

COL874: Advanced Compiler Techniques

Modules 171-175
By: Aditya Senthilnathan
Source: [Link](#)

Undefined Values (Module 171)

Why do we need undef?

- The IR has to cater to multiple languages all of which have different semantics
 - The C Language says that access of uninitialized values is undefined behavior i.e anything can happen
 - Other languages like will just assign constant values like 0 to uninitialized variables or assign arbitrary values

Why do we need undef?

- The poison value is insufficient to handle some language semantics
 - In the below example, if x is poison, then y will also be poison and printing it might result in error or something else due to UB.
 - However, some languages might expect a random value to be printed which cannot be done with poison value

```
int x;  
int y = x*0;  
print(y);
```

The undef value

- As a solution to such cases, the LLVM IR has the undef value which is less non-deterministic than poison
- An undef value represents an **unspecified bit pattern** of its type

```
i8 a = undef;  
// Variable 'a' of type i8 can take values {0,...,255}
```

Transformations with undef

- When transforming programs with non-determinism,
 - The target program can only exhibit a subset of the behaviors of the source program
- This condition informally means that we are allowed to go from higher non-determinism to lower non-determinism but not the reverse
- Example

```
a = undef  -->  a = 7
```

Transformations with undef

- Valid Transformations

```
a = b + undef --> a = undef
a = b - undef --> a = undef
a = b ^ undef --> a = undef
a = b < undef --> a = undef
```

Assuming
wrap-around

For any value of variable a in the target program, we can choose a value for $undef$ in the source program such that the two values for a are same. Thus, target program exhibits a subset of behaviors of the source program

- Eg. If a in target is 10 and b is 7, then a in source can be 10 by setting $undef$ to 3

Transformations with undef

- Invalid Transformations

```
a = or i32 b, undef  -/->  a = undef
```

The target program can take values that the source program can never have.
Hence it is invalid

- Eg. If b is 1, then a in source program can only be odd values but a in target can be both odd and even values.

Transformations with undef

- However, this transformation is valid

```
a = or i32 b, undef --> a = b
```

Source can be reduced to target by setting undef in source to 0

- The examples so far serve to illustrate that we can reduce the set of behaviors of the program from source to target but not increase it

The undef value

- The undef value also has the property that each time it is observed, it can return a different bit pattern

Invalid

```
a = xor i32 undef, undef  -/->  a = 0
```

Valid

```
a = xor i32 undef, undef  -->  a = undef
```

- While xor of a value with itself is normally 0, here the first transformation is invalid and the second valid, because the two undef values need not specify the same value and could be any two values

The undef value

- The undef value also has the property that each time it is observed, it can return a different bit pattern

Invalid

```
a = undef
b = a + a
c = b % 2
assert (c == 0)
```

Valid

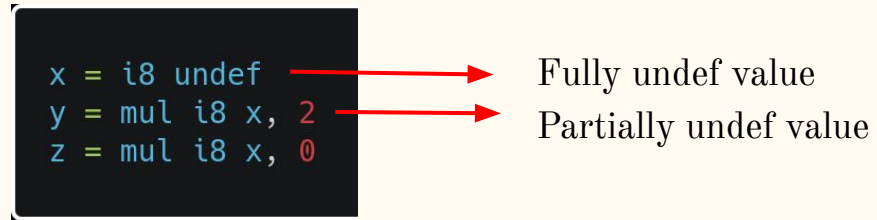
```
a = undef
b = a * 2
c = b % 2
assert (c == 0)
```

Undef Operational Semantics (Module 172)



Semantics of undef

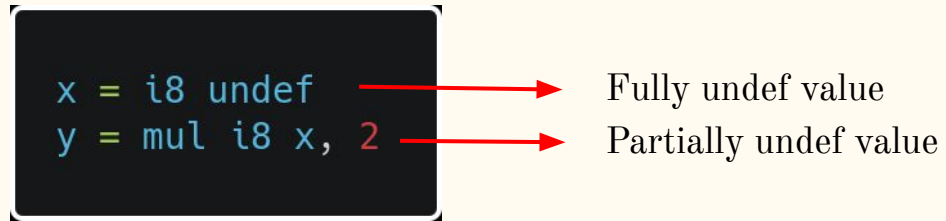
- Undef is non-deterministic in that it represents a set of values from which it returns a value each time it is observed



- Each time `x` is read, any value in $\{0,1,2,\dots,255\}$ can be returned
- Each time `y` is read, any value in $\{0,2,4,\dots,254\}$ can be returned
- Each time `z` is read, only `0` can be returned and thus `z` is deterministic. It can also be thought of as a partial undef value with singleton set of values

Semantics of undef

- Undef is non-deterministic in that it represents a set of values from which it returns a value each time it is observed



- While a fully undef value represents the entire range of permissible values for its type, partially undef values represent only a proper subset of the full range

Semantics of undef

- Branching on undef is Undefined Behavior (like poison)
 - The alternative is to branch non-deterministically but this interferes with the kind of transformations the compiler usually performs such as Dead Code Elimination

Semantics of undef

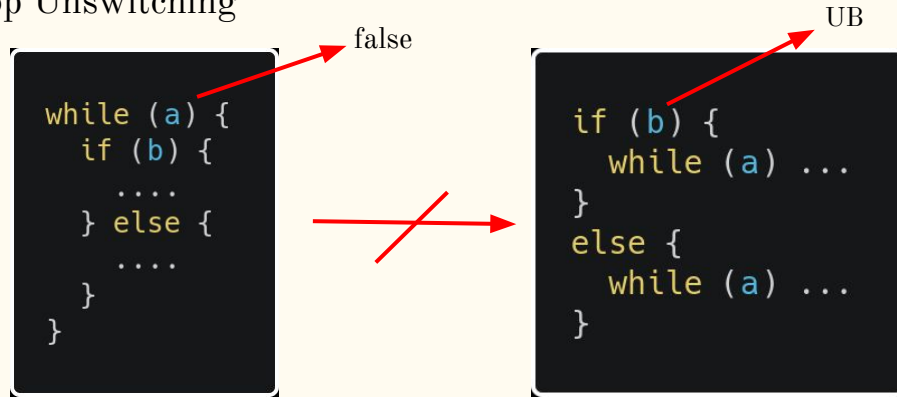
- Branching on undef is Undefined Behavior
 - Implications: Hoisting

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

- Transformation is valid
 - Eg. a = false, b = undef

Semantics of undef

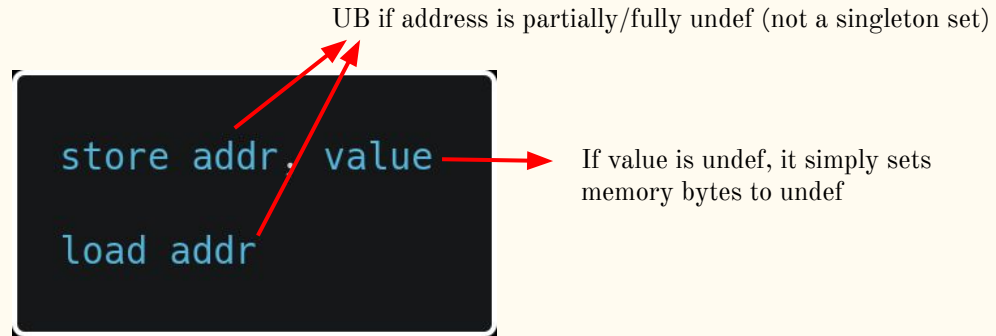
- Branching on undef is Undefined Behavior
 - Implications: Loop Unswitching



- Transformation is invalid
 - Eg. `a = false`, `b = undef`

Semantics of undef

- Memory




- Storing to an undef address is UB, because abstract machine doesn't know where to store
- As a result, hoisting is disabled for load and store instructions when the address could potentially be undef

Semantics of undef

- Division by undef

```
u = partially or fully undef  
b = x/u
```



Triggers UB if 0 is one of the values
u can take

Semantics of undef

- Select instruction

```
select c, a, b
```

- a, b can be undef or poison and according to c, the appropriate value is returned
- If c is undef/poison,
 - Undef - one of a and b is returned non-deterministically according to the set c represents
 - Poison - poison is returned because operations with poison return poison
 - Unlike in branching, UB is not triggered (Because select can only return expressions and is therefore less flexible)

WHYs of Undef semantics (Module 173)

Semantics of undef

- Why is undef allowed to yield different values when observed at different points?
 - Because this relieves the compiler of having to save and restore garbage values in registers, thereby saving register space
 - Furthermore, this obeys “replace all uses with” semantics of the IR which is ingrained in many compiler transformations such as constant propagation

```
x = undef + undef
```

Semantics of undef

- Why is branch on undef and poison UB?
 - The Global Value Numbering transformation is valid only if branch on undef and poison is UB

```
if (x==u)      -->      if (x==u)
  f(x)         f(u)
```

- Eg. If branch on undef is non-deterministically chosen, then source would execute $f(1)$ but target would execute $f(\text{undef})$ which is invalid
- Whereas, if branch on undef is UB, the code becomes unreachable and we can ignore this particular input

Non-determinism of poison and undef

- Relative degree of non-determinism between poison and undef
 - Undef is more deterministic than poison. For example,

```
x = undef * 2  -/->  x = 1
x = poison * 2 -->   x = 1
```

- Since poison is more non-deterministic of the two, poison can be converted to undef but the reverse is invalid

```
poison --> undef
undef -/-> poison
```

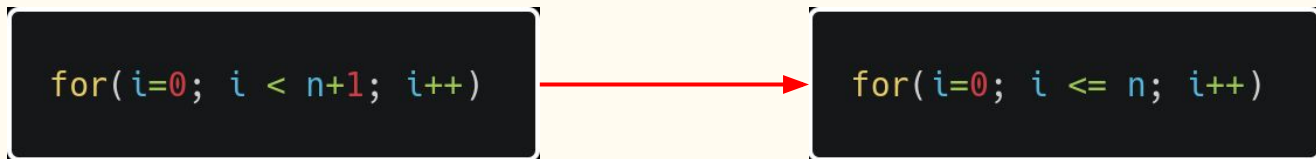

Non Determinism and Optimization Opportunity

- More non-determinism in the source program implies more optimization opportunities. Thus, poison has more optimization opportunities than undef. For example,

```
x = nd * y
if (expensive_op(x)) {
    . . .
}
```

- If nd is poison, the if block is unreachable and can be deleted because it will trigger UB as any operation on poison is poison
- If nd is undef, it's possible that the operation can return a deterministic value and therefore the if block won't trigger UB and can't be optimized

Non Determinism and Optimization Opportunity



- If we assume wrap around semantics, this transformation is invalid because loop in source will never execute whereas the one in target will
- If integer overflow is either poison or undef, this transformation is valid because, then in the source program, UB is triggered by branching on poison or undef for the $n = \text{INT_MAX}$ case and therefore the compiler can ignore this corner case and the transformation is valid for all other cases

Non Determinism and Optimization Opportunity

```
sgt a + x, a  -->  sgt x, 0
```

- If signed integer overflow is poison, this transformation is valid, because $a+x$ will be poison which makes the entire expression poison and poison can be transformed to the target expression
- If signed integer overflow is undef, this transformation is invalid. Consider the case $a=INT_MAX$, $x=1$. In this case, the source will always evaluate to false and target will always evaluate to true

Non Determinism and Optimization Opportunity

- Why have undef if poison is better for optimization?
 - Undef gives us more predictable behavior such as when we print an uninitialized variable
 - Furthermore, it helps us express certain language semantics in the IR which cannot be done with poison

Freeze Opcode in LLVM (Module 174)

Why do we need freeze opcode?

- There are limitations with poison and undef
 - They trigger UB when used for branch condition as a result of which transformations like hoisting, loop unswitching, etc. are invalid
 - Since undef can return different values on different observations, certain transformations such as the following involving strength reduction are disabled

```
a = x / y      -->      a = x/y
b = x % y      -->      b = x - (a*y)
```

Consider the case $x = \text{undef}$, $y = 1$. b in source would be 0 but b in target would be undef

Freeze Opcode

- Freezing a variable
 - Returns the same value if the variable is well-defined/deterministic
 - If the variable is non-deterministic, it picks an arbitrary value out of the set of values that variable could have taken and returns that value thus making the result a well-defined value

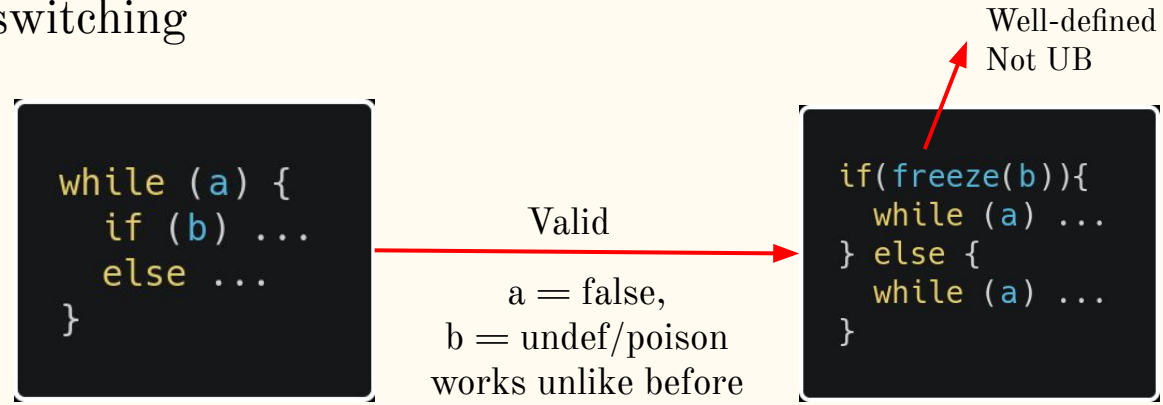
```
x = freeze i8 y
```



Well-defined

Transformations with freeze

- Using the freeze opcode, we can make certain transformations which were previously invalid, valid
- Loop Unswitching



Transformations with freeze

- Select to branching

```
z = select c, x, y
```

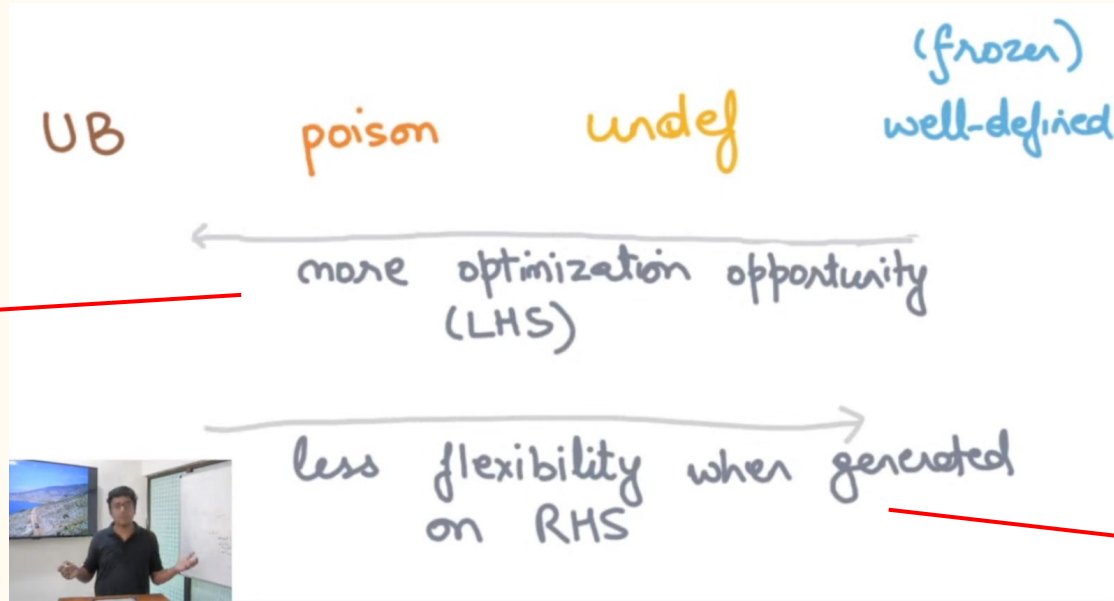
Valid

```
if (freeze(c)) {  
  z = x  
} else {  
  z = y  
}
```

Well-defined
Not UB

Degrees of Non-Determinism

- With undef, poison and freeze, we now have varying degrees of non-determinism in LLVM and the IR has to walk a fine line between them

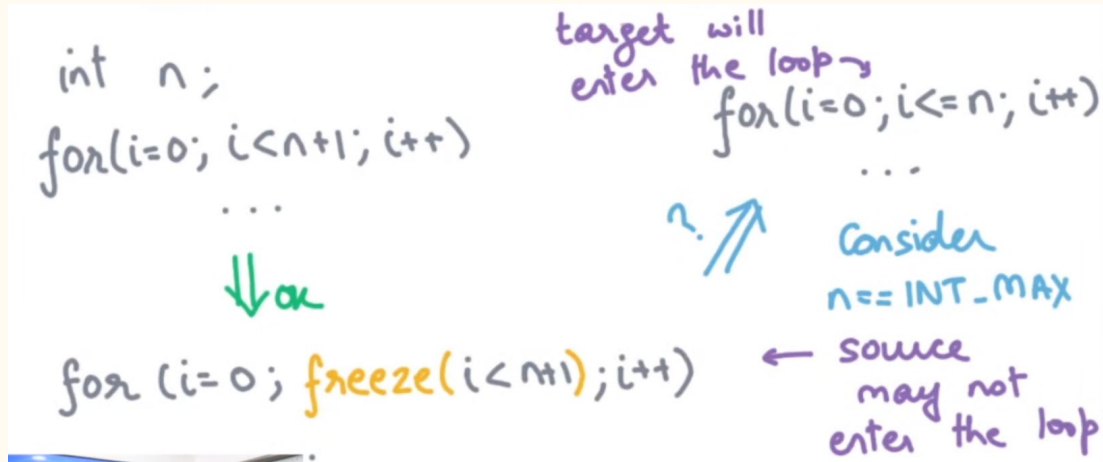


Optimization with Freeze

- When freeze is present on,
 - Target side (RHS): It **enables** optimizations
 - Source side (LHS): It **disables** optimizations (Makes values less non-deterministic, thus reducing optimization opportunity)

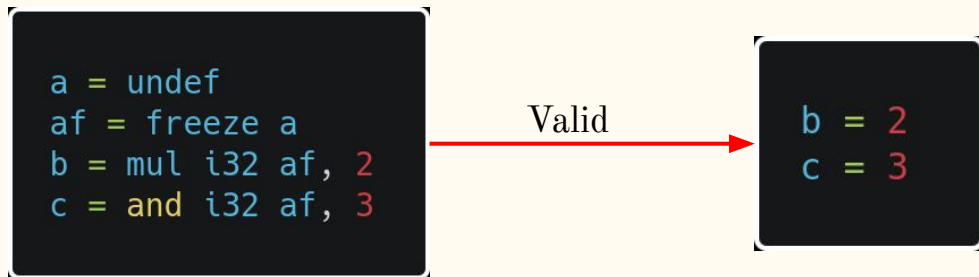
- Due to this nature of freeze, depending on how it is used, it can thwart certain optimizations

Optimization disabled with freeze



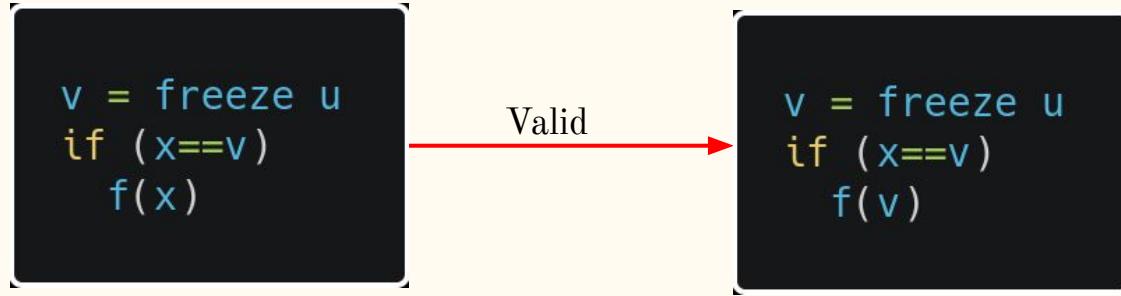
- The transformation is valid because the freeze(poison) can always choose true for the `n==INT_MAX` case
- However, if we had `i < freeze(n+1)`, the transformation is invalid, because `i` would eventually become `INT_MAX` and there is no value that freeze can pick to make the condition true. The source and target will thus diverge

Optimization disabled with freeze



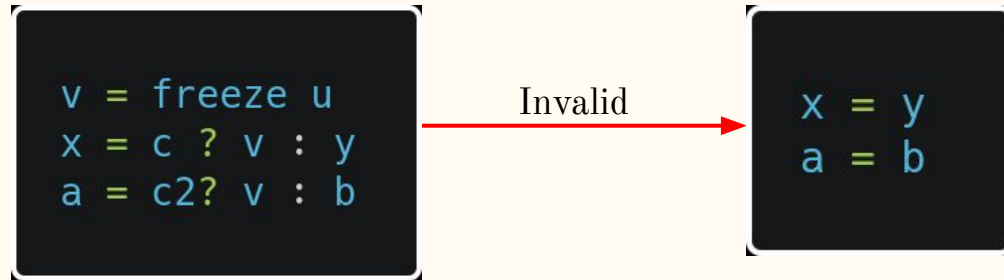
- This transformation is thwarted by freeze because af can't have different valuations for evaluating b and c
- However, the compiler can accordingly reason and figure out that its acceptable to set af to 1 and therefore transform c to 1 in the target program
- This shows that if freeze is recklessly used, transformations are thwarted but with careful reasoning and proper use of freeze, the same transformations can be enabled. However, the catch is that the compiler has to work harder to find the valid transformations

Optimizations with freeze



- This optimization which was previously invalid, is now valid with freeze

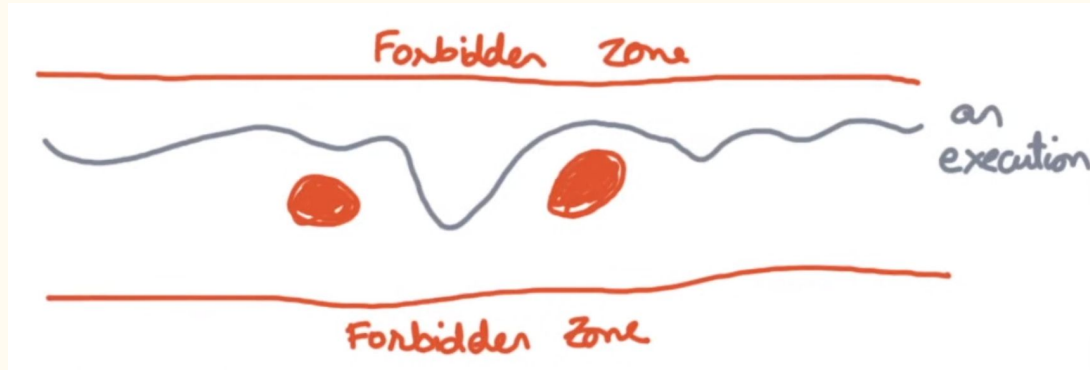
Optimization disabled with freeze



- This optimization is invalid because `v` is a deterministic value and can't be both `y` and `b` if `y` is not equal to `b`

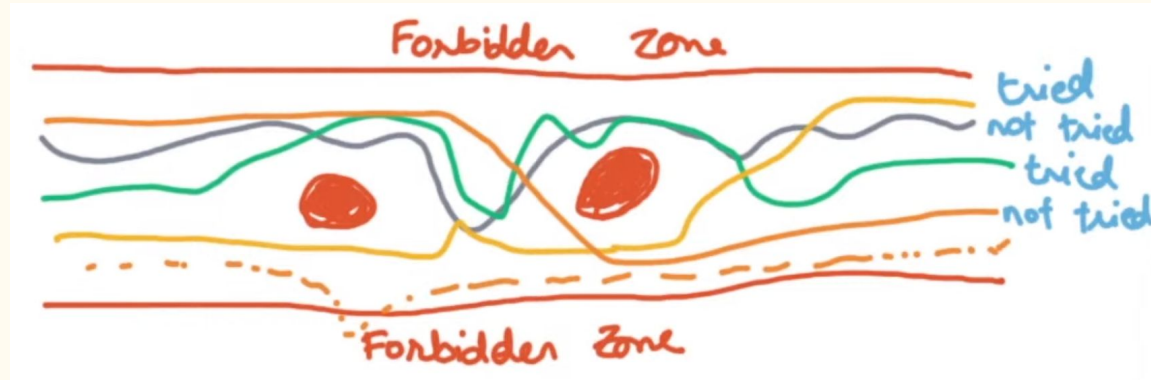
Intro to Static Analysis Approaches (Module 175)

Existing Approaches



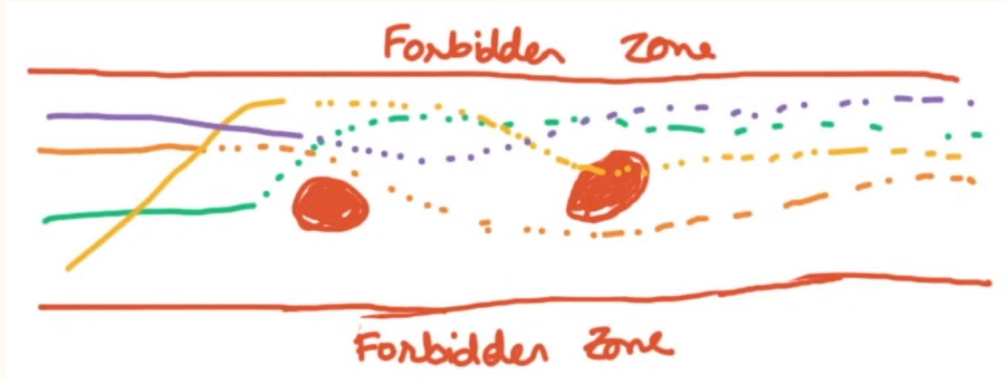
- The forbidden zone consists of cases leading to system failure such as seg fault, etc. that the program execution trajectory should not cross through
- The objective is to ensure that possible trajectories for execution stay away from the forbidden red regions

Testing



- This approach tries out various input instances to check if the trajectory crosses into a forbidden region
- Drawbacks
 - Unchecked input instances can go into forbidden region
- Neither sound nor complete

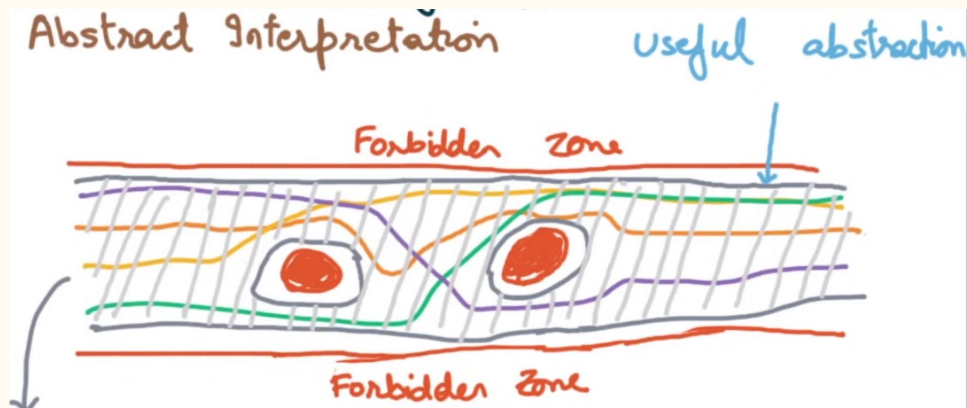
Bounded Model Checking



```
//Check upto n = 10000  
for (i=0; i<n; i++)  
    ...
```

- This approach exhaustively checks all input instances upto a certain bound
- Drawbacks
 - Hard to scale
 - Some input instances beyond the bound might go into forbidden region
- Neither sound nor complete

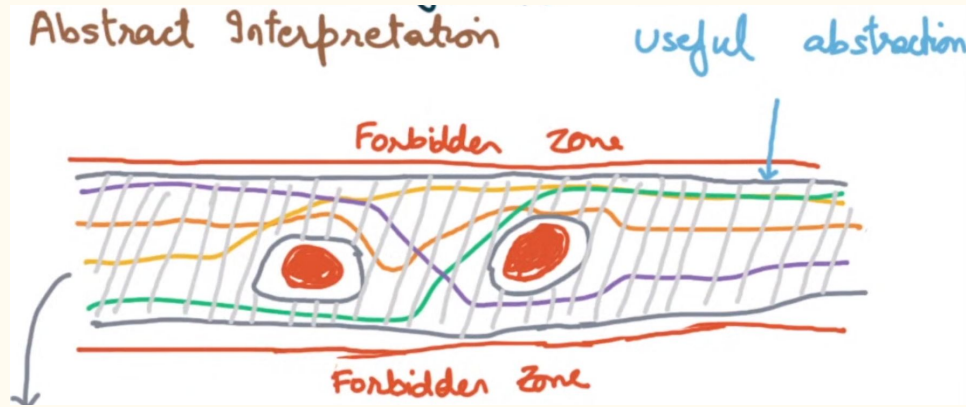
Abstract Interpretation



```
for (i=0; i>=0 and i<n; i+=f())  
    |  
    v  
for (...; i>=0 and i<n; .....)
```

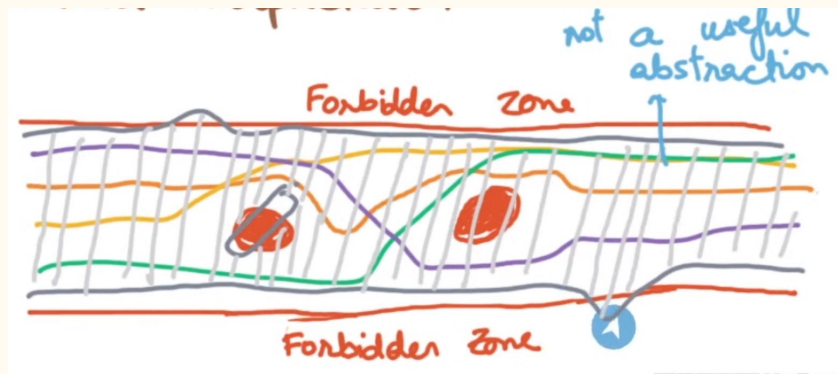
- This approach constructs an abstraction of the actual program and reasons about the abstraction instead of the program
- The abstraction can be thought of as an overapproximate enclosure which encompasses all possible execution trajectories and possibly more

Abstract Interpretation



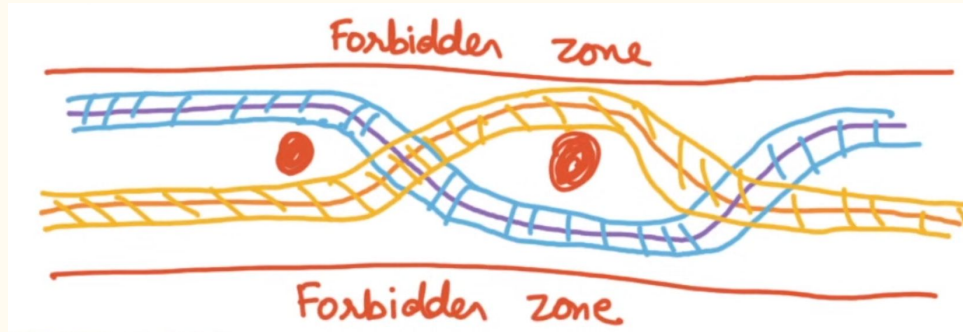
- If a successful overapproximate abstraction is constructed that represents all possible trajectories and it doesn't encompass any forbidden regions, then that is a proof of correctness
- This guarantees soundness i.e the actual program will never enter a forbidden region

Abstract Interpretation



- However, failure to find an abstraction doesn't mean the program will enter a forbidden region
- It might just be that the algorithm's incapable of constructing the correct abstraction.
- Thus, this method is not complete i.e failure to find abstraction doesn't imply that the program is incorrect

Bug Finding Tools



- This approach tries to enhance testing by exploring the neighborhood of the execution trajectories using methods such as fuzzing, etc.
- This approach is neither sound nor complete

Soundness and Completeness

- Soundness is guaranteed when
 - If tool finds proof of no violations then there are no violations in program
- Completeness is guaranteed when
 - If there exist no possible violations then the tool will find a proof

The End