Undefined Behaviour

Advanced Compiler Techniques

Source:
https://www.youtube.com/playlist?list=PLf3ZkSCyj1tf3rPAkOKY5hUzDrDoekAc7
Videos: 166 - 170

Module 166

Undefined Behaviour Semantics

What is undefined behaviour?

- For some statements (for instance divide by 0), the behaviour is not logically well defined.
- The language designer may choose to define this behaviour (for example, in languages such as Java and Python, an exception may be generated), or they may choose to leave it undefined (for example, C/C++).

What happens if is such behaviour is left undefined?

- The code may behave differently on different machines.
- The exact implementation decides what will happen. The language provides no guarantees about the behaviour.
- Example: The compiler may leave it up to the assembler to handle the undefined behaviour.
- Example: The hardware may crash the machine if such behaviour is triggered.
- Example: The hardware may simply ignore such instructions (effectively replacing the instruction by a nop)

Cost of handling undefined behaviour

- Consider an illegal memory access (for instance, due to an out of range index).
- The language may choose to define such behaviour by adding explicit checks and throwing exceptions if they fail. Such checks will add 3-4 instructions to the code for every access.
- Some other languages may say that such cost is not justified as the programmer's are not likely to make such mistakes.
- Allowing undefined behaviour may lead to security vulnerabilities.

Exploiting UB: Overflows

Is this legal? (Hint: Think about overflows)

```
int n = ...;
for(int i = 0; i  < n+1; i++)
    // Some code
```

$\Longrightarrow$

```
int n = ...;
for(int i = 0; i  <= n; i++)
    // Some code
```

Exploiting UB: Overflows

What if n = INT_MAX and we have wrap around semantics?

```
int n = ...;
for(int i = 0; i  < INT_MAX+1; i++)
    // Some code
```

$\longrightarrow$

```
int n = ...;
for(int i = 0; i  <= INT_MAX; i++)
    // Some code
```

In such a case, the transformation is illegal as both the pieces of code do different things.

Exploiting UB: Overflows

What if signed overflow was UB?

```
int n = ...;
for(int i = 0; i  < n+1; i++)
    // Some code
```

$\longrightarrow$

```
int n = ...;
for(int i = 0; i  <= n; i++)
    // Some code
```

In this case, this is a perfectly legal transformation as the compiler no longer specifies the correct behaviour.

UB allows for optimizations!

Exploiting UB: Type Aliasing

Legal? What if b[i] aliases with a?

```
void foo(int *a, float *b)
{
    for(int i = 0; i < N; i++)
        b[i] = *a;
}
```

→

```
void foo(int *a, float *b)
{
    int local_a = *a;
    for(int i = 0; i < N; i++)
        b[i] = local_a;
}
```

Exploiting UB: Type Aliasing

Legal if we assume type based strict aliasing.

```
void foo(int *a, float *b)
{
    for(int i = 0; i < N; i++)
        b[i] = *a;
}
```

→

```
void foo(int *a, float *b)
{
    int local_a = *a;
    for(int i = 0; i < N; i++)
        b[i] = local_a;
}
```

Exploiting UB: Based Variables

Legal? What if n >= 257 and A[256] aliases with B[0]?

```
int A[256];
int B[1];

void foo(int n)
{
    int *p = A;
    for(int i = 0; i < n; i++, p++)
        *p += b[0];
}
```

```
int A[256];
int B[1];

void foo(int n)
{
    int *p = A;
    int local_b = b[0];
    for(int i = 0; i < n; i++, p++)
        *p += local_b;
}
```

Exploiting UB: Based Variables

Legal if p is based on A, i.e. p never goes out of A's allocated memory region.

```
int A[256];
int B[1];

void foo(int n)
{
    int *p = A;
    for(int i = 0; i < n; i++, p++)
        *p += b[0];
}
```

```
int A[256];
int B[1];

void foo(int n)
{
    int *p = A;
    int local_b = b[0];
    for(int i = 0; i < n; i++, p++)
        *p += local_b;
}
```

UB in IR

- Signed overflows in LLVM are UB.
- If the source language is not C/C++, the frontend will add explicit checks in IR while generating code.
- Some instructions can have multiple flavours to handle different UB semantics.

```
x = add y, z
x = add nsw y, z    // UB if signed overflow
x = add nuw y, z    // UB if unsigned overflow
```

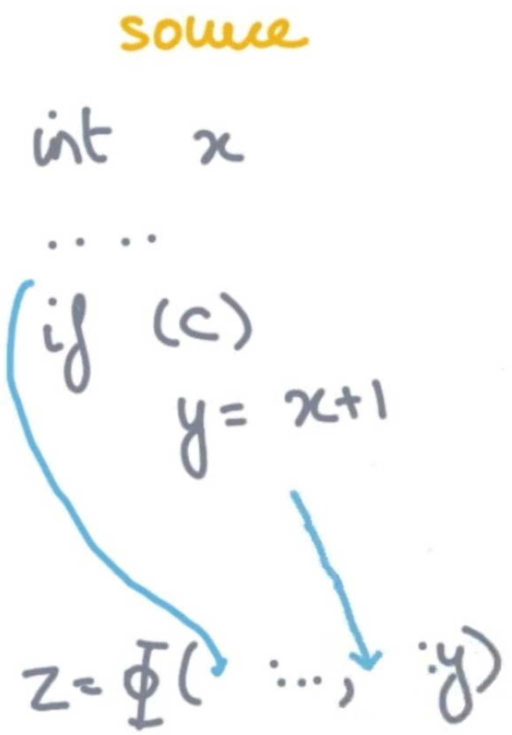
Module 167

Undefined Behaviour in IR

UB in source and target

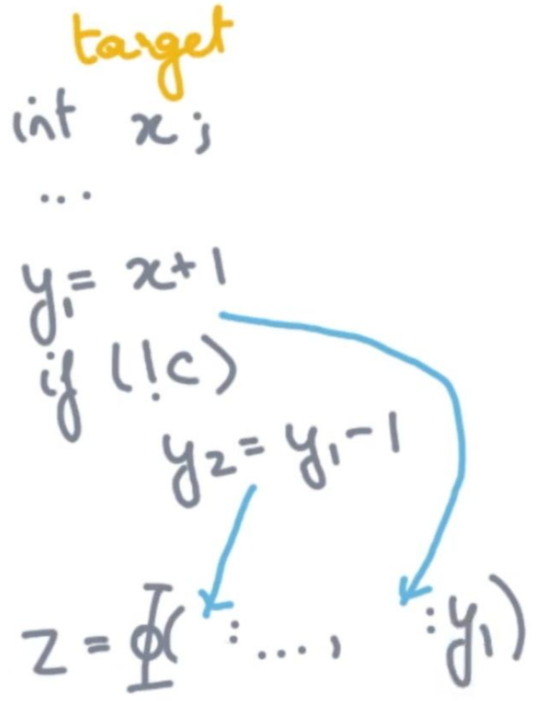
- UB in a source usually provides opportunities for optimizations.
- UB in target may cause issues as from the correctness perspective, the compiler cannot *add* more non-determinism.

When does UB reduce optimization opportunities?



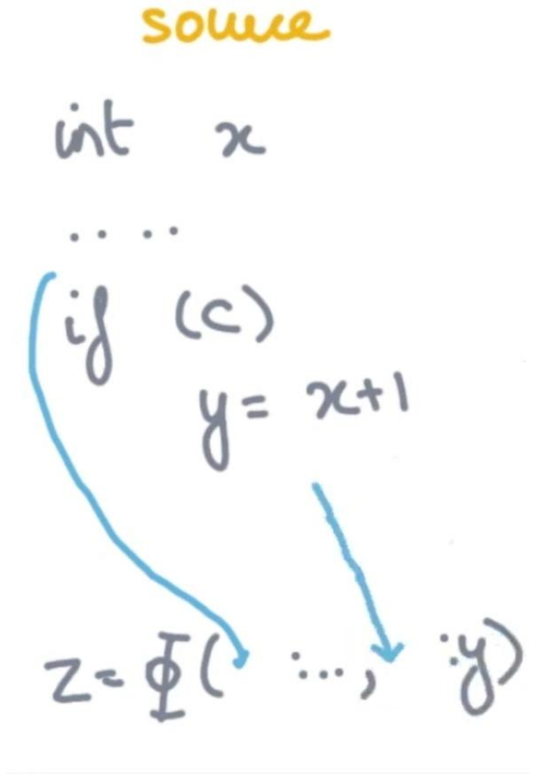
```
int x
... ...
if (c)
    y = x+1
z = Φ( : ..., :y)
```

Valid?

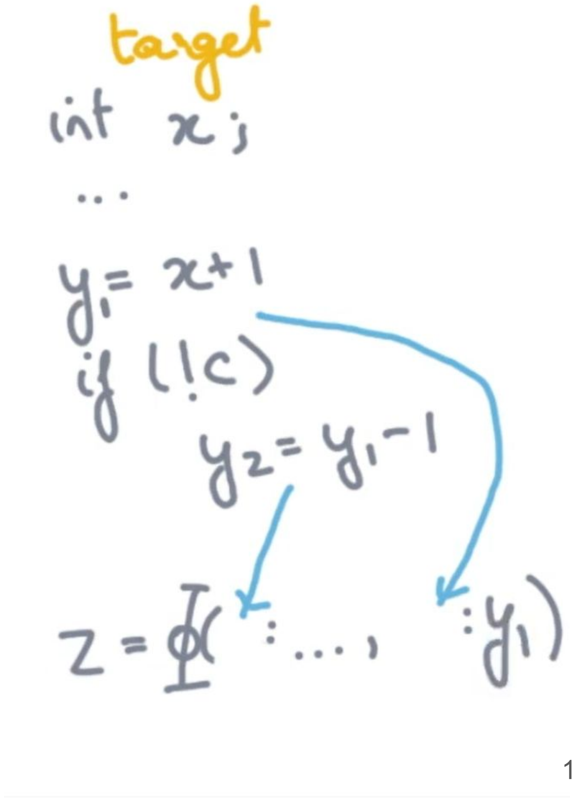


```
int x;
...
y1 = x+1
if (!c)
    y2 = y1-1
z = Φ( : ..., :y1)
```

When does UB reduce optimization opportunities?



What if x = INT_MAX and overflow is UB?
If c = false, left never triggers UB but right always does.



UB may limit optimizations

source

```
int x;
..
while (...) {
    ...
    y = x+1
    ...  // use(y)
}
```

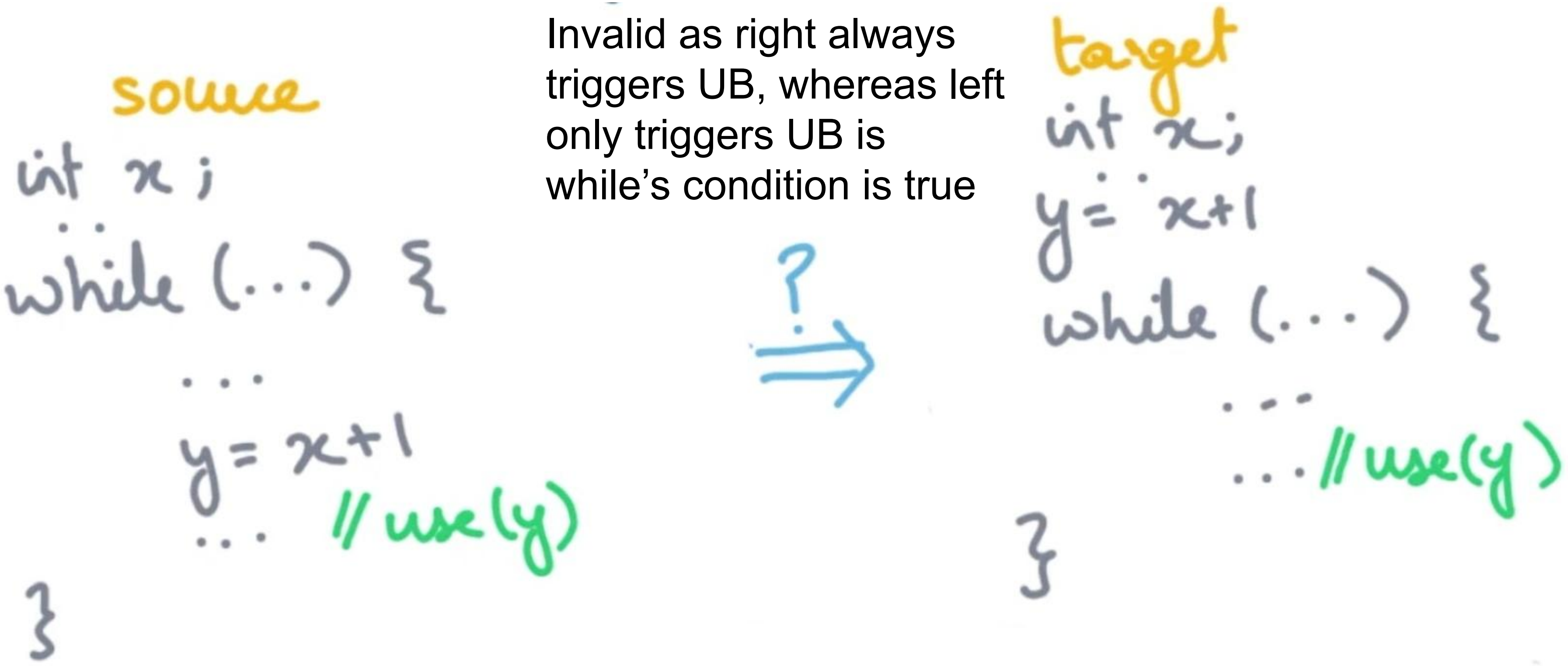
$$\overset{?}{\Longrightarrow}$$

target

```
int x;
..
y = x+1
while (...) {
    ...
    ... // use(y)
}
```

UB may limit optimizations

Invalid as right always triggers UB, whereas left only triggers UB is while’s condition is true



```
int x;
..
while (...) {
    ...
    y = x+1
    ...  // use(y)
}
```

```
int x;
..
y = x+1
while (...) {
    ...
    ... // use(y)
}
```

UB in source and target

- UB in source enables optimizations.
- UB in targets hinders optimizations.
- Ideally, the source has lots of UB and the target has no UB.
- IR designer has to find a middle ground.

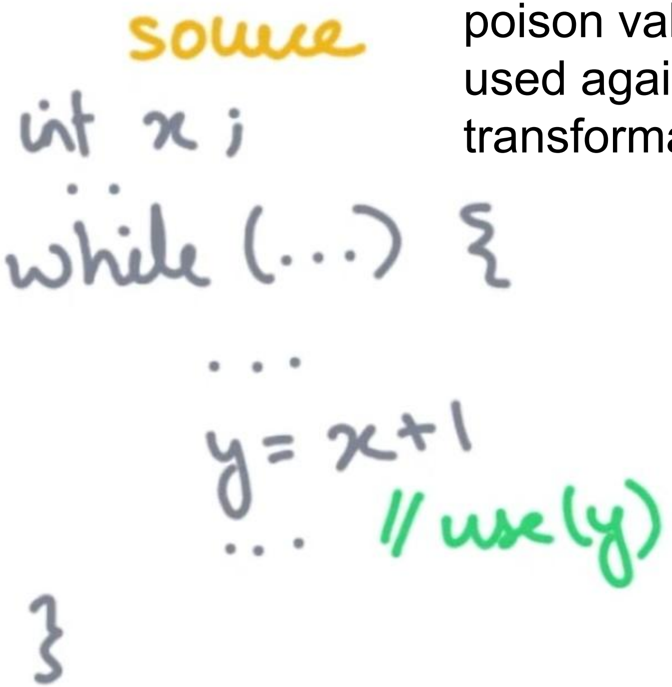
Module 168

Poison Values

What are poison value semantics?

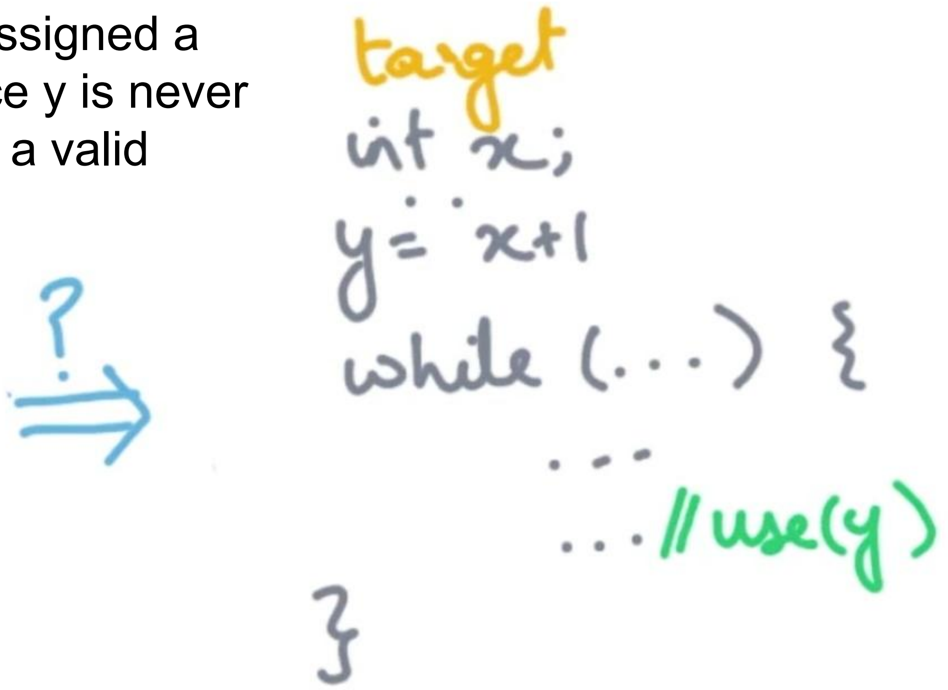
- If a program as UB, anything can happen.
- Out all such possibilities, we pick the possibility of generating an error. This is called a poison value.
- Example: If x = a + b and the addition overflows, instead of classifying this statement as UB, we simply assign an error variable to x.

How does poison value help?



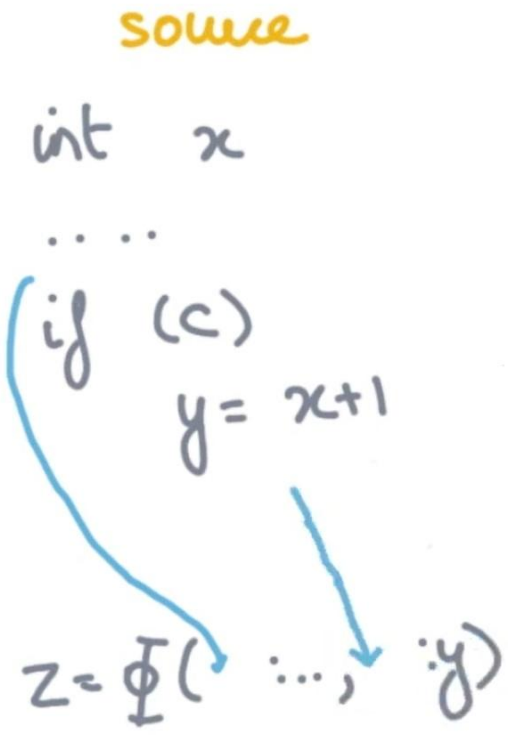
```
source
int x ;
..
while (...) {
    ...
    y = x+1
    ... //use(y)
}
```

In this case, y is assigned a poison value. Since y is never used again, this is a valid transformation



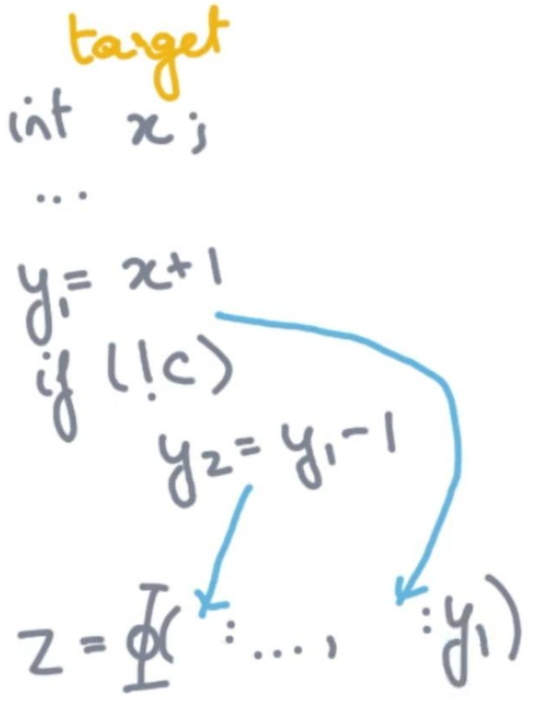
```
target
int x;
..
y = x+1
while (...) {
    ...
    ... //use(y)
}
```

How does poison value help?



```
int x
.. ..
if (c)
    y = x+1
z = Φ( :..., :y)
```

Under poison value overflow semantics, this is a valid transformation. If c is false, then the target never uses y_1.



```
int x;
...
y1 = x+1
if (!c)
    y2 = y1-1
z = Φ( :..., :y1)
```

Operations on poison value

source

```
int x;
...
while (...) {
    ..
    y = x+1;
    z = y/2;
    ..
}
```

Option 1: any op on a poison value triggers UB

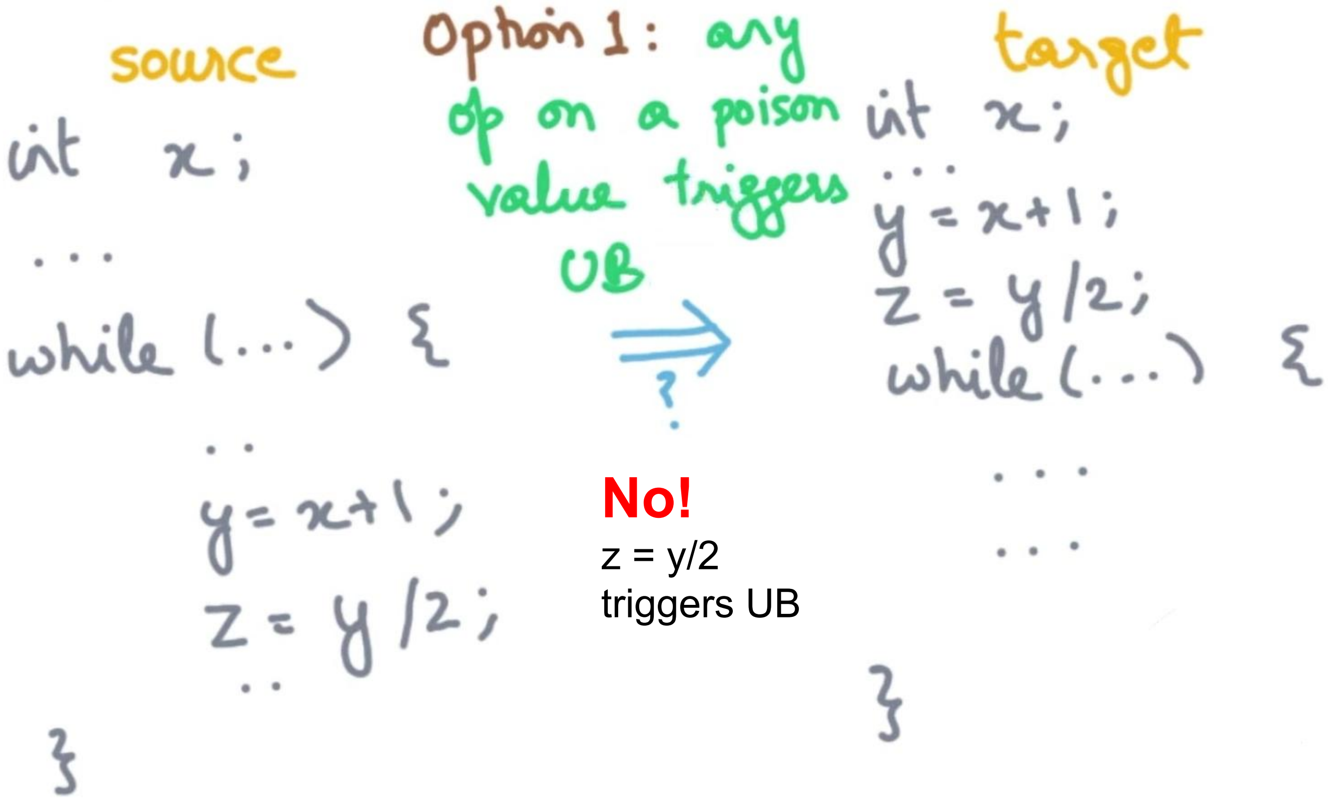
$\overset{}{\underset{?}{\Rightarrow}}$

target

```
int x;
...
y = x+1;
z = y/2;
while (...) {
    ...
    ...
}
```

Legal?

Operations on poison value: Option 1



source

```
int  x;
...
while (...) {
    ..
    y = x+1;
    z = y/2;
    ..
}
```

Option 1: any op on a poison value triggers UB

$\overset{?}{\Longrightarrow}$

target

```
int  x;
...
y = x+1;
z = y/2;
while (...)  {
    ...
    ...
}
```

No!
z = y/2
triggers UB

Operations on poison value: Option 2

source

```
int x;
...
while (...) {
    ..
    y = x+1;
    z = y/2;
    ..
}
```

Option 2: any op on a poison value results in a poison value

$\underset{\text{ok}}{\Longrightarrow}$

target

```
int x;
...
y = x+1;
z = y/2;
while (...) {
    . .
    ...
}
```

Consider x==INT_MAX and loop never taken

target returns poison in z which is never used

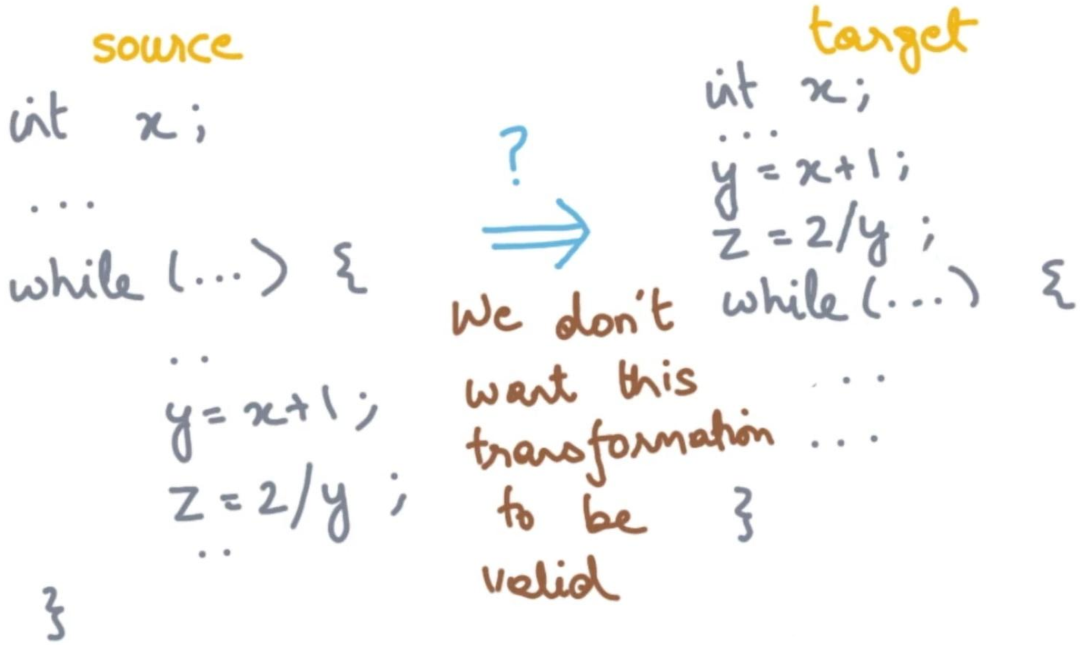
Operations on poison value

- Option 2 is preferable as it allows for more optimizations.

(x_1 op x_2) may generate a poison value if *any* of the arguments is a poison value. A poisoned value poisons the rest of the computation.

Should we allow this transformation?

What if the hardware is designed to terminate the program on divide-by-zero? In this case the target will always get terminated. Such a transform is NOT sound.



```
int   x;
...
while (...) {
      ..
      y = x+1;
      z = 2/y ;
      ..
}
```

```
int  x;
...
y = x+1;
z = 2/y ;
while (...) {
        ...
        ...
}
```

We don't want this transformation to be valid

Why did this happen?

- Divide by zero is more dangerous than other transforms. If we hoist the computation out of the loop, the program will surely terminate (due to hardware semantics).
- Therefore in some cases, operations on poison values can result in UB.
- Ops which trigger dangerous behaviour (such as termination) should trigger UB.
- Ops which don’t generate dangerous behaviour (they may merely result in incorrect computation), should return poison values.

Poison value semantics

Some ops trigger UB on poison

$y = x / \text{poison}$ triggers UB

$y = \text{poison} / x$ returns poison into y

Module 169

Poison Value Operational Semantics

Guiding Principle

- If a poison value exists, then the original program had UB. Therefore any choice is valid from the correctness point of view.
- However, we want to choose a behaviour which maximize optimization opportunities.

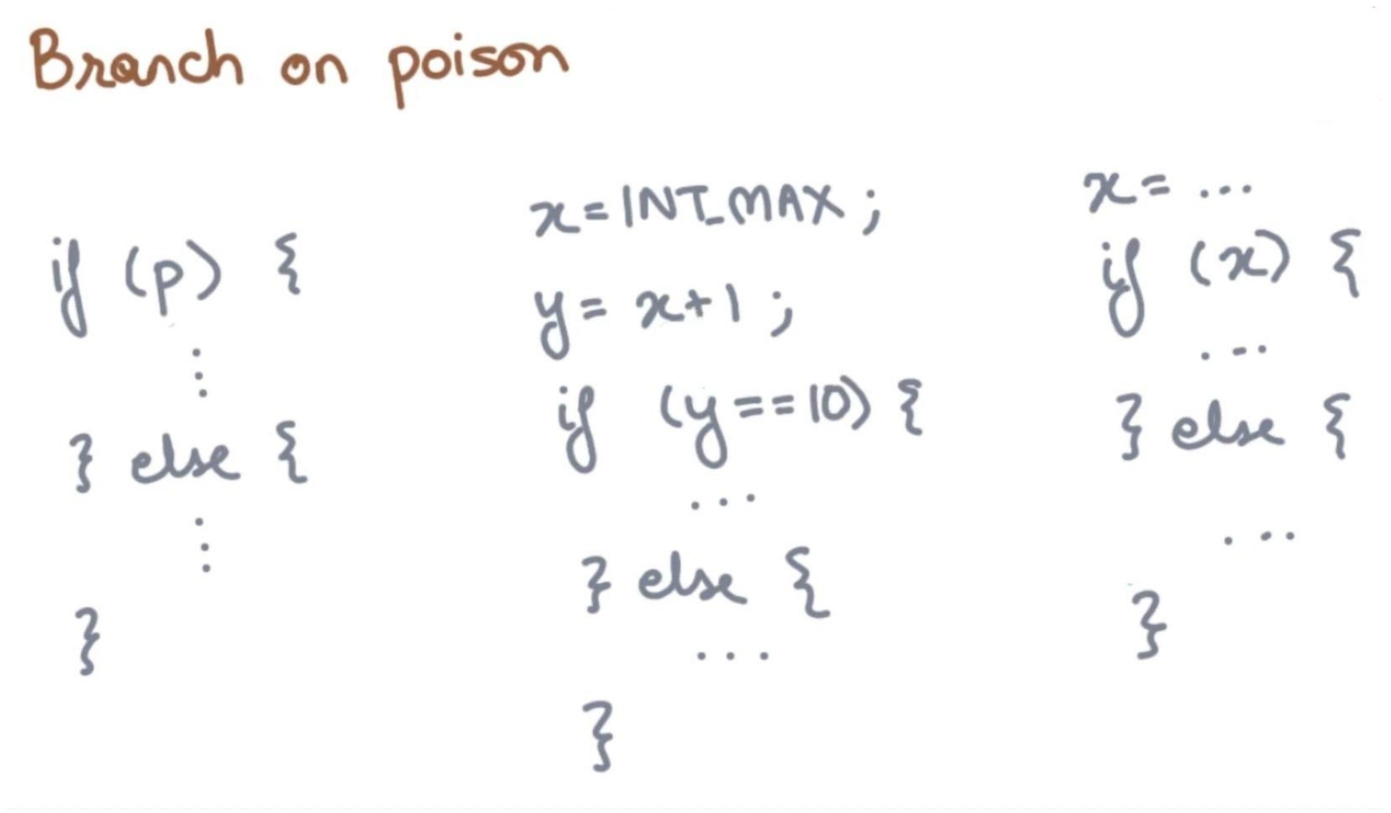
Poison Value Semantics

```
p = add nsw i32 0x7fffffff, 1

p2 = sub i32 p, 1

p3 = and i32 p, 0

p4 = or i32 p, 1

    ...
```

Poison Value Semantics

- All of these return poison values.
- ANDing poison with 0 could have been defined as 0. However, a poison value is more general than 0. Moreover, a poison value can always be converted to 0.
- **Poison value is more flexible.**
- As long as the operation does not generate a dangerous behaviour, returning a poison value maximizes optimization opportunities.

Branch semantics in the presence of poison values



Branch on poison

```
if (p) {
    :
} else {
    :
}
```

```
x=INT_MAX;
y= x+1;
 if  (y==10) {
     ...
 } else {
     ...
 }
```

```
x= ...
if (x) {
   ...
} else {
    ...
}
```

Branch semantics in the presence of poison values

- We can non-deterministically choose one of the branch. However, this can make the rest of the analysis difficult.
- We can make every instruction in both the branches nops. This may not always make logical sense.
- We may use information from other passes (if available) to make a decision. Example: if the compiler can prove that x is either 0 or poison, then always picking the else branch may be optimal.
- There is no clear choice. Hence, **LLVM defines such cases as UB**.

The compiler marks such branches as unreachable

Branch on poison

Easy option: Branch on poison causes UB

```
if (p) {
    :
} else {
    :
}
```

$\overset{\text{ok}}{\Longrightarrow}$ unreachable

When is this sub-optimal? Example 1

Branch on poison

Easy option: Branch on poison causes UB

Consequences?

```
if (a) {
    if (b) {
        ...
    }
}
```

$\not\Rightarrow$

not equal if b is poison and a is false

```
if (a && b) {
    ....
}
```

When is this sub-optimal? Example 2

Branch on poison

Easy option: Branch on poiso

causes UB

Consequences?

Another Example

Loop Unswitching

```
while (c)  {
      if (d) s1;
      else  s2;
}
```

$\not\Rightarrow$ if d = poison and c = false

```
if (d) {
   while(c) s1;
else {
   while (c) s2;
}
```

Memory accesses with poison values

- If the data is poison but the address is not, then store poison value in the memory. Reading from the same location will then return a poison value.

```
*x = p; // Store a poison value
y = *x; // y now has a poison value
```

Memory accesses with poison values

- If the address is poison, trigger UB.

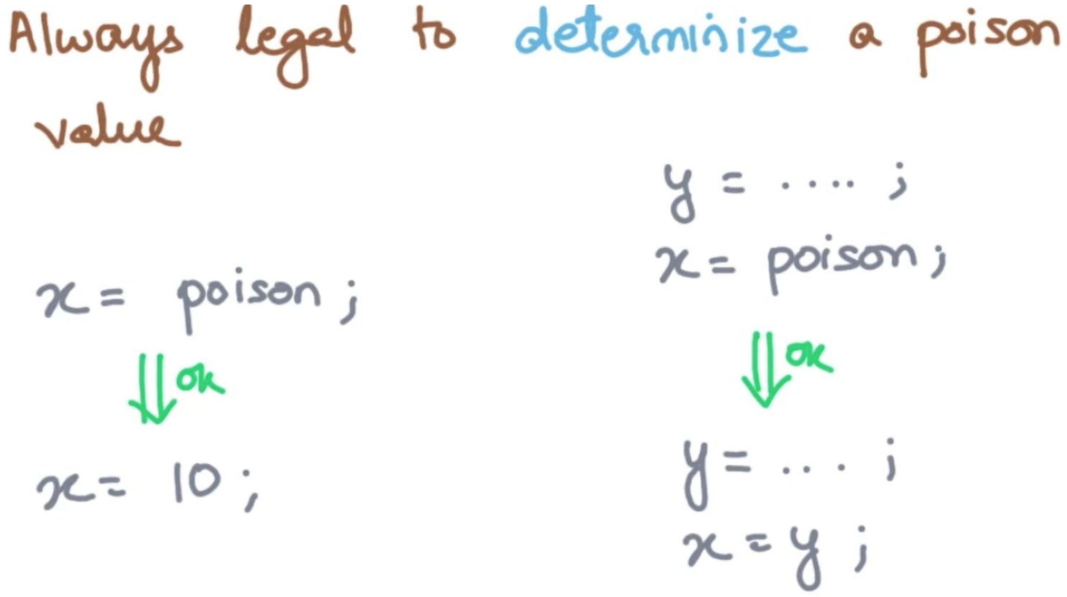
```
*p = x; // UB
y = *p; // UB
```

Poison values and cost of UB

- Everytime we define a poison value operation semantic as UB, we reduce the optimization opportunities.
- For example, if an instruction triggers UB, it cannot be hoisted out of the loop.
- Some opcodes have constrained motion (for example division by poison), whereas others have unconstrained motion.

Converting poison values to deterministic values

- From a correctness standpoint, it is valid to determinize poison values.
- However, it reduces the opportunities for optimizations. Therefore, compiler typically avoid such operations.



Always legal to determinize a poison value

```
x = poison;
    ⇓ ok
x = 10;
```

```
y = .... ;
x = poison;
    ⇓ ok
y = ... ;
x = y ;
```

Module 170

Immediate v/s Deferred UB

Immediate vs Deferred UB

- In some cases LLVM will immediately generate UB, whereas in other cases, it generates a poison value.

Example	C	LLVM
int y = INT_MAX + 1;	Immediate UB	Poison Value
int y = x / 0;	Immediate UB	Immediate UB
int y = x + <poison value>;	-	Poison Value
if(<poison value>) {…} else {...}	Immediate UB	Immediate UB

Immediate vs Deferred UB

- If poison value is never used in a dangerous operation, we can avoid UB.
- Therefore poison values are a form of deferred UB.

Thanks!

- Setu Gupta