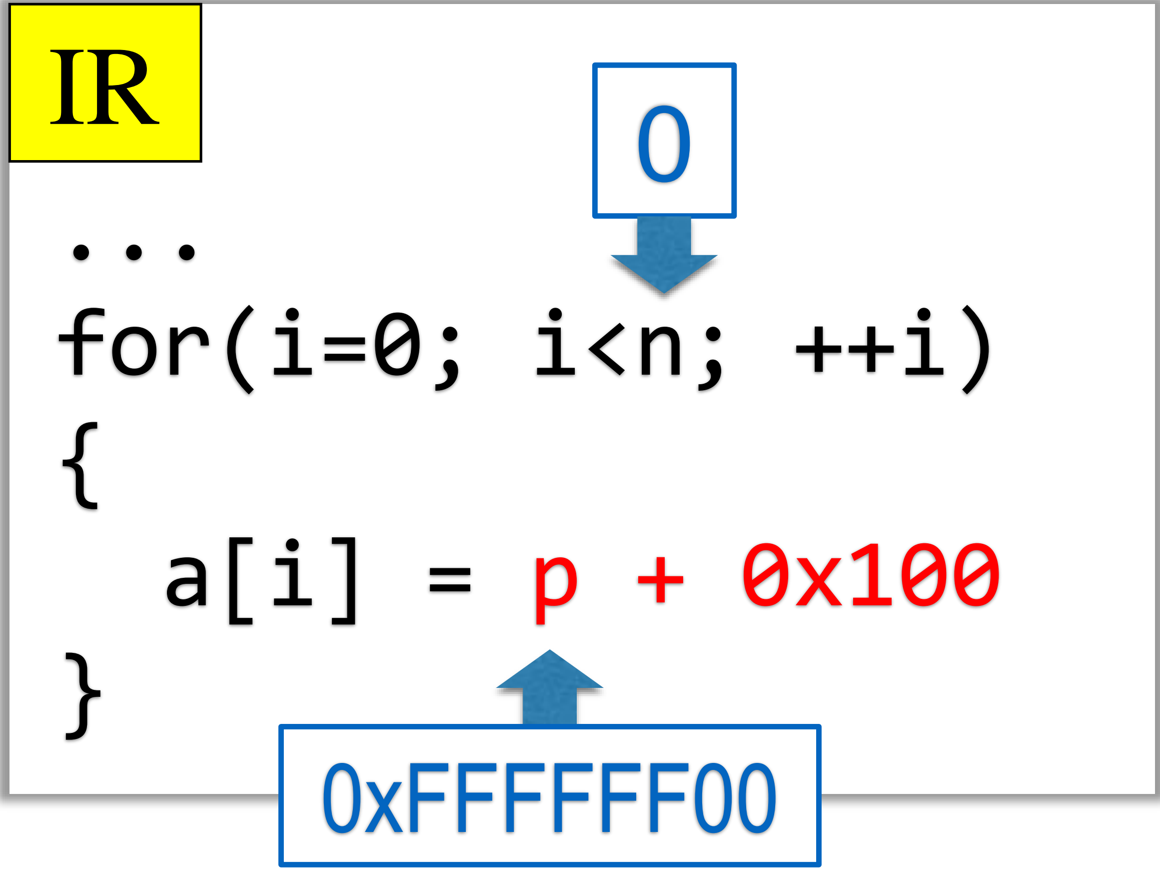
UB in C ≠ UB in IR

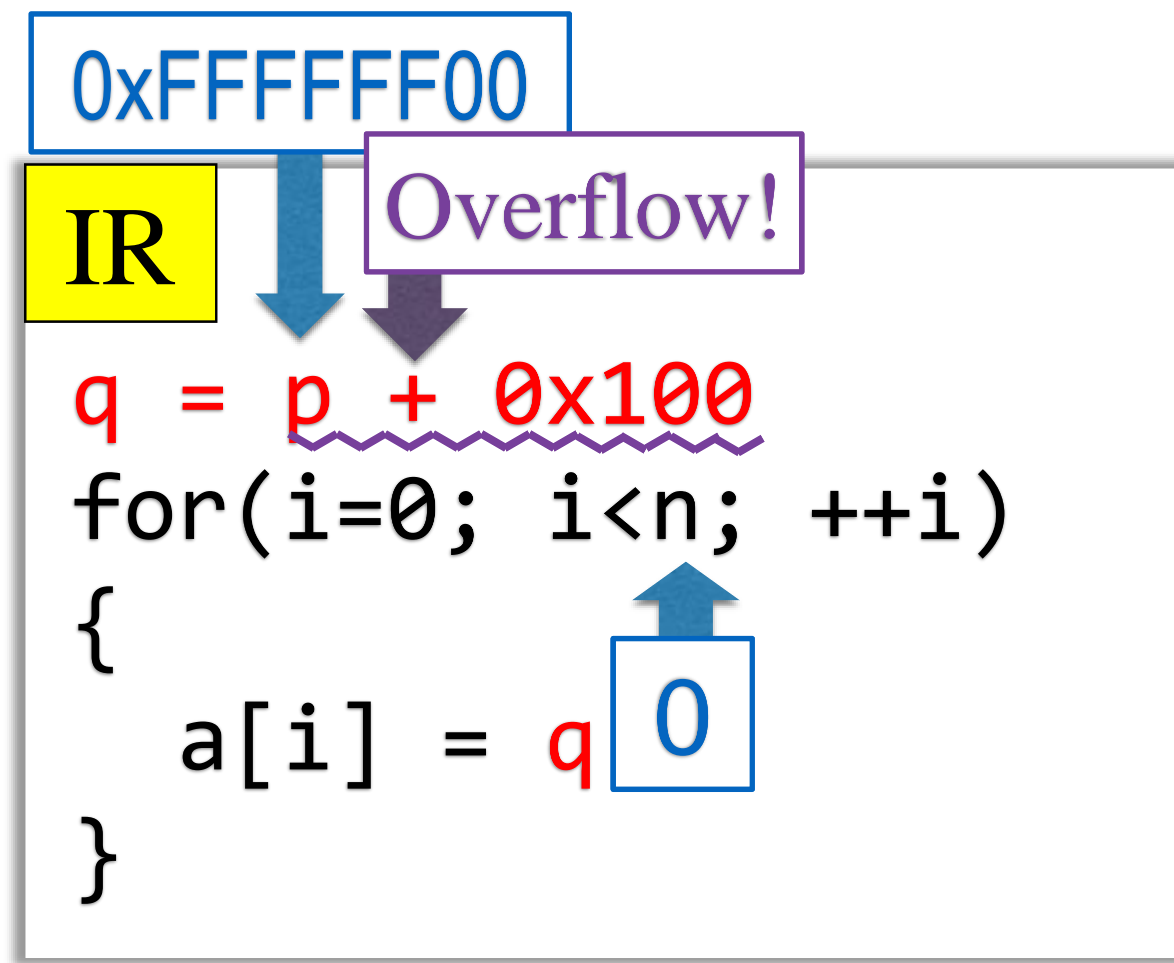
Loop Invariant Code Motion

C's UB Model:

Pointer Arithmetic Overflow is

Undefined Behavior





UB in C ≠ UB in IR

Loop Invariant Code Motion

C's UB Model:

Pointer Arithmetic Overflow is

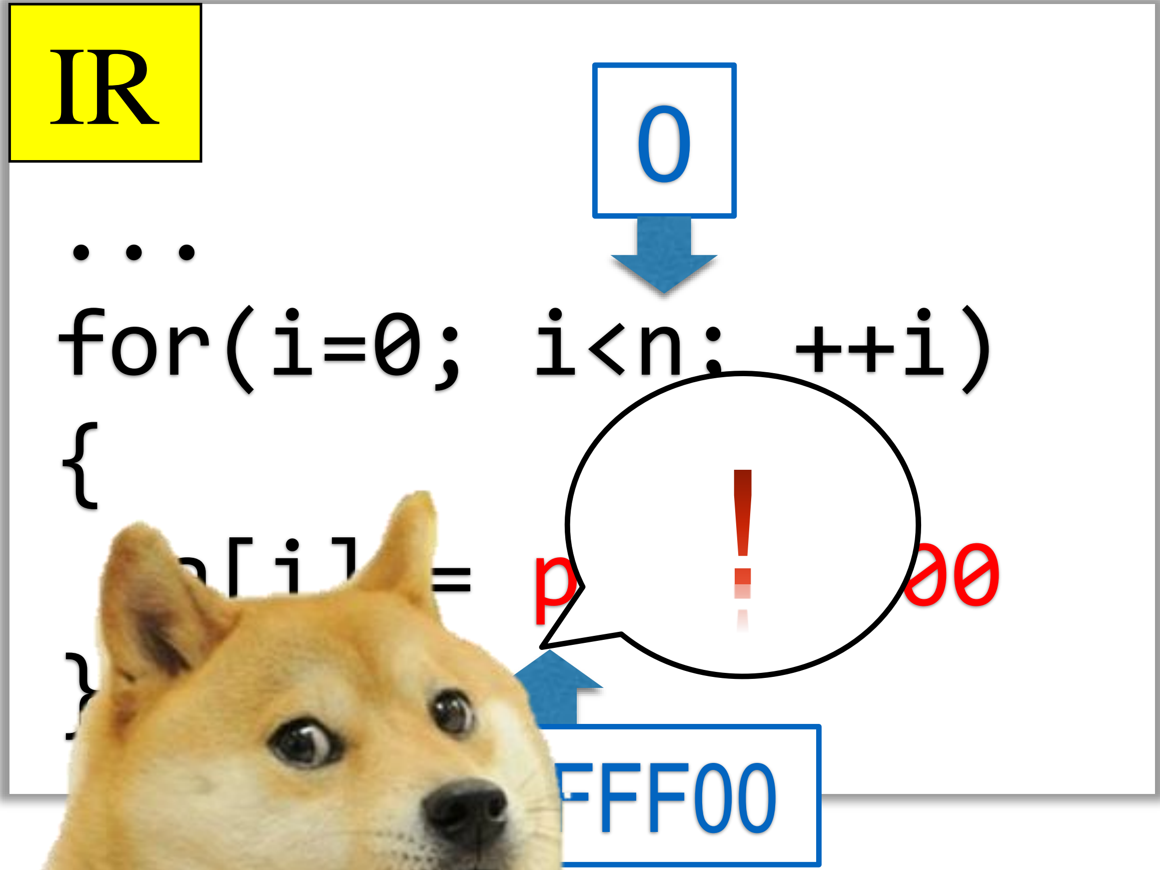
Undefined Behavior

IR

0

```
...
for(i=0; i<n; ++i)
{
  a[i] = p    00
}
```

FFF00



0xFFFFFF00

Overflow!

UB

IR

```
q = p + 0x100
for(i=0; i<n; ++i)
{
  a[i] = q
}
```

0

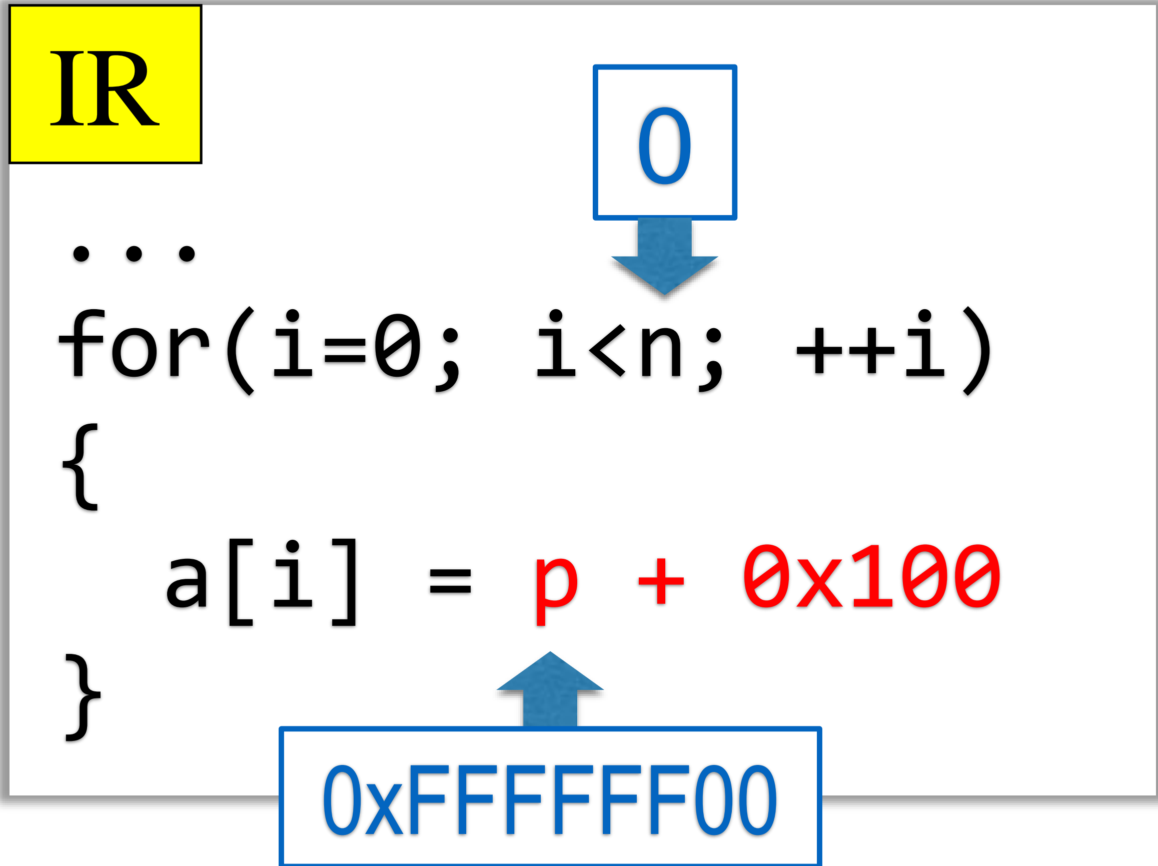
UB in C ≠ UB in IR

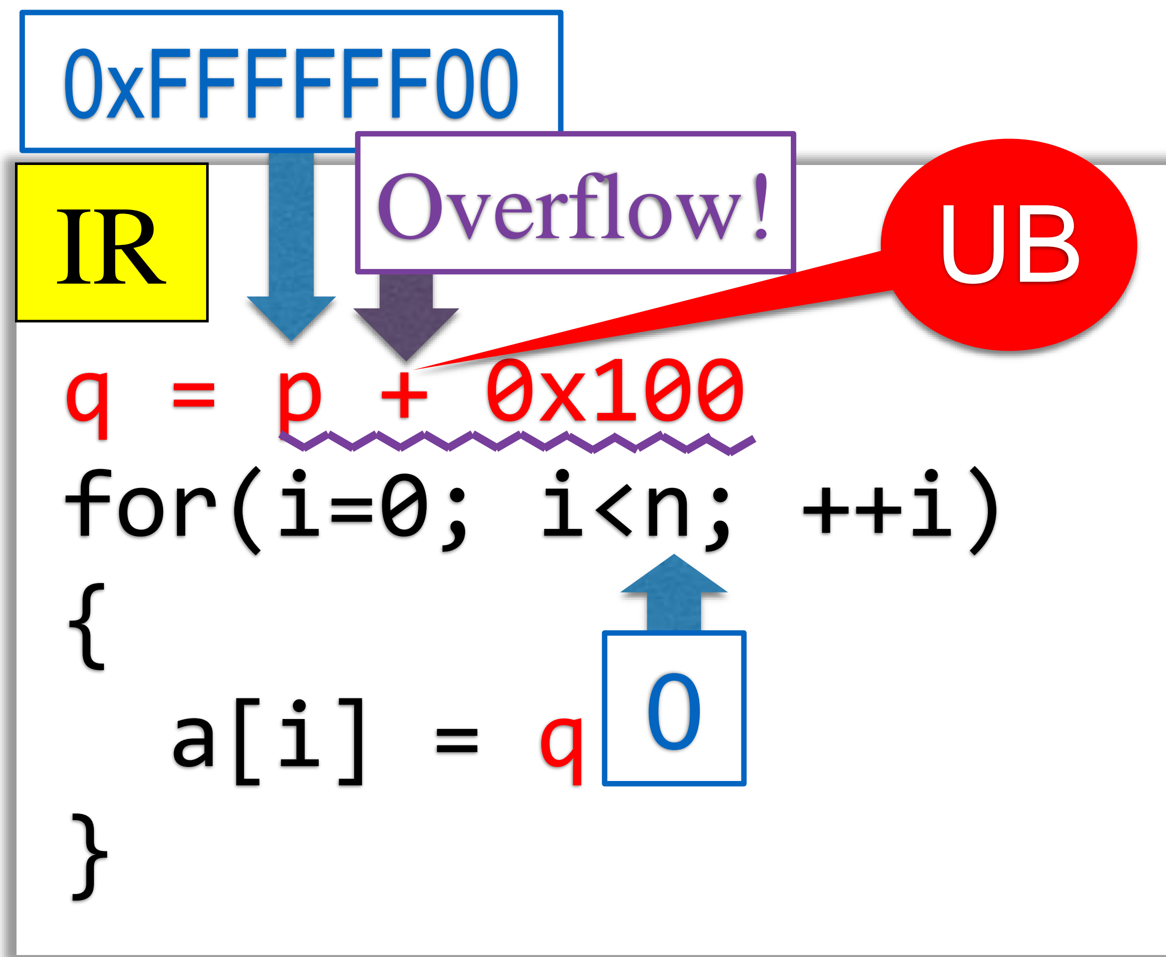
Poison Value: A Deferred UB

C's UB Model:

Pointer Arithmetic Overflow is

Undefined Behavior





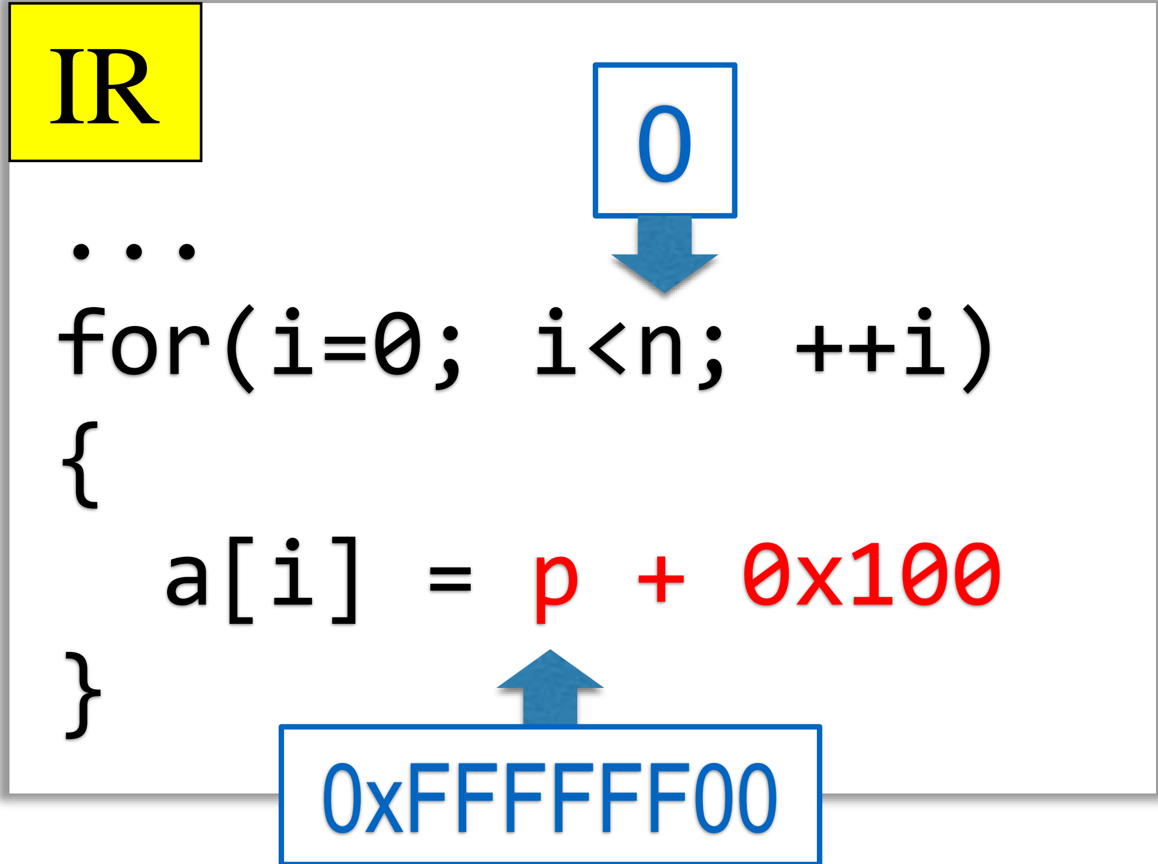
UB in C ≠ UB in IR

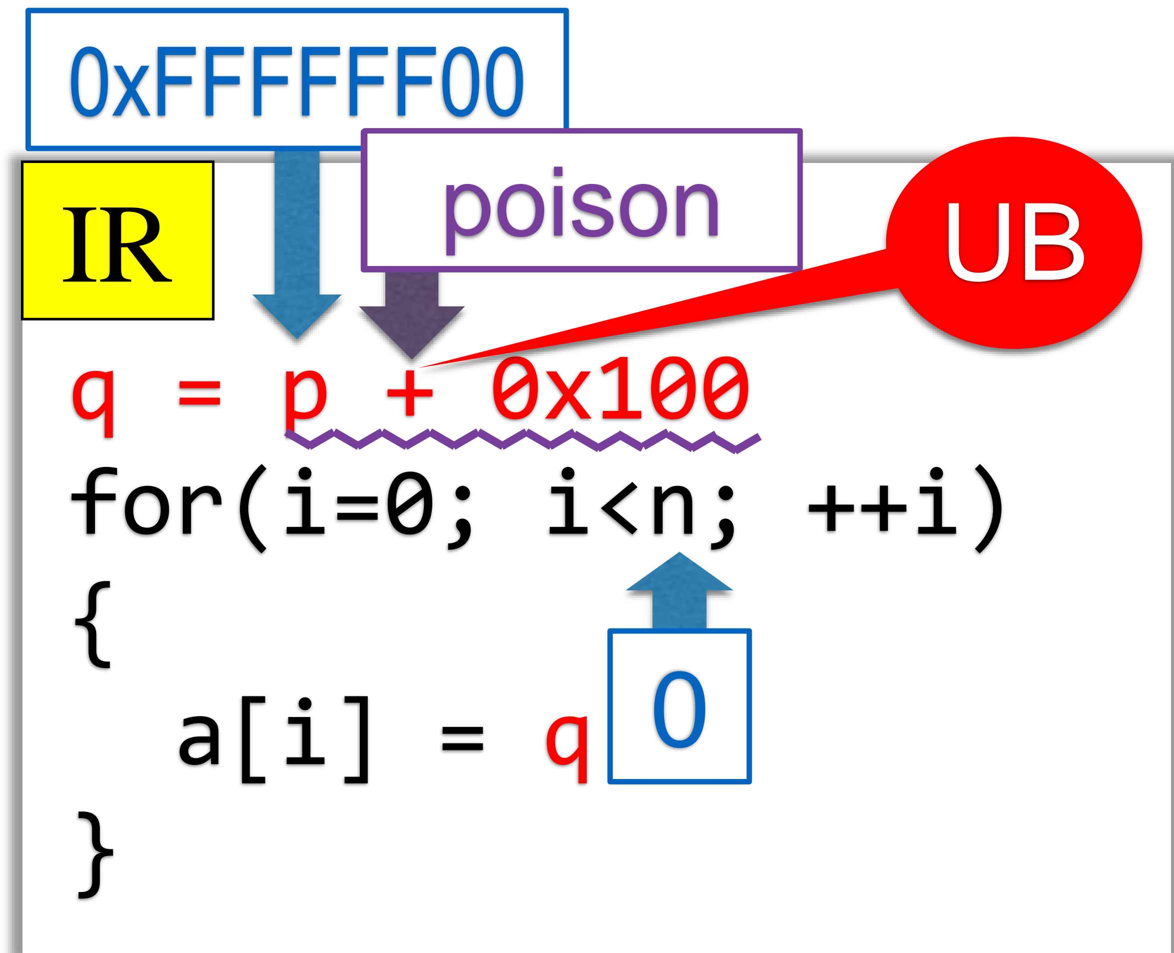
Poison Value: A Deferred UB

LLVM's UB Model:

Pointer Arithmetic Overflow is

A Poison "Value"

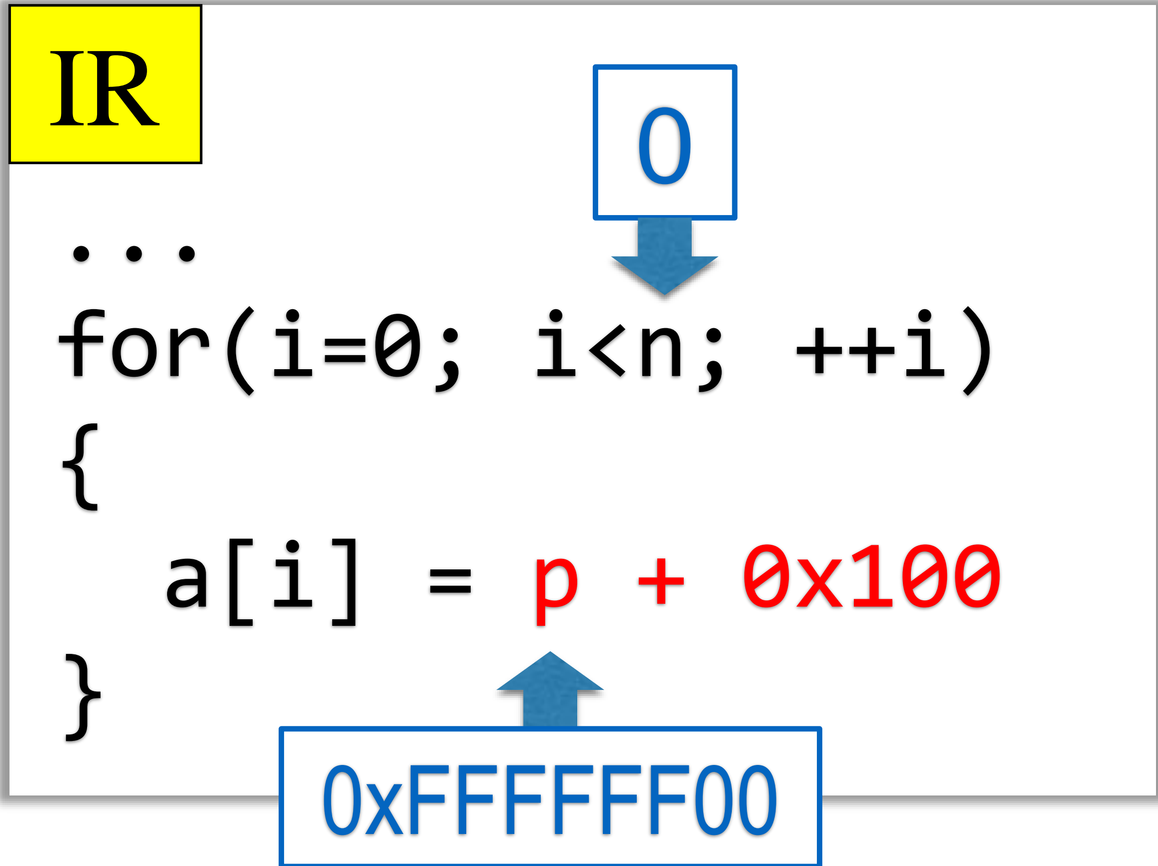


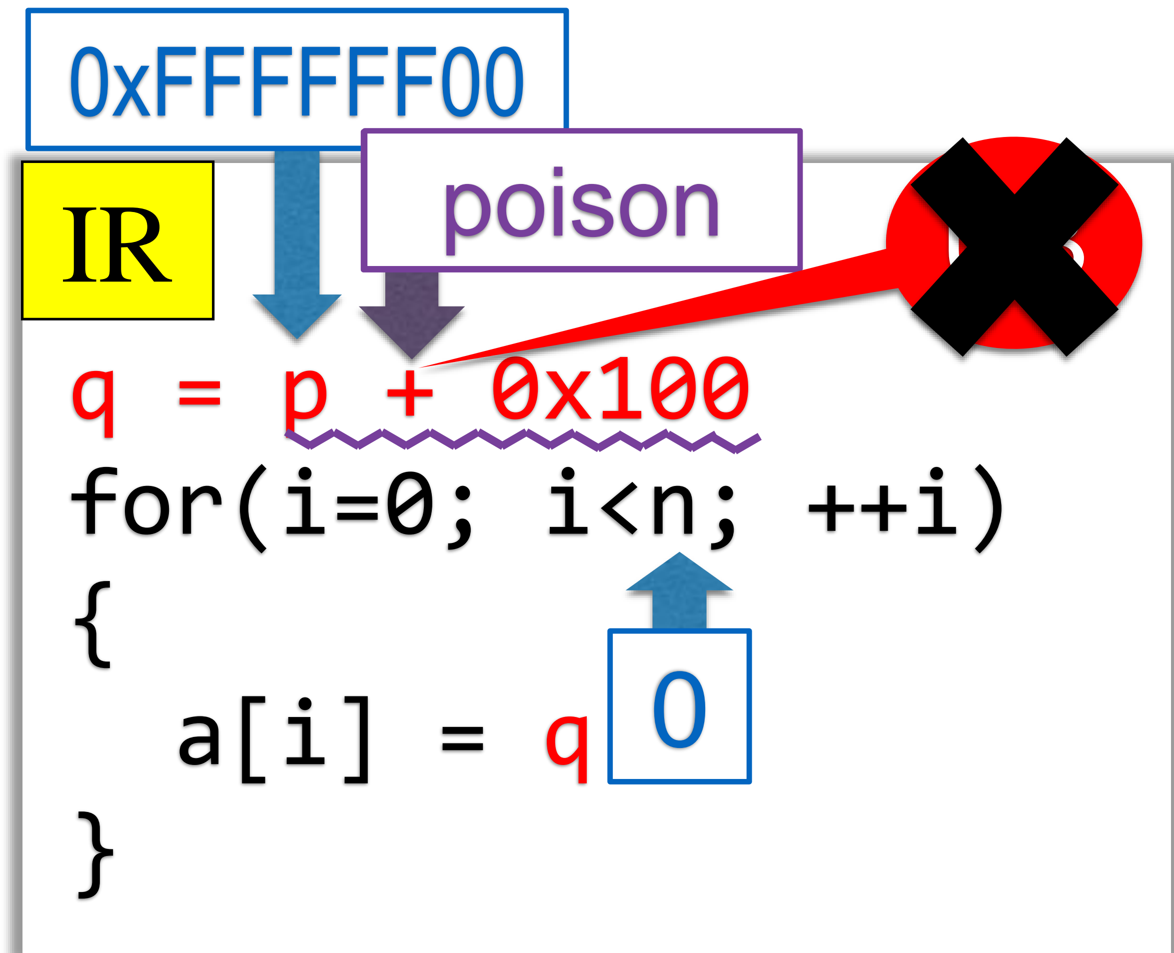


UB in C ≠ UB in IR

Poison Value: A Deferred UB

LLVM's UB Model:
Pointer Arithmetic Overflow is
A Poison "Value"





UB in C ≠ UB in IR

Poison Value: A Deferred UB

LLVM's UB Model:
Pointer Arithmetic Overflow is
A Poison "Value"

IR

```
output(p + a > p + b)
```

0xFFFFFF00 | UB | 0x100 | 0 | 0x0 (Overflow!)

IR

```
output(a > b)
```

0x100 | 0

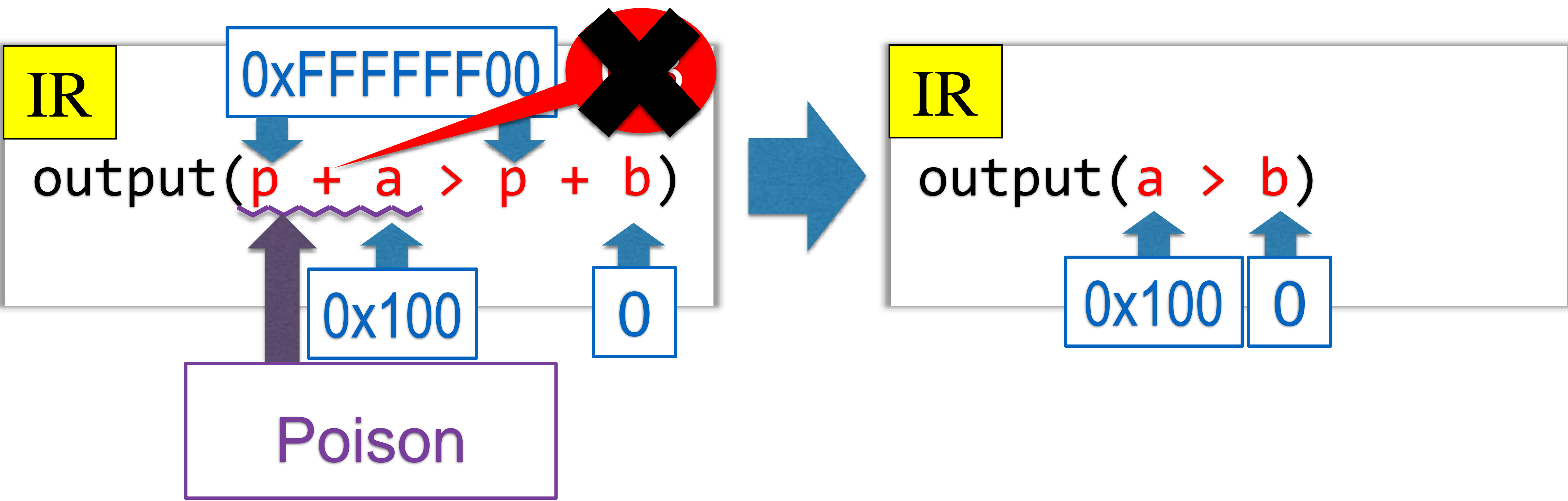
UB in C ≠ UB in IR

Poison Value: A Deferred UB

LLVM's UB Model:

Pointer Arithmetic Overflow is

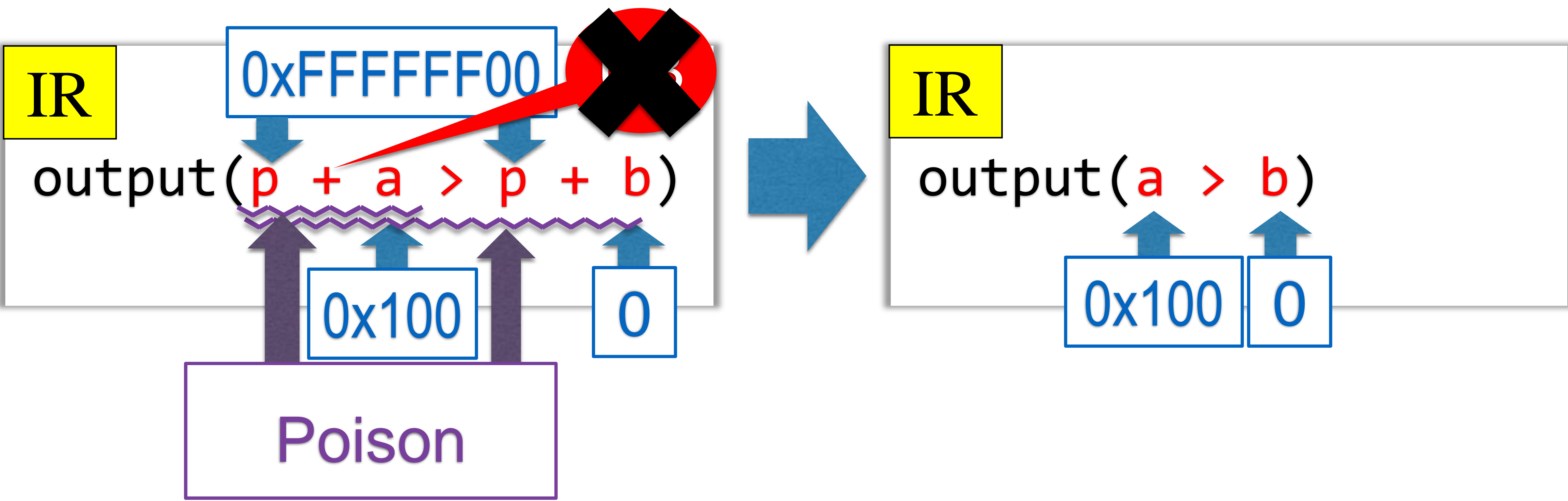
A Poison "Value"



UB in C ≠ UB in IR

Poison Value: A Deferred UB

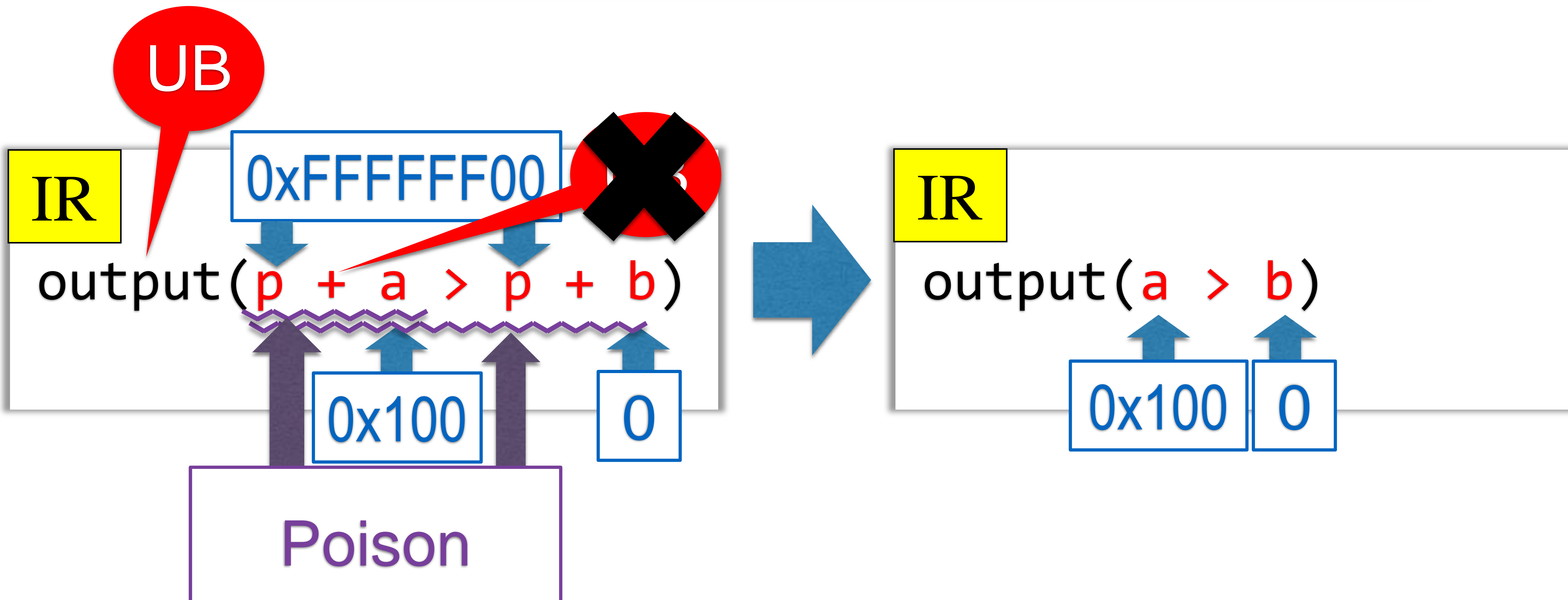
LLVM's UB Model:
Pointer Arithmetic Overflow is
A Poison "Value"



UB in C ≠ UB in IR

Poison Value: A Deferred UB

LLVM's UB Model:
Pointer Arithmetic Overflow is
A Poison "Value"



UB in IR is only for C?

- Example: Java
  - Type checker:
    "Function args are either null or dereferenceable."
  - Put 'dereferenceable_or_null' tag to them!
  - It's UB for them to have invalid pointers

Summary

- C's UB $\neq$ LLVM IR's UB.
- The notion of 'deferred UB' helps further opt.
- UB works for well-typed languages, too

Problem of UB in LLVM IR & Solution

PLDI 2017 Barcelona

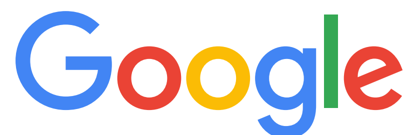
Taming Undefined Behavior in LLVM

Seoul National Univ.

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur

Azul Systems

Sanjoy Das



Google

David Majnemer

University of Utah

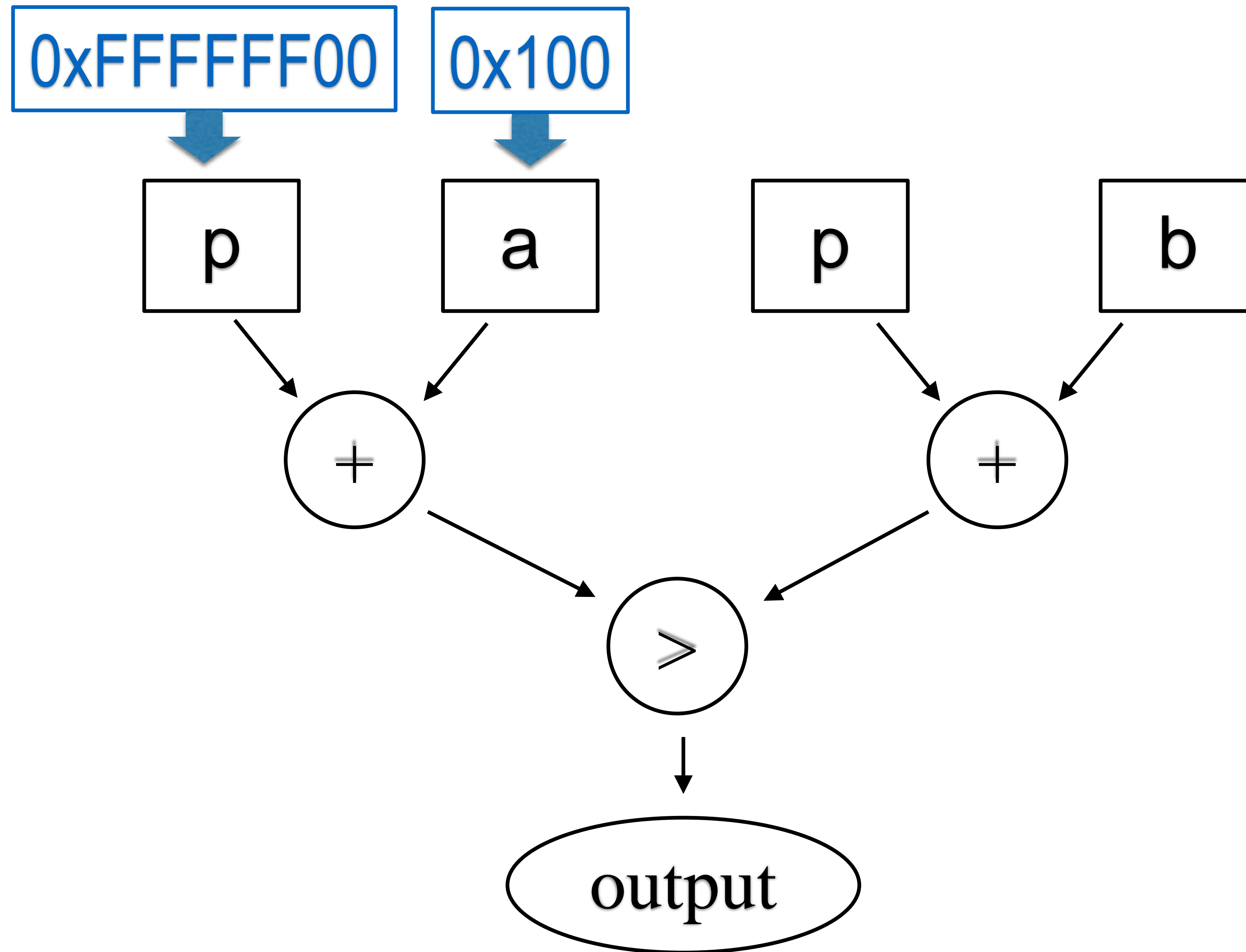
John Regehr

Microsoft Research

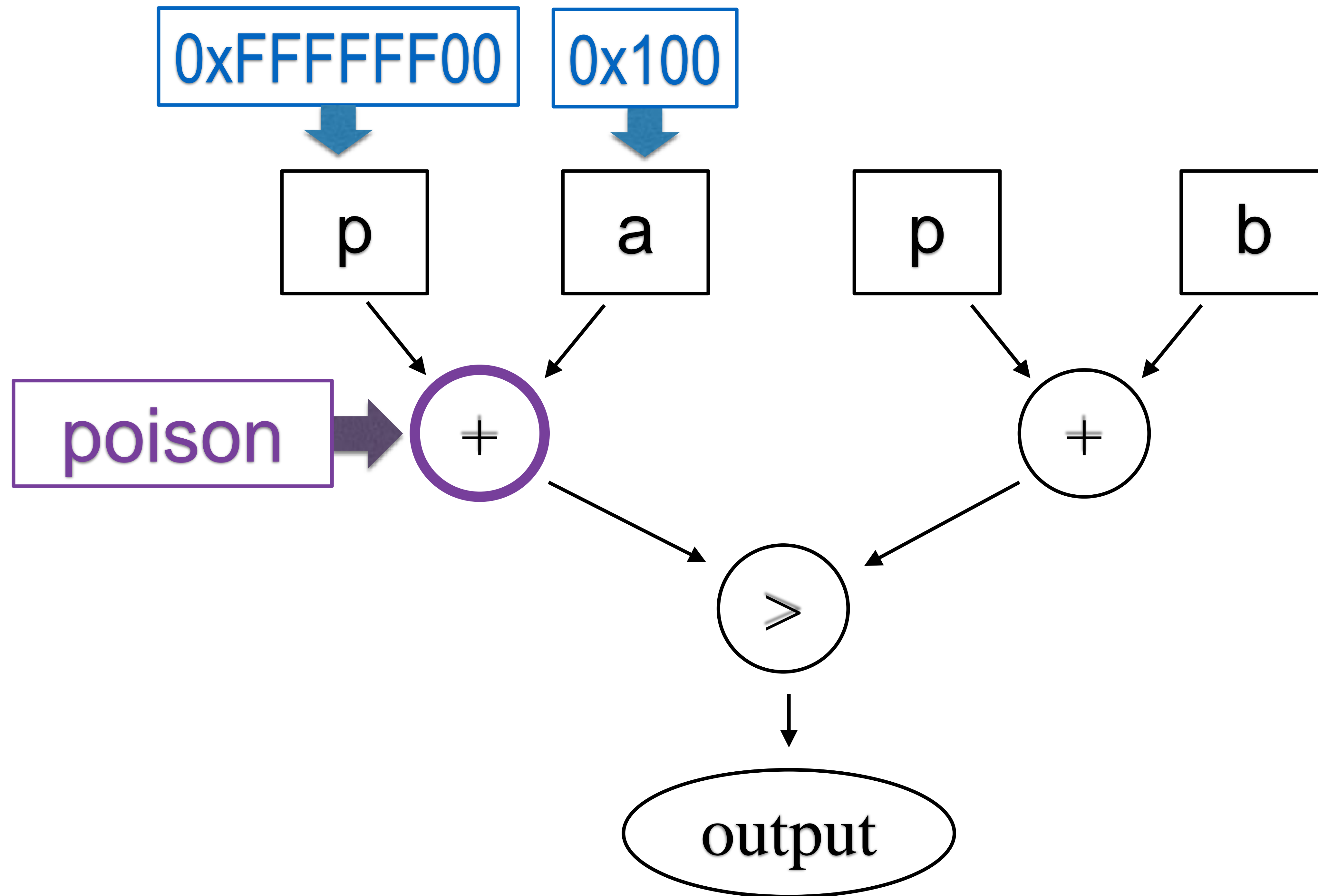
Nuno P. Lopes

Software Foundations Lab

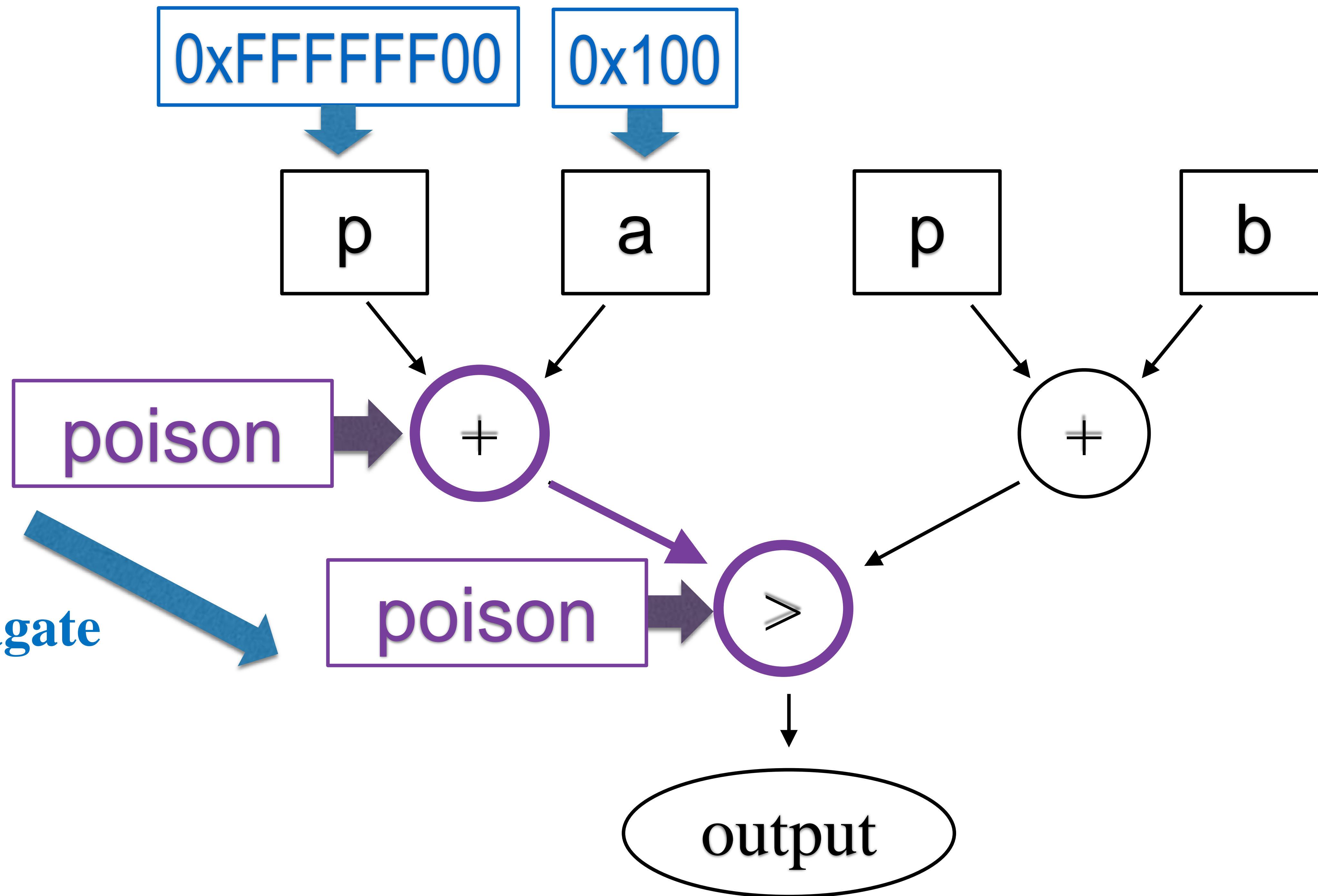
Problem of Poison



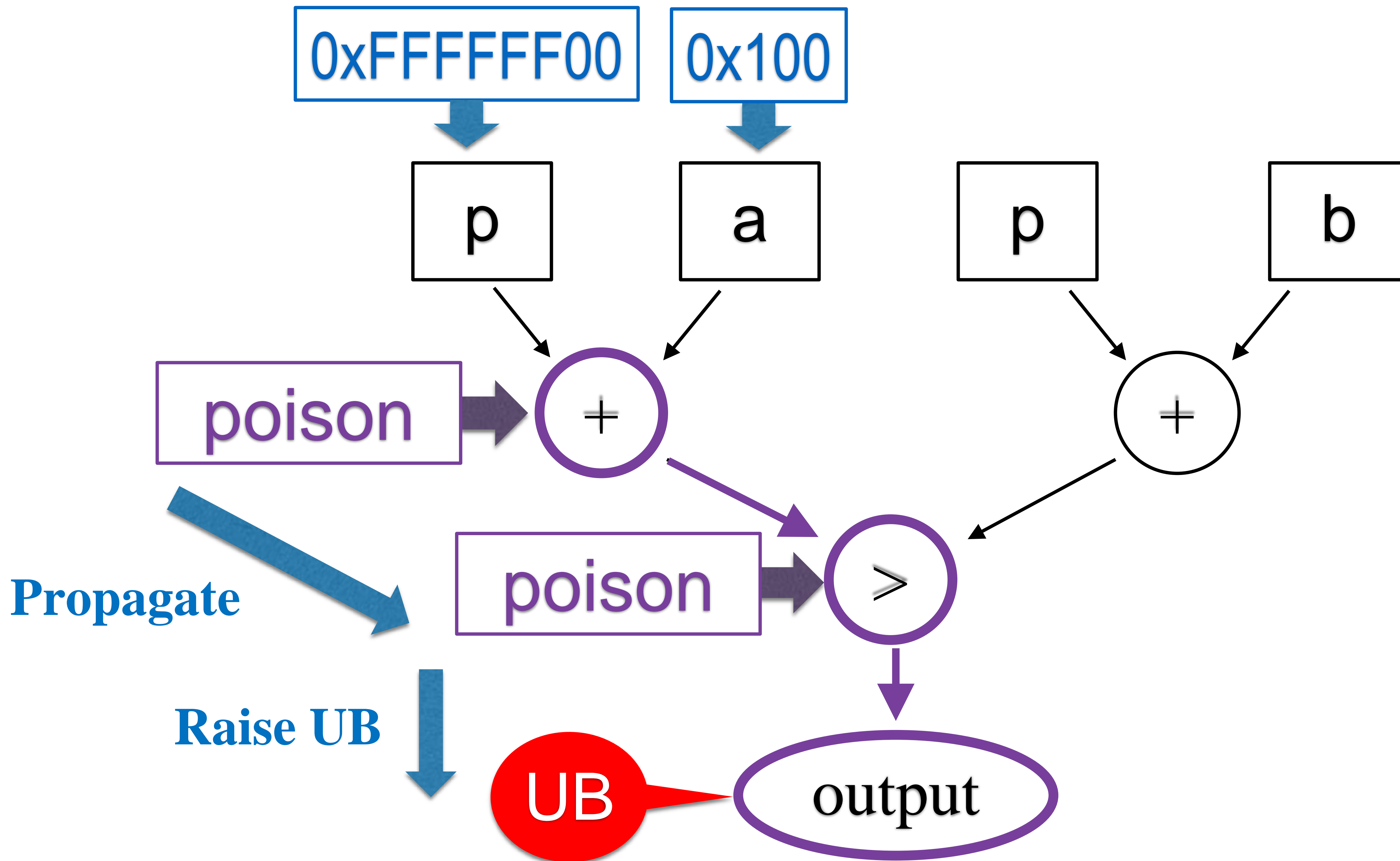
Problem of Poison



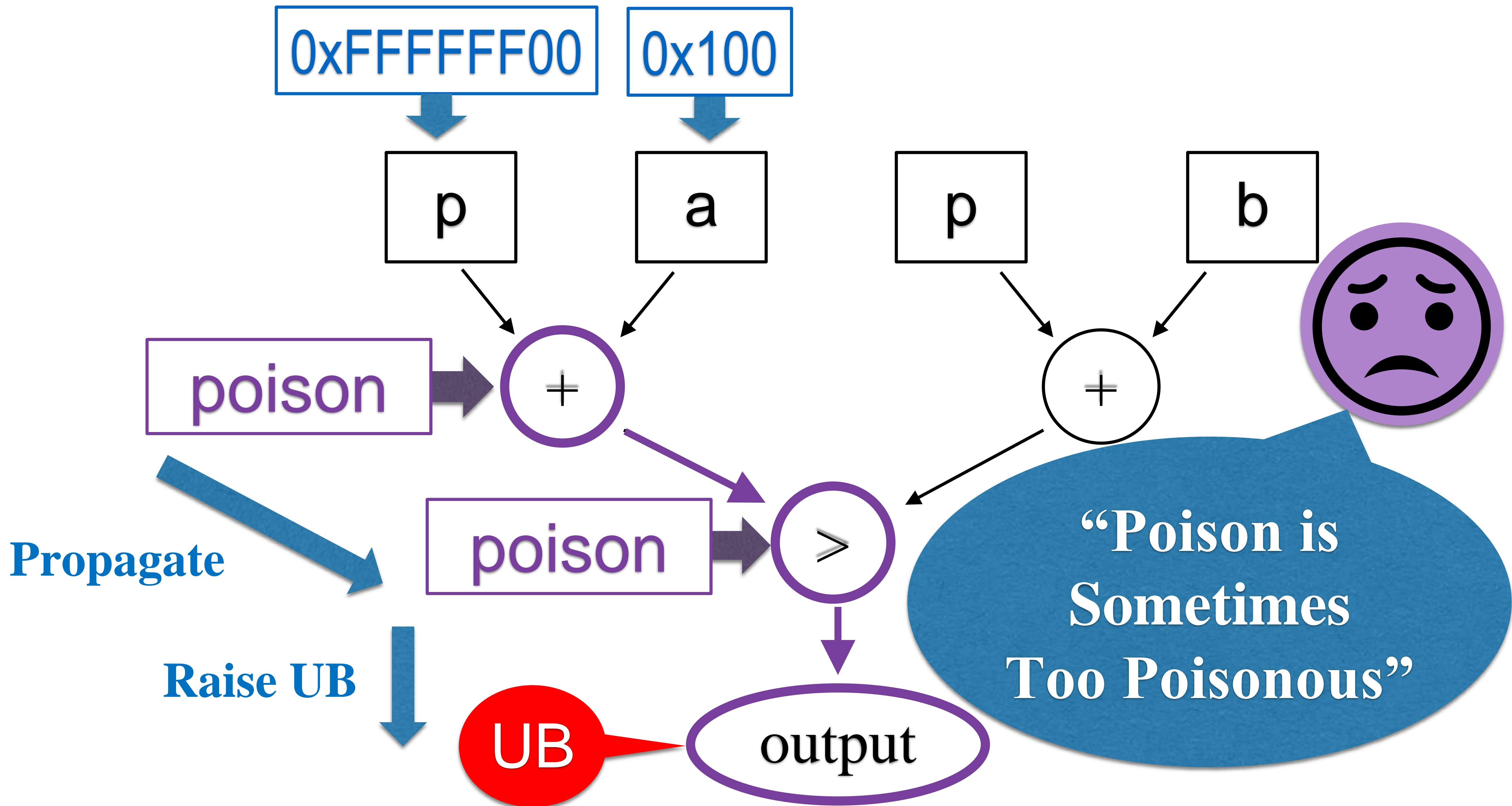
Problem of Poison



Problem of Poison



Problem of Poison



Global Value Numbering (GVN)

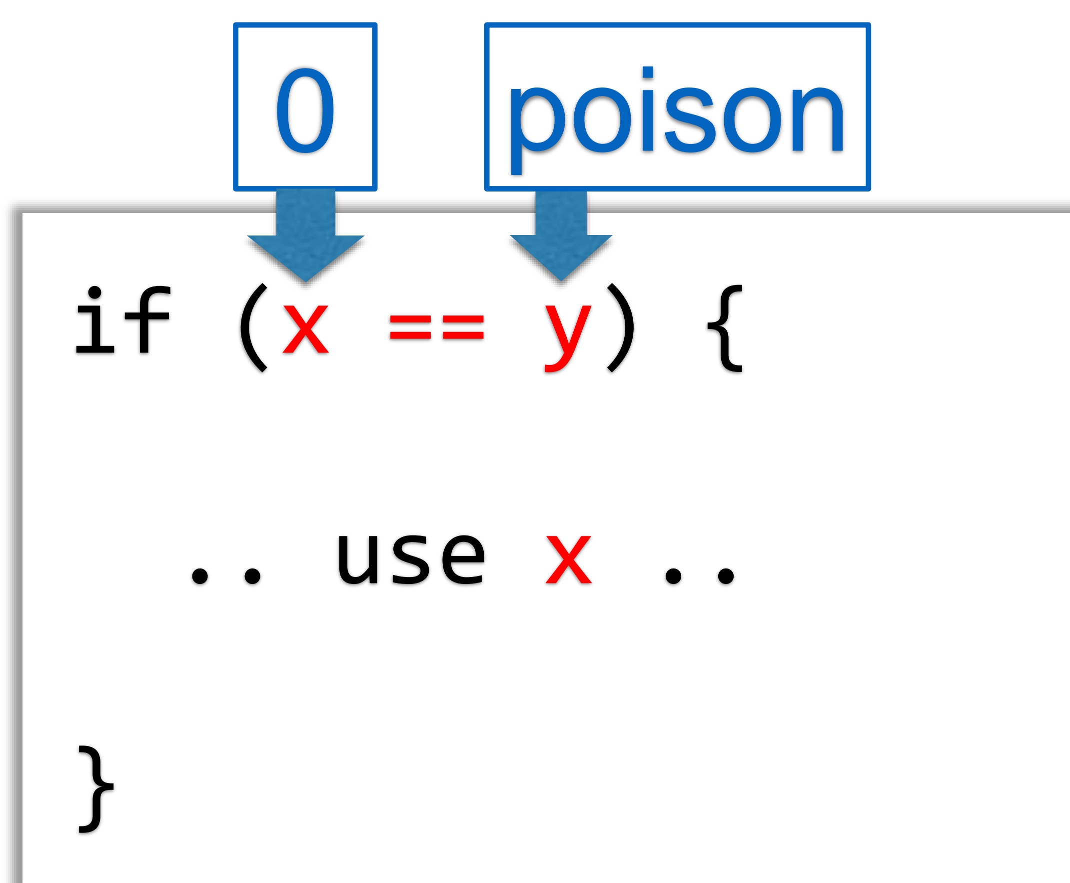
LLVM's UB Model:
Branching on poison is
???

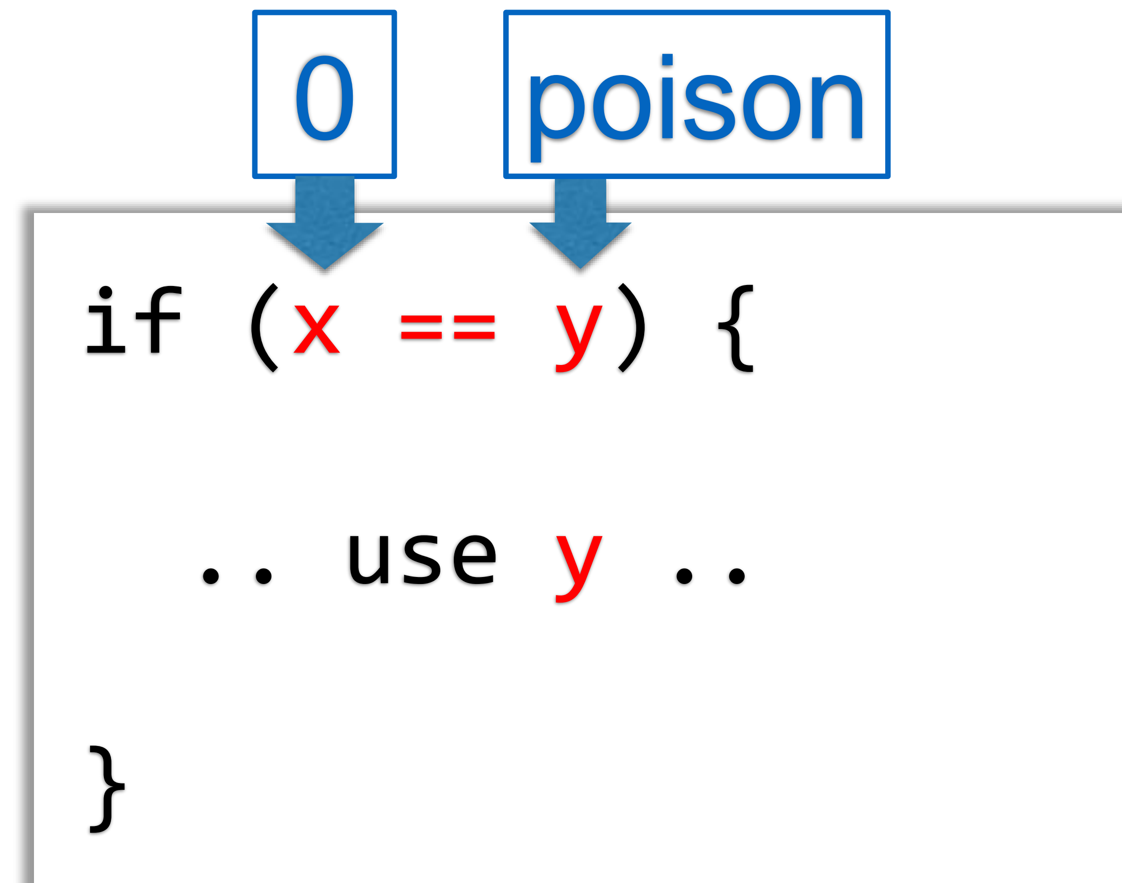
```
if (x == y) {
  .. use x ..
}
```

```
if (x == y) {
  .. use y ..
}
```

Global Value Numbering (GVN)

LLVM's UB Model:
Branching on poison is
???



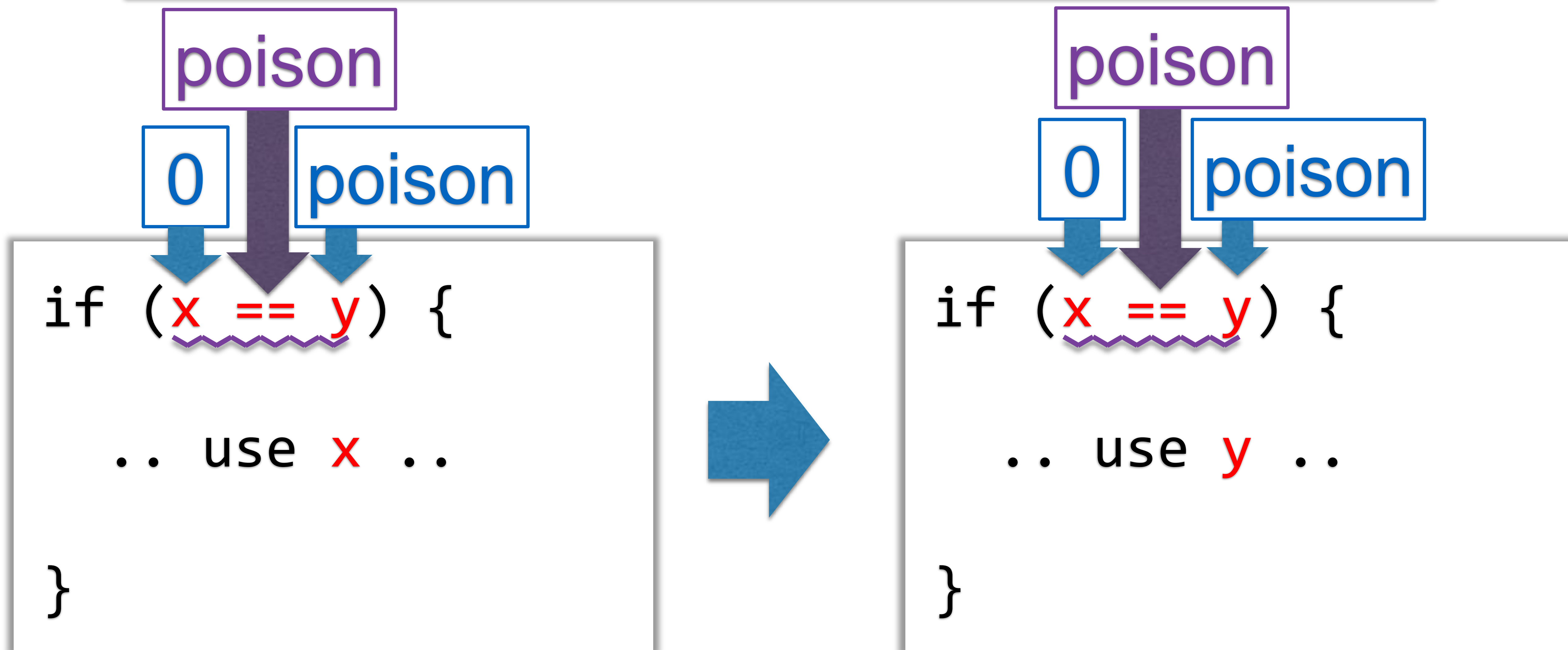


Global Value Numbering (GVN)

LLVM's UB Model:

Branching on poison is

???



Global Value Numbering (GVN)

LLVM's UB Model:
Branching on poison is
???

poison
0
poison

```
if (x == y) {
  .. use x ..
}
```

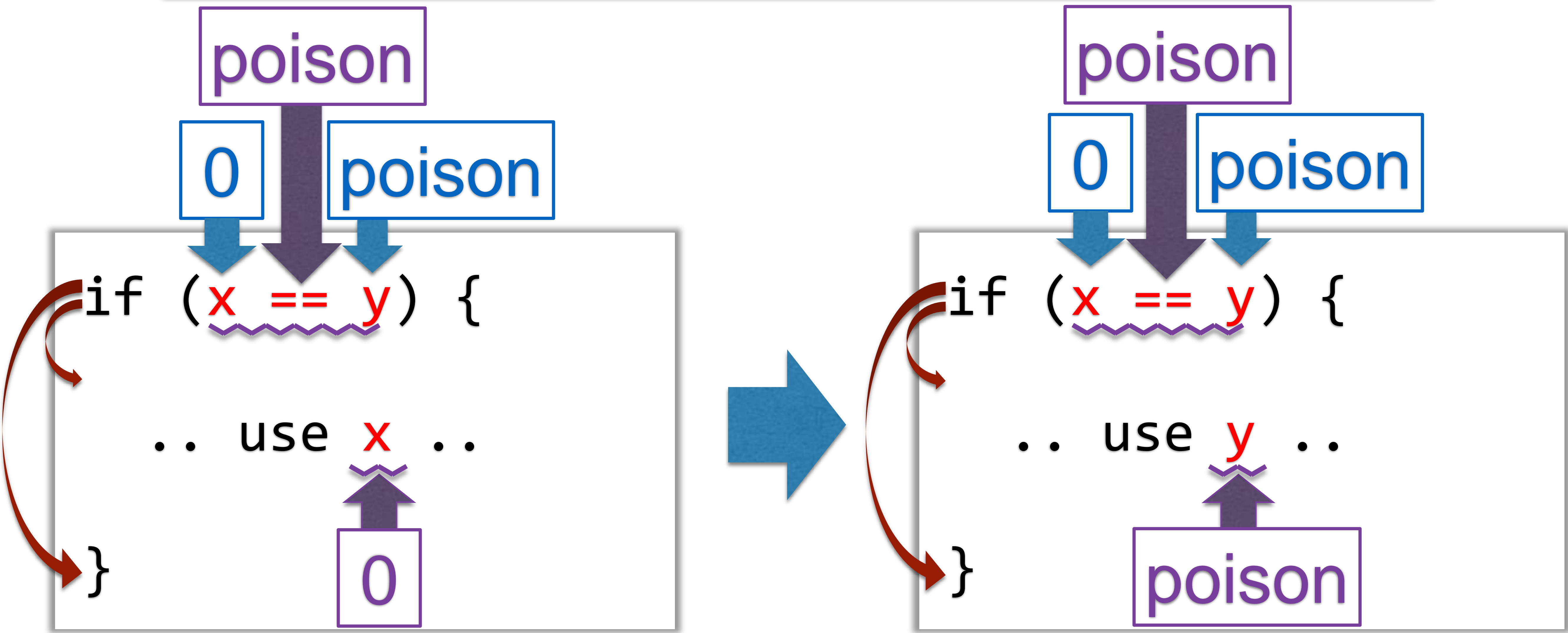
poison
0
poison

```
if (x == y) {
  .. use y ..
}
```

Problems with LLVM's UB

Global Value Numbering (GVN)

LLVM's UB Model:
Branching on poison is
???



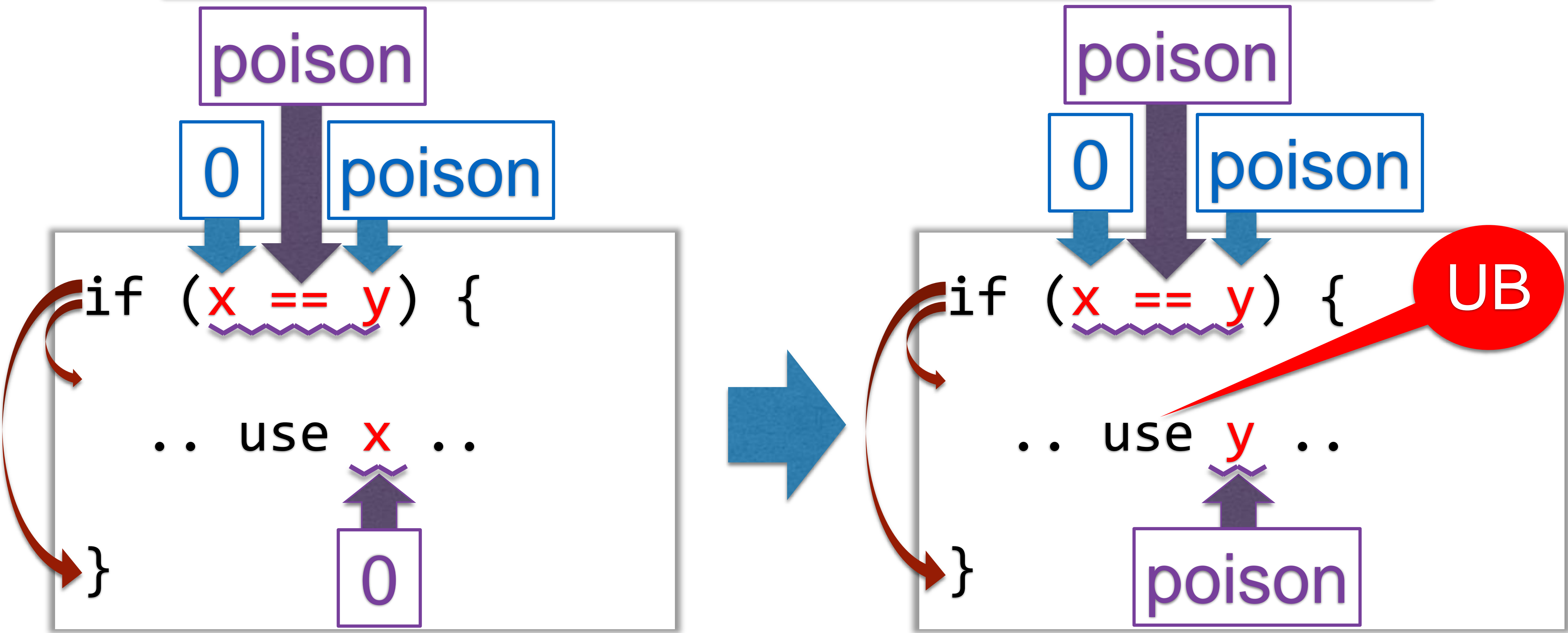
Problems with LLVM's UB

Global Value Numbering (GVN)

LLVM's UB Model:

Branching on poison is

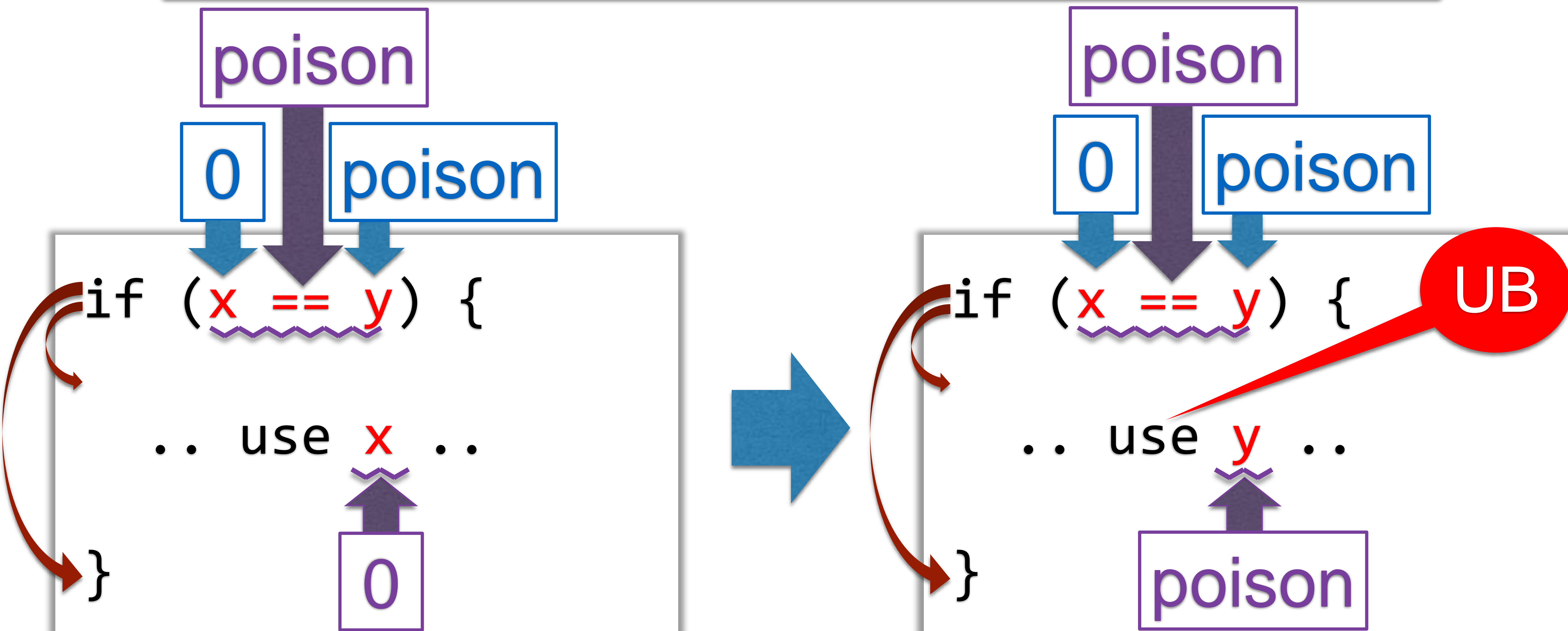
???



Problems with LLVM's UB

Global Value Numbering (GVN)

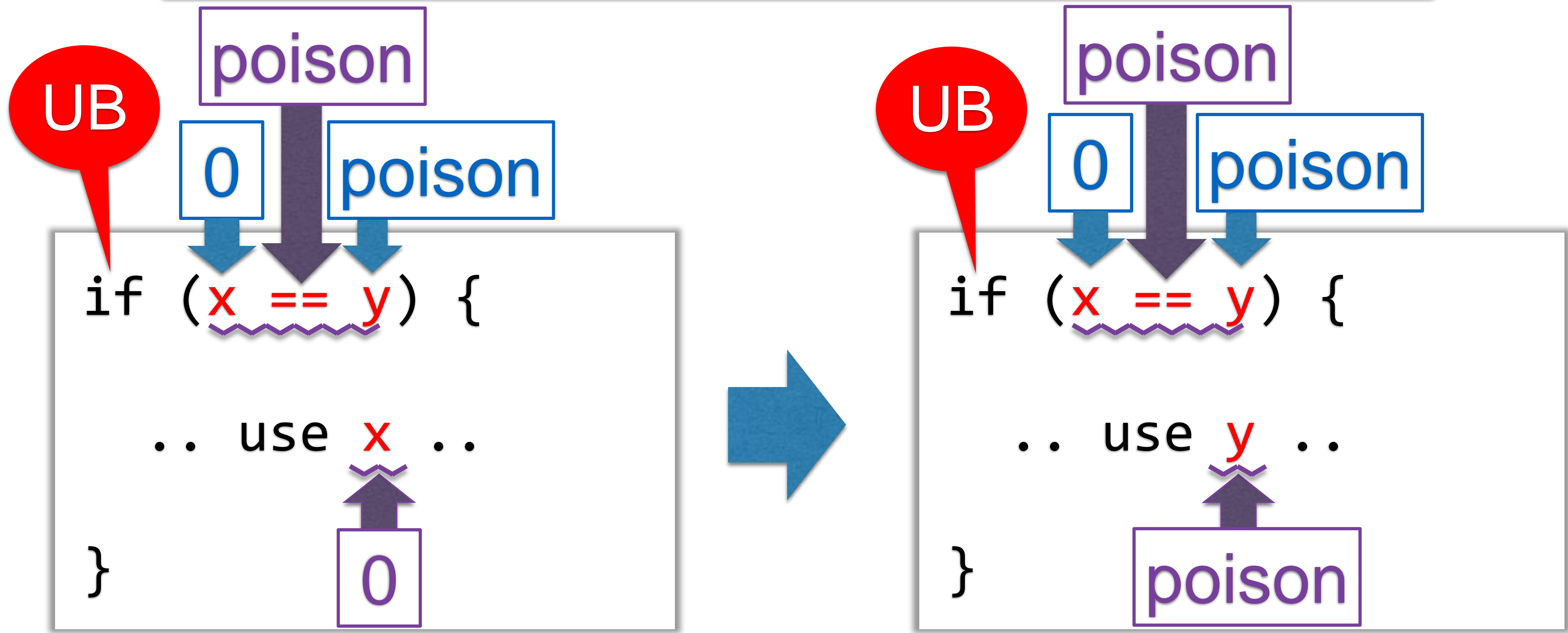
LLVM's UB Model:
Branching on poison is
Undefined Behavior



Problems with LLVM's UB

Global Value Numbering (GVN)

LLVM's UB Model:
Branching on poison is
Undefined Behavior



Loop Unswitching (LU)

LLVM's UB Model:
Branching on poison is
Undefined Behavior

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

```
if (cond)
  while (n > 0)
  { A }
else
  while (n > 0)
  { B }
```

Loop Unswitching (LU)

LLVM's UB Model:
Branching on poison is
Undefined Behavior

0

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

poison

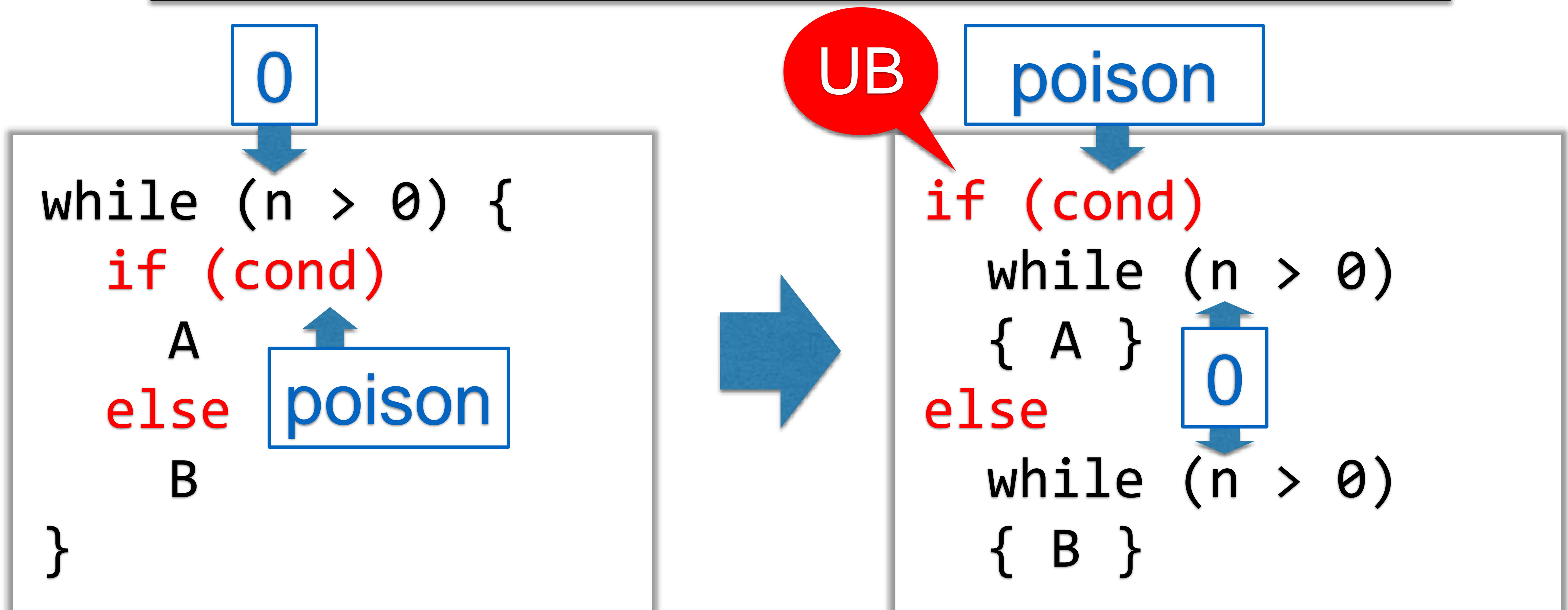
poison

```
if (cond)
  while (n > 0)
  { A }
else
  while (n > 0)
  { B }
```

0

Loop Unswitching (LU)

LLVM's UB Model:
Branching on poison is
Undefined Behavior

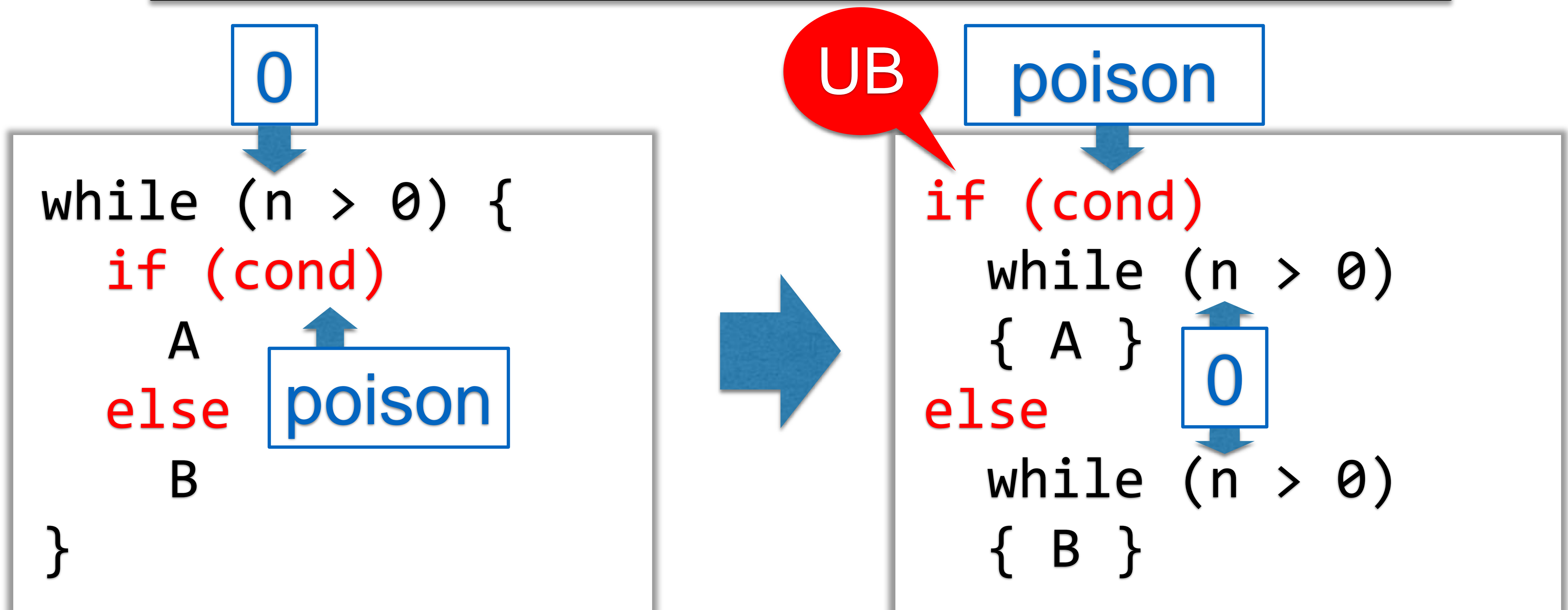


Loop Unswitching (LU)

LLVM's UB Model:

Branching on poison is

Undefined Behavior



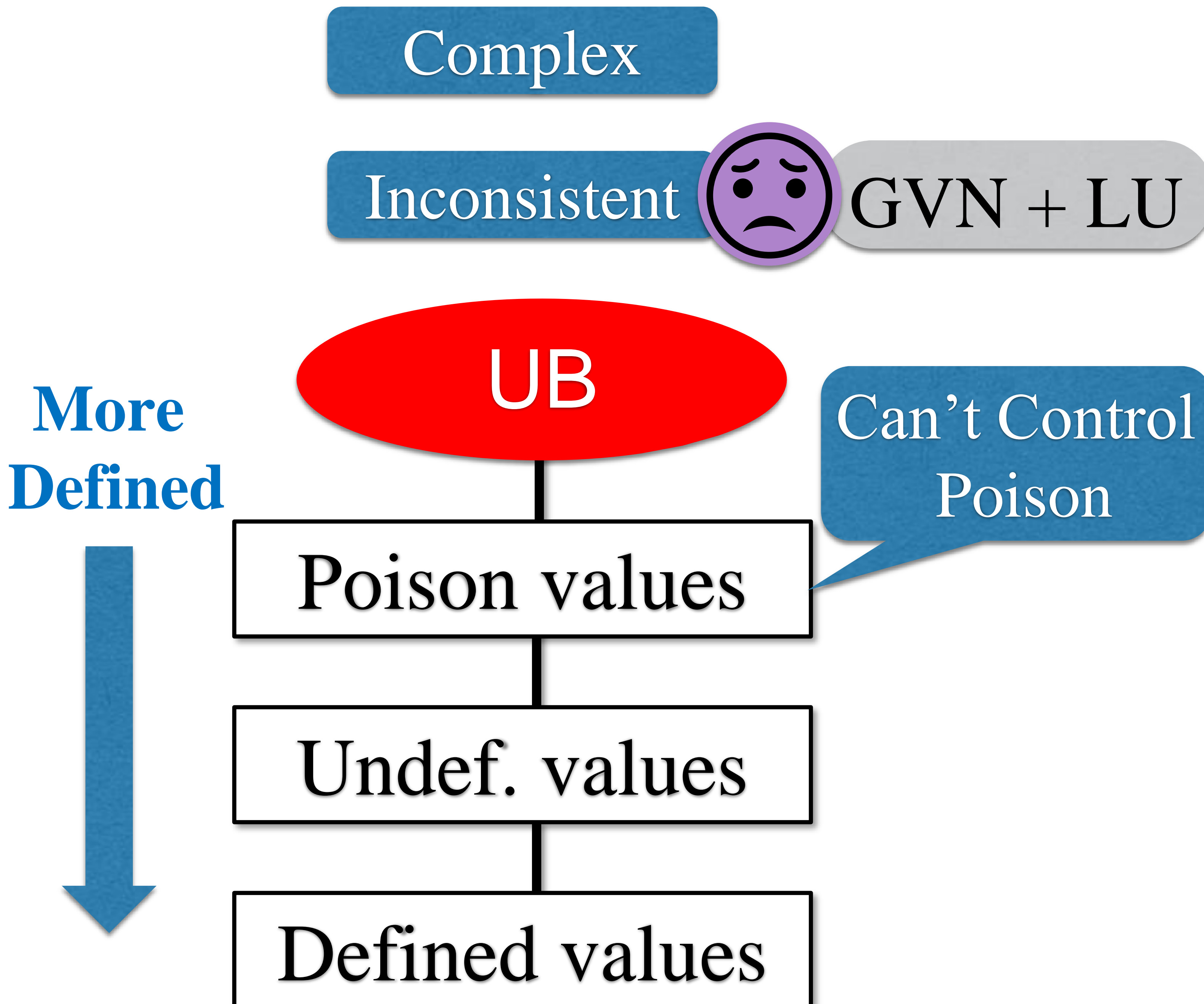
Inconsistency in LLVM

- GVN + LU is inconsistent.

- We found a miscompilation bug in LLVM due to the inconsistency (LLVM Bugzilla 31652).
  - It is being discussed in the community
  - No solution has been found yet

Overview

Existing Approaches



Overview

Existing Approaches	Our Approach
Complex	Simpler
Inconsistent	

GVN + LU

Existing approaches:

UB

Poison values

Undef. values

Defined values

Can't Control Poison

Our approach:

UB

Poison values

Defined values

freeze

Overview

Existing Approaches

Complex

Inconsistent GVN + LU

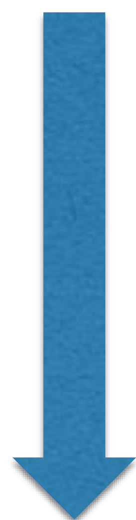
UB

Can't Control Poison

Poison values

Undef. values

Defined values



More Defined

Our Approach

Simpler

UB

Poison values

Can Control Poison

freeze

Defined values

Overview

Existing Approaches | **Our Approach**

Key Idea: “Freeze”

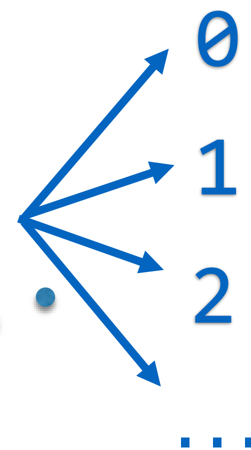
- Introduce a new instruction

```
y = freeze x
```

- Semantics:

When `x` is a defined value: `freeze x` ⟶ `x`

When `x` is a poison value: `freeze x`



Nondet. Choice of
A Defined Value

Our Solution

Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior

0

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

UB

poison

```
if (cond)
  while (n > 0)
  { A }
else
  while (n > 0)
  { B }
```

Our Solution

Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior

0

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

UB

poison

```
if (freeze(cond))
   while (n > 0)
   { A }
else
   while (n > 0)
   { B }
```

Our Solution

Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior

0

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

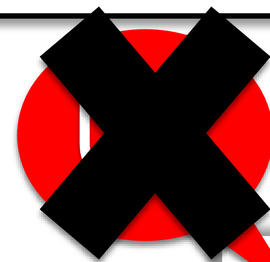
UB | true | false | poison

```
if (freeze(cond))
  while (n > 0)
  { A }
else
  while (n > 0)
  { B }
```

Our Solution

Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior



0

```
while (n > 0) {
  if (cond)
    A
  else
    B
}
```

true | false | poison

```
if (freeze(cond))
  while (n > 0)
  { A }
else
  while (n > 0)
  { B }
```

Summary of Freeze

Compilers can control poison!

- Branching on freeze(poison) => Nondet.
  - Used for Loop Unswitching
- Branching on poison => UB
  - Used for Global Value Numbering

Summary of Freeze

Compilers can control poison!

- Branching on freeze(poison) => Nondet.
  - Used for Loop Unswitching
- Branching on poison => UB
  - Used for Global Value Numbering

Freeze can also fix many other UB-related problems.

Further Example

Hoisting Division

```
// bitwise-or
k = x | 0x1

while (n > 0)
  use(100 / k)
```

```
// bitwise-or
k = x | 0x1
t = 100 / k
while (n > 0)
  use(t)
```

Further Example

Hoisting Division

poison

```
k = x | 0x1

while (n > 0)
  use(100 / k)
```

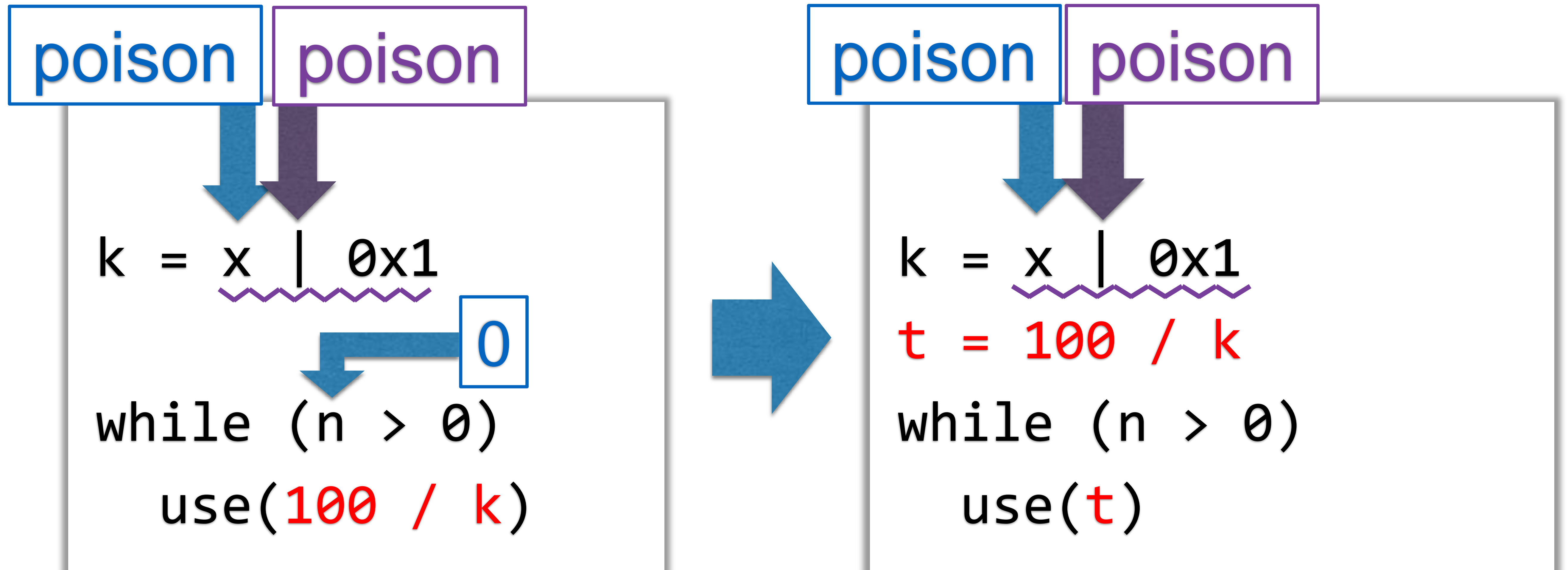
0

poison

```
k = x | 0x1
t = 100 / k
while (n > 0)
  use(t)
```

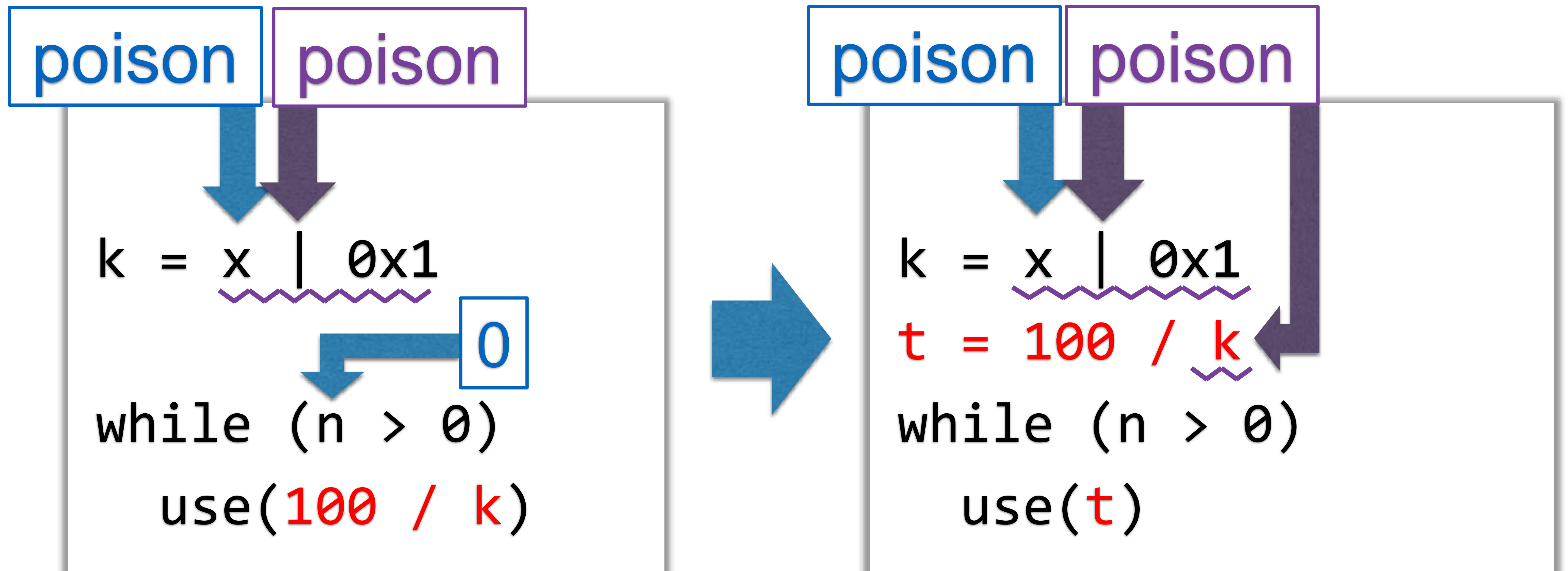
Further Example

Hoisting Division



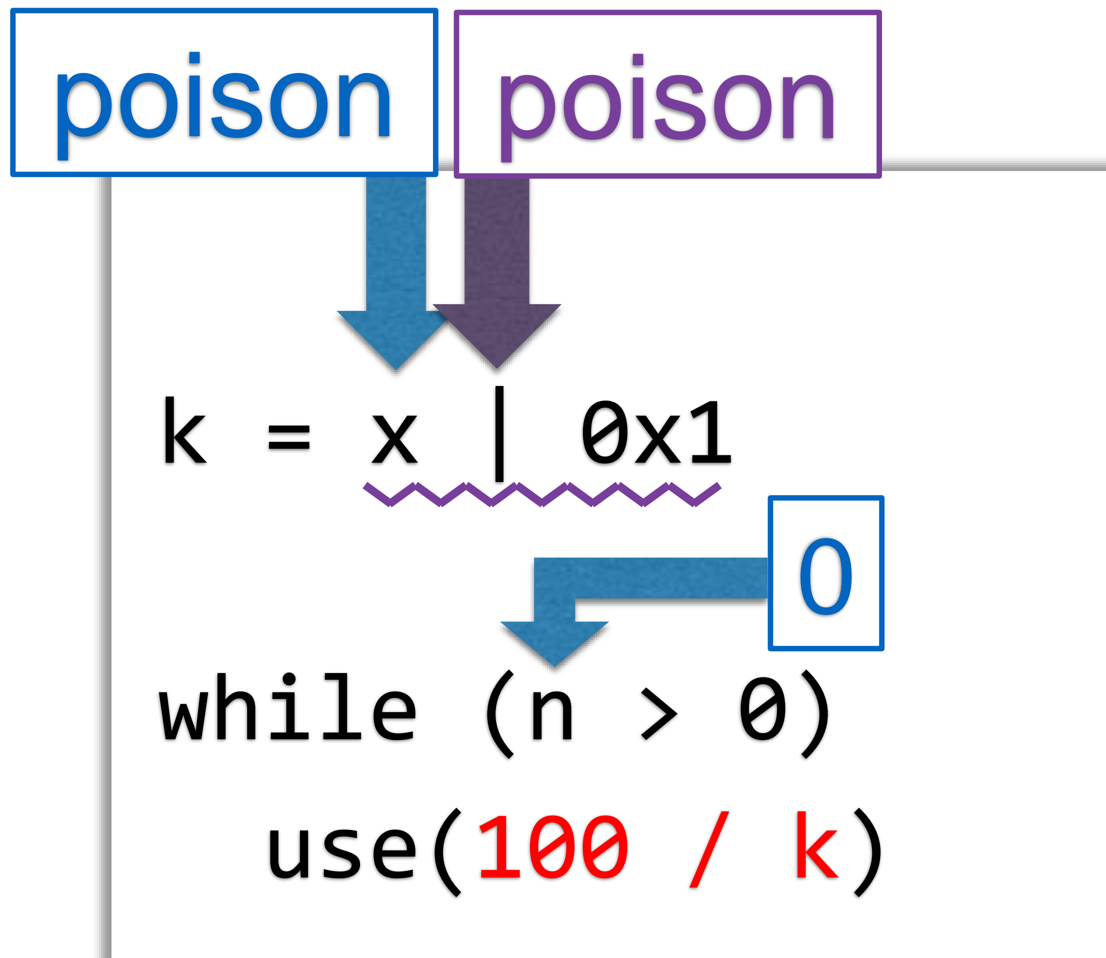
Further Example

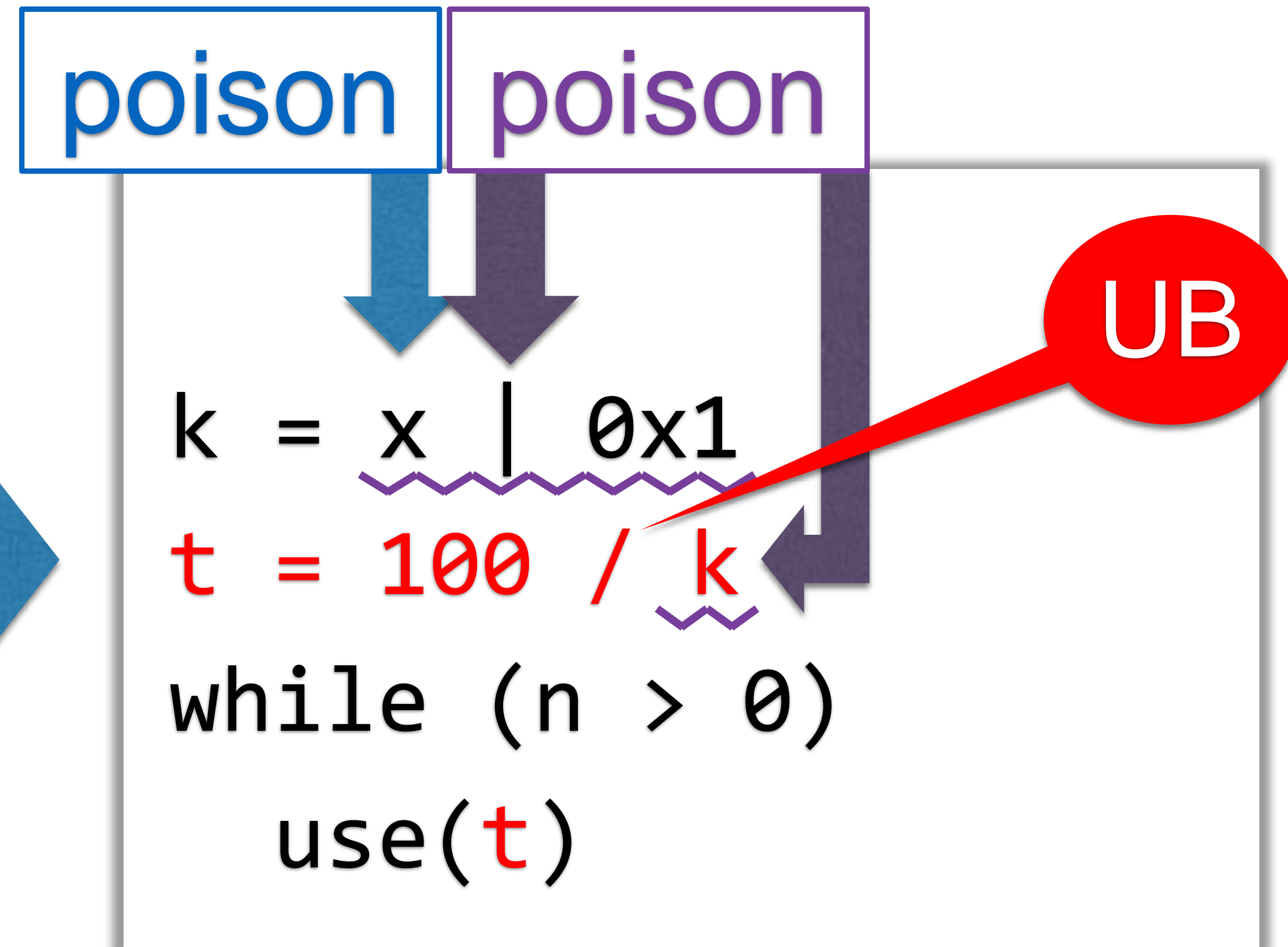
Hoisting Division



Further Example

Hoisting Division

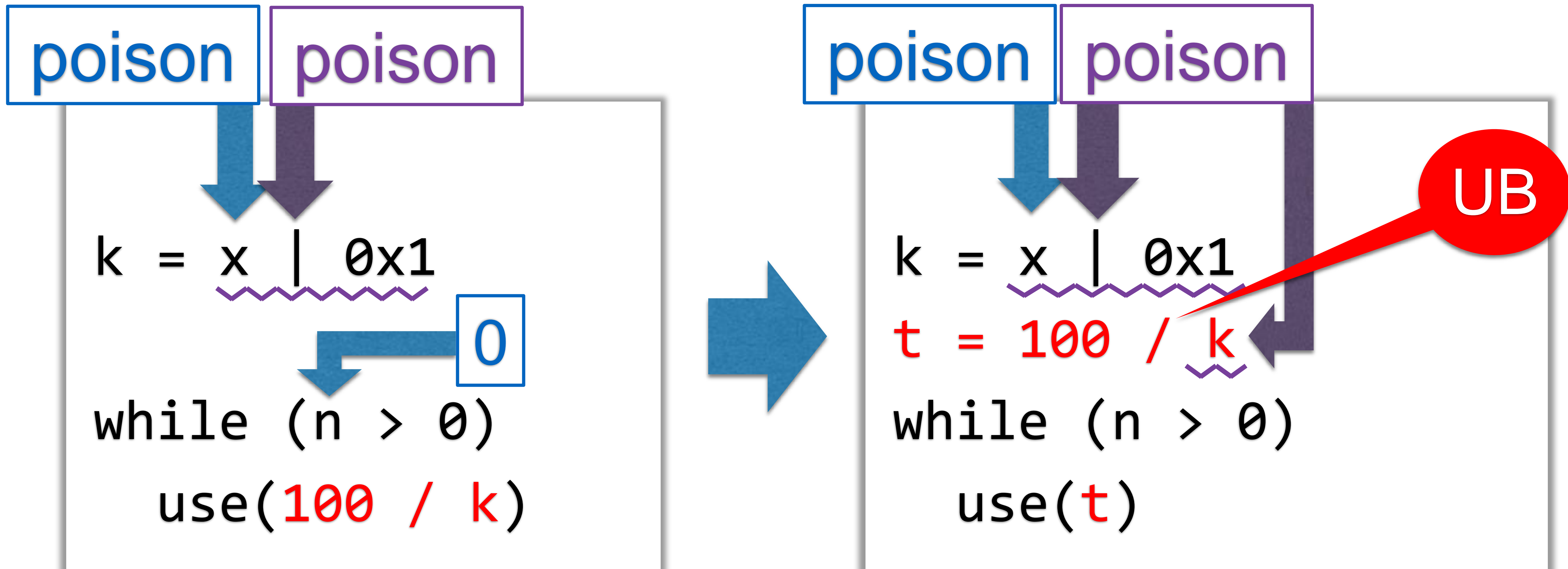




Further Example

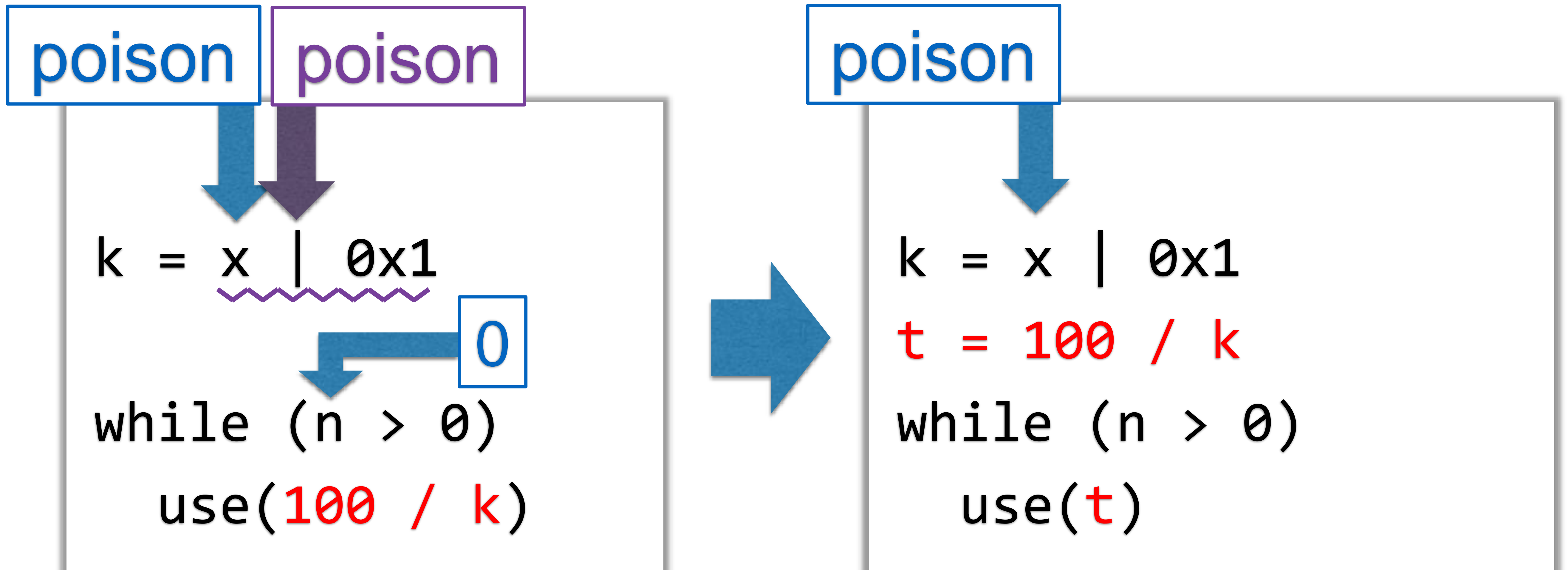
Hoisting Division

LLVM does not currently support it.



Further Example
Hoisting Division

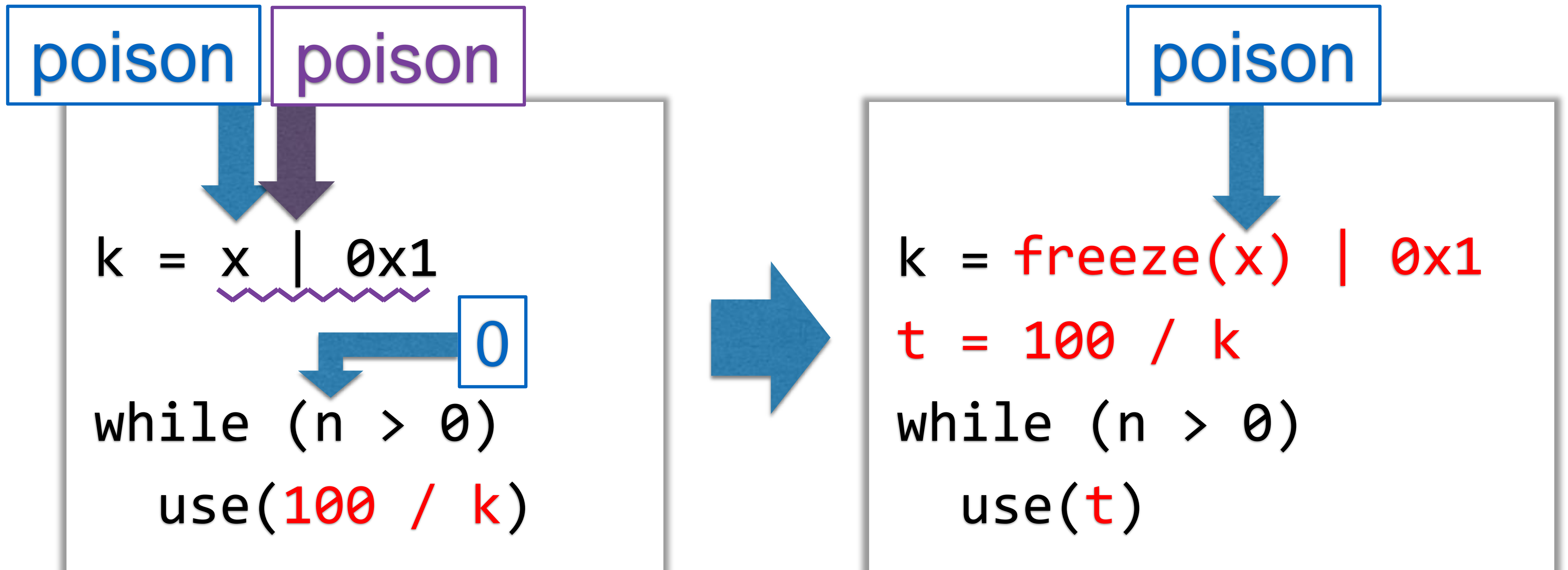
LLVM does not currently support it.



Further Example

Hoisting Division

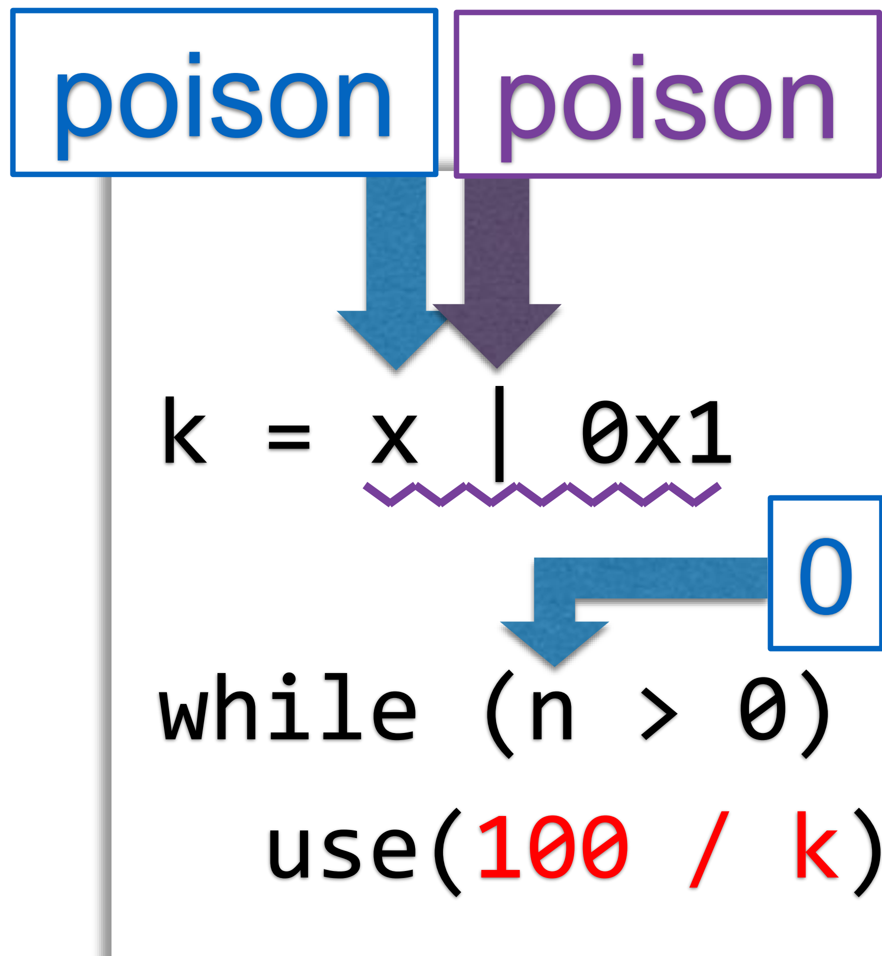
LLVM does not currently support it.

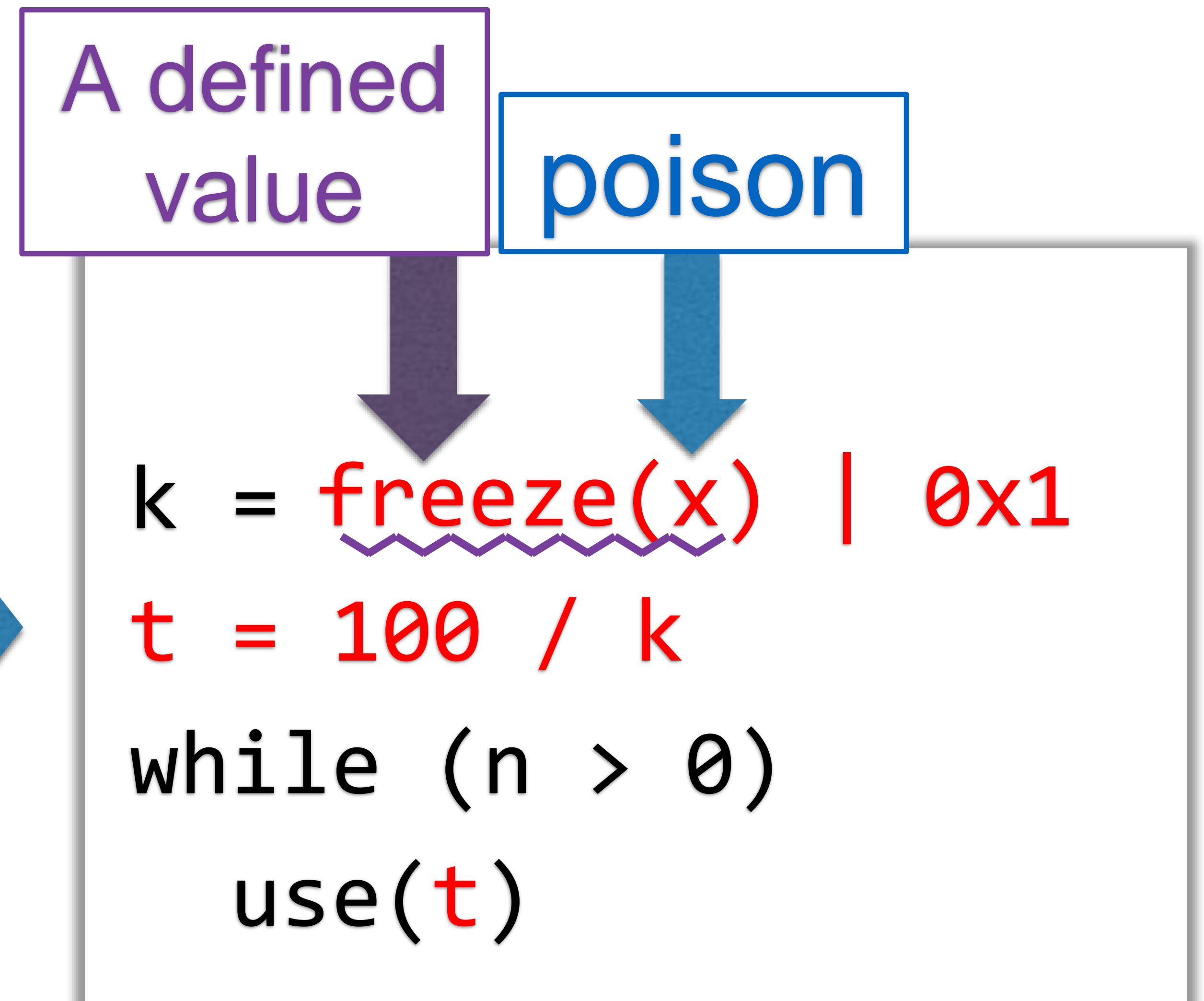


Further Example

Hoisting Division

LLVM does not currently support it.

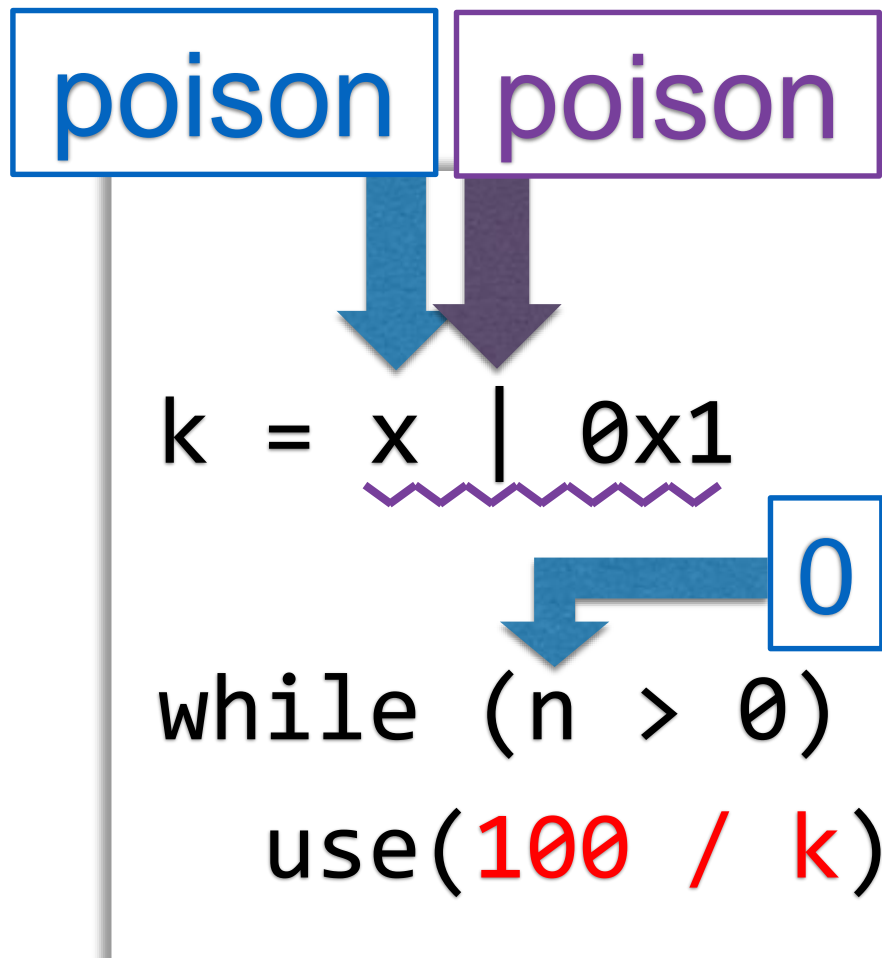


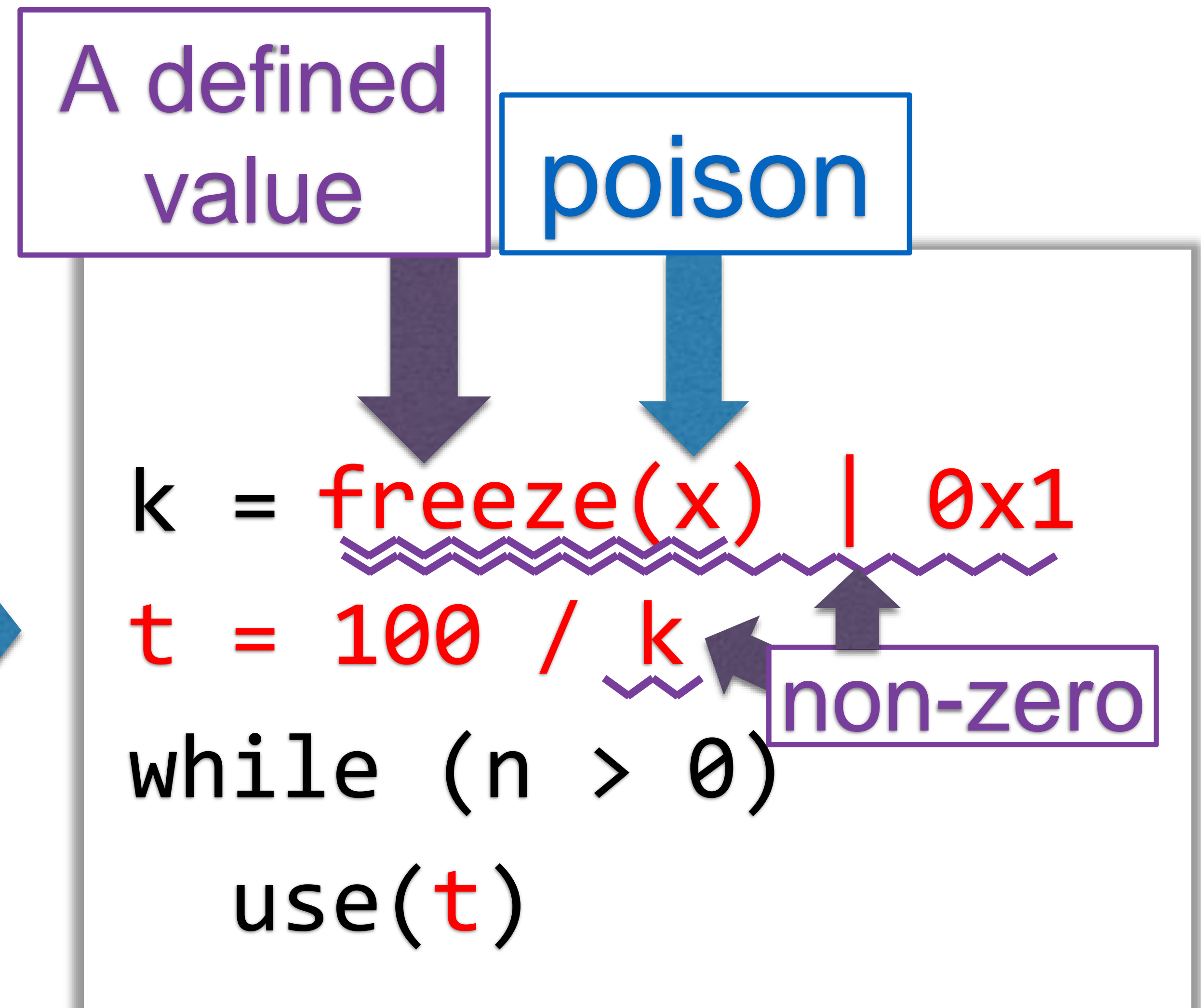


Further Example

Hoisting Division

LLVM does not currently support it.

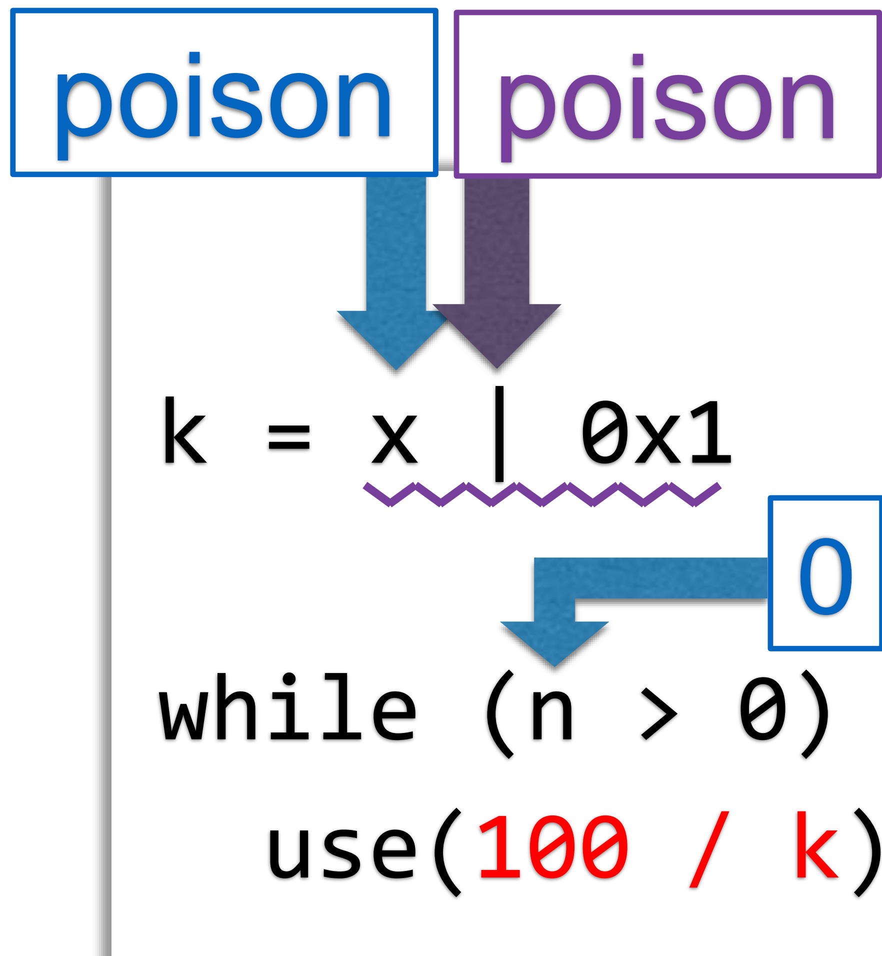


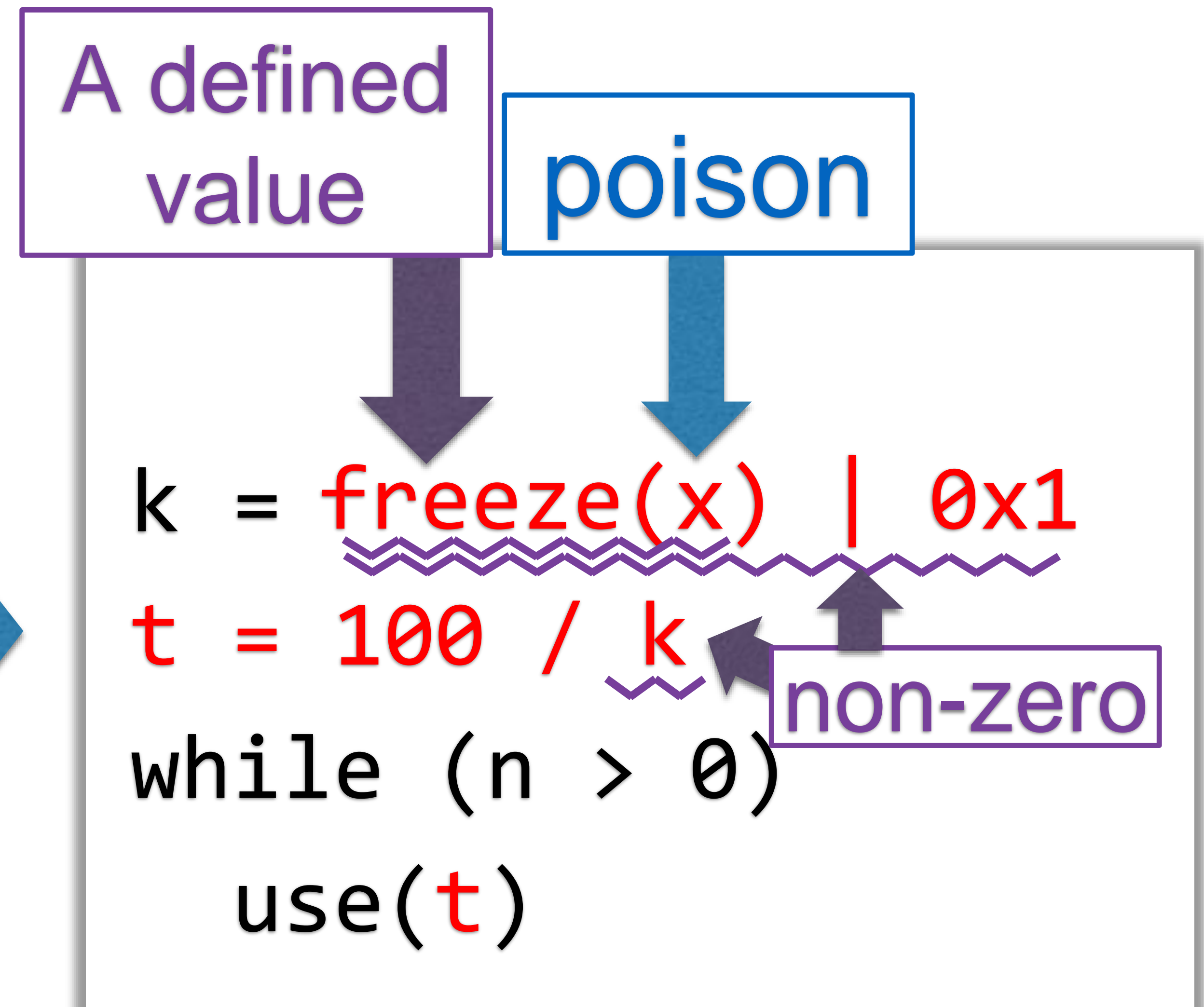


Further Example

Hoisting Division

Freeze can make LLVM support it!





Implementation

- Target: LLVM 4.0 RC 4 (Mar. 2017)
- Add Freeze instruction to LLVM IR
- Bug Fixes Using Freeze
  - Loop Unswitching Optimization
  - C Bitfield Translation to LLVM IR
  - InstCombine Optimizations

* More details are given in the paper

Experiment Results

- Benchmarks (4.6M LOC):
  - SPEC CPU2006
  - LLVM Nightly Test
  - Large Single File Benchmarks
- Compilation Time: ± 1%
- Compilation Memory Usage: Max + 2%
- Generated Code Size: ± 0.5%
- Execution Time: ± 3%

* More details are given in the paper

“Freeze” Can Fix UB Semantics Without Significant Performance Penalty

- Benchmarks (4.6M LOC):
  - SPEC CPU2006
  - LLVM Nightly Test
  - Large Single File Benchmarks
- Compilation Time: $\pm$ 1%
- Compilation Memory Usage: Max + 2%
- Generated Code Size: $\pm$ 0.5%
- Execution Time: $\pm$ 3%

* More details are given in the paper

Summary

- Modern compilers' UB models cannot support some textbook optimizations.
- We propose "freeze" to fix such problems.
- Freeze has little impact on performance.