

關於生命週期的一點事兒

The relationship of Lifetimes and DataFlow

Rnic / H.-S. Zheng

Aug 17, 2019 @ COSCUP

Audience

- 讀過 Rust Book
- 想要了解編譯器怎麼看待 Lifetimes
- 對編譯器有那麼一點興趣
- ~~想要輕鬆駕馭 Rust's Lifetimes~~
- ~~想要快快樂樂寫 Rust~~

Outline

- 1. Introduction
 - Example1
 - Basic Lifetimes Concepts
- 2. Borrow Checker
 - Collaborate with Data Flow
 - Example2
 - Datafrog (a datalog engine used in Polonius)

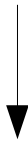
Compilation of Rust

Rust Code

```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
  
println!("{}", *p);
```

Compilation of Rust

Rust Code

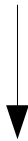


HIR

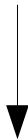
```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
  
println!("{}", *p);
```

Compilation of Rust

Rust Code



HIR



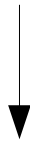
MIR



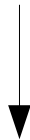
```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
  
println!("{}", *p);
```

Compilation of Rust

Rust Code



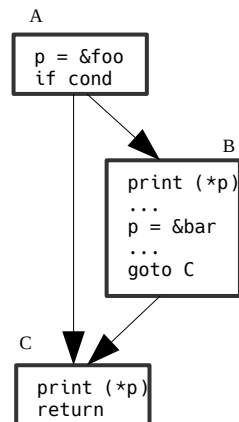
HIR



MIR



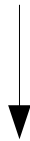
```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
  
println!("{}", *p);
```



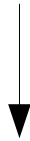
Control Flow Graph

Compilation of Rust

Rust Code



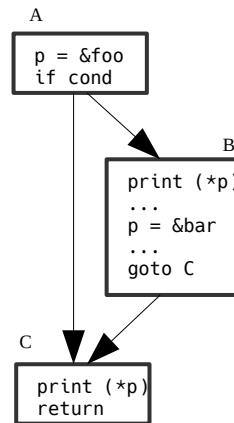
HIR



MIR



```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
println!("{}", *p);
```



Control Flow Graph



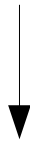
Borrow checker

Theorem: Data Flow Analysis

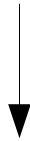
Tool: Datafrog

Compilation of Rust

Rust Code



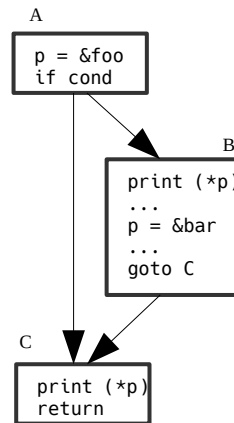
HIR



MIR



```
let foo: T = Foo {};  
let bar: T = Bar {};  
  
let mut p = &foo;  
if cond {  
    println!("{}", *p);  
    ...  
    p = &bar;  
    ...  
}  
println!("{}", *p);
```



Control Flow Graph

Documents

NLL RFC:
2094-nll

Polonius:
an-alias-based-formulation
-of-the-borrow-checker



Borrow checker

Theorem: Data Flow Analysis

Tool: Datafrog

Example

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

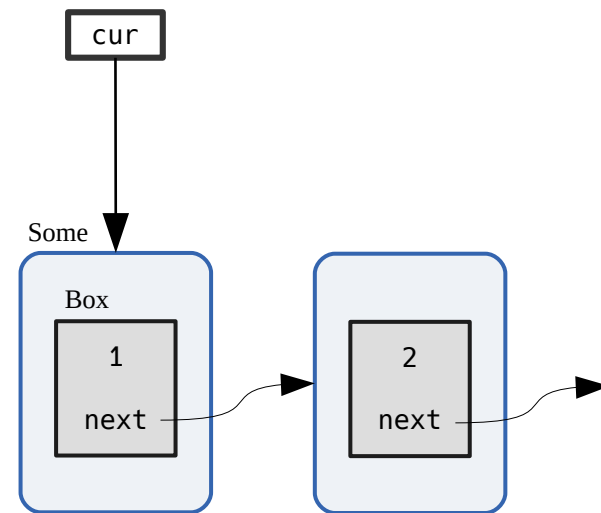
    head
}
```

Example

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```

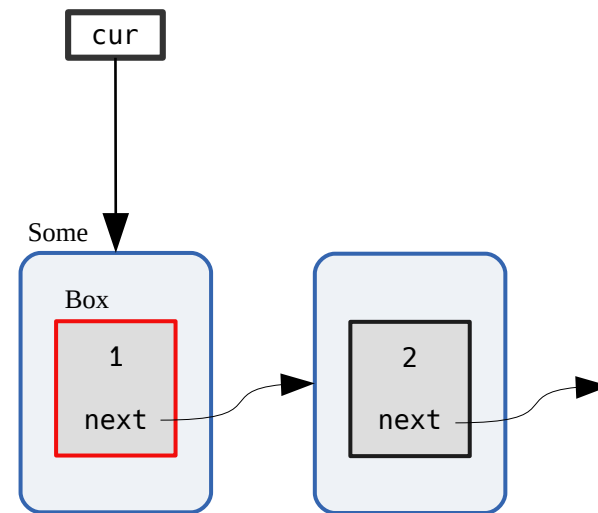


Example

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```

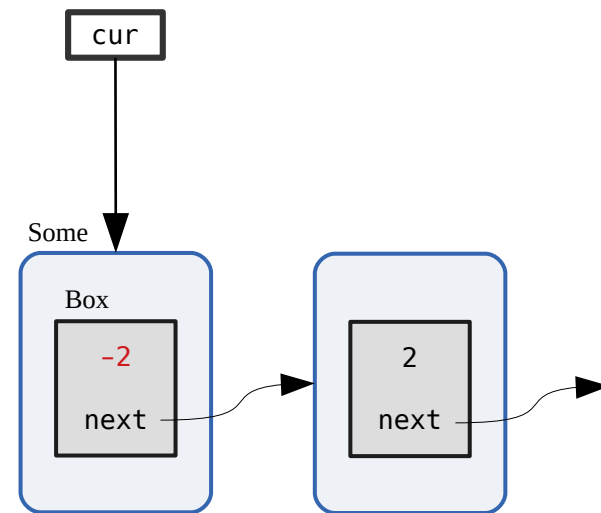


Example

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```



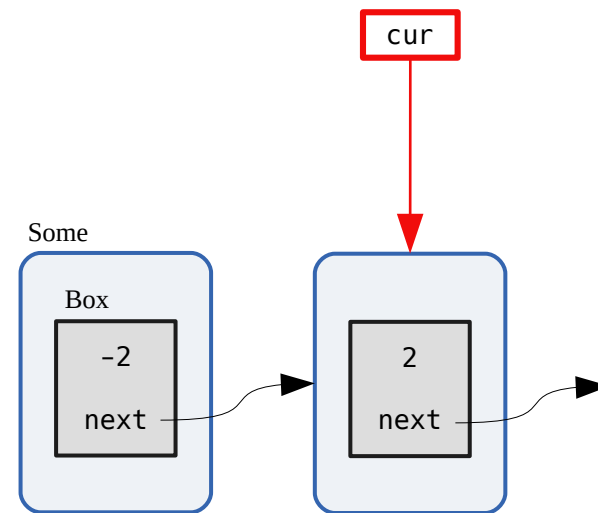
Example

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;

        cur = &mut nodeBox.next;
    }

    head
}
```



依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```

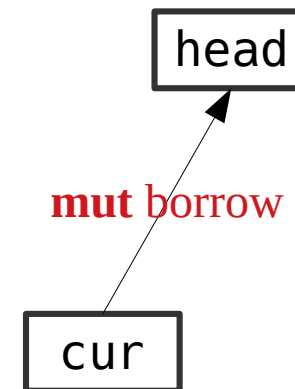
依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head

}
```



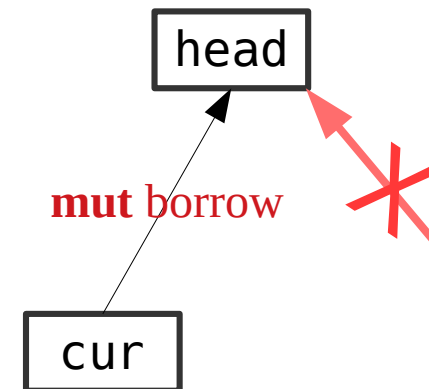
依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head

}
```



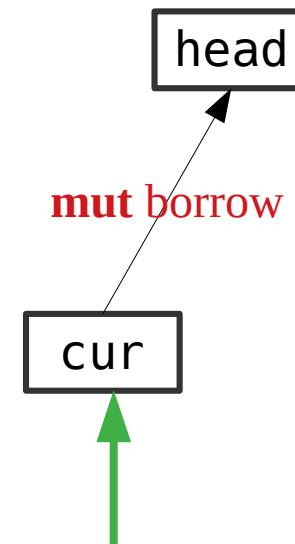
依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head

}
```



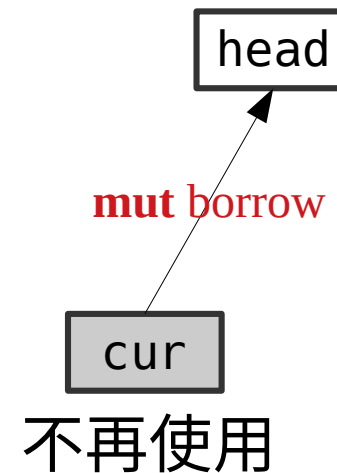
依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head

}
```



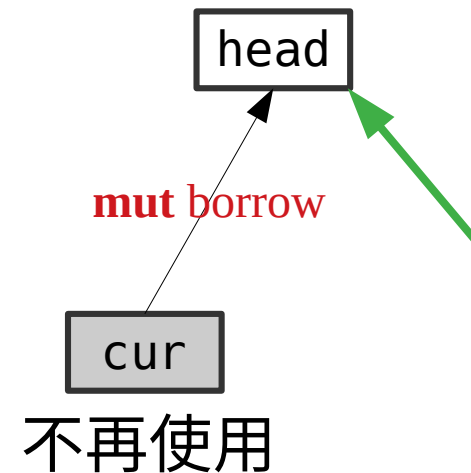
依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head

}
```



依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
```

```
{
```

```
    let mut cur = &mut head;
```

`cur` only used here

```
    while let Some(nodeBox) = cur.as_mut() {
```

```
        nodeBox.val = !nodeBox.val;
```

```
        cur = &mut nodeBox.next;
```

```
    }
```

head

```
}
```

head

mut borrow

cur

不再使用

依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

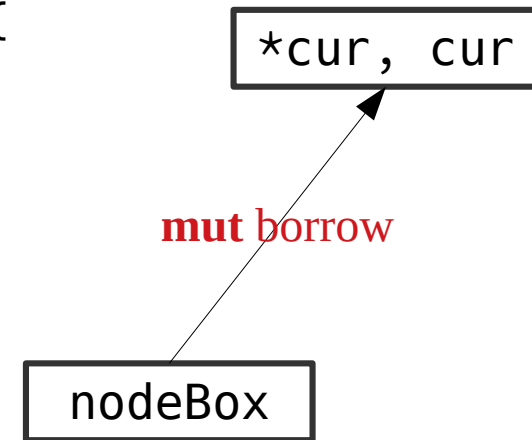
    head
}
```

依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```

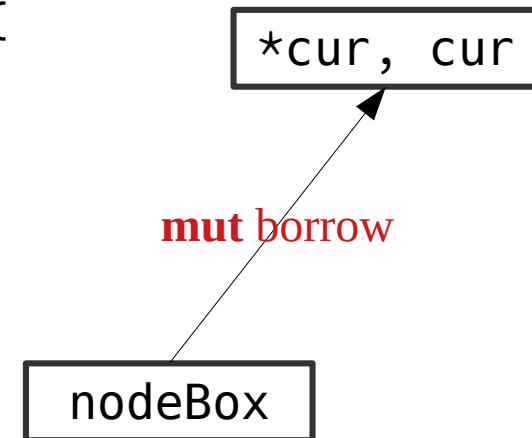


依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```

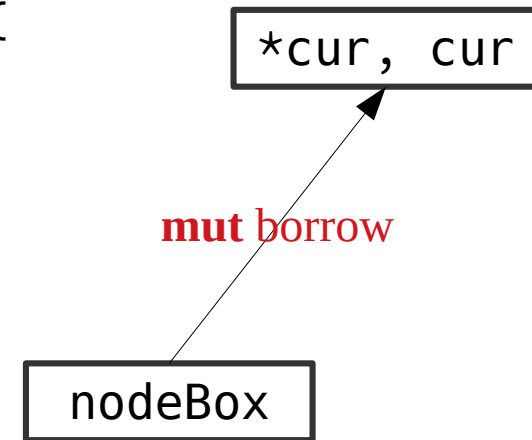


依照使用區間判斷

```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

    while let Some(nodeBox) = cur.as_mut() {
        nodeBox.val = !nodeBox.val;
        cur = &mut nodeBox.next;
    }

    head
}
```



1. **nodeBox** finally used here
2. Assignment to **'cur'** killed the borrow expression

Borrow

P_0 : **let** r : $\&T$

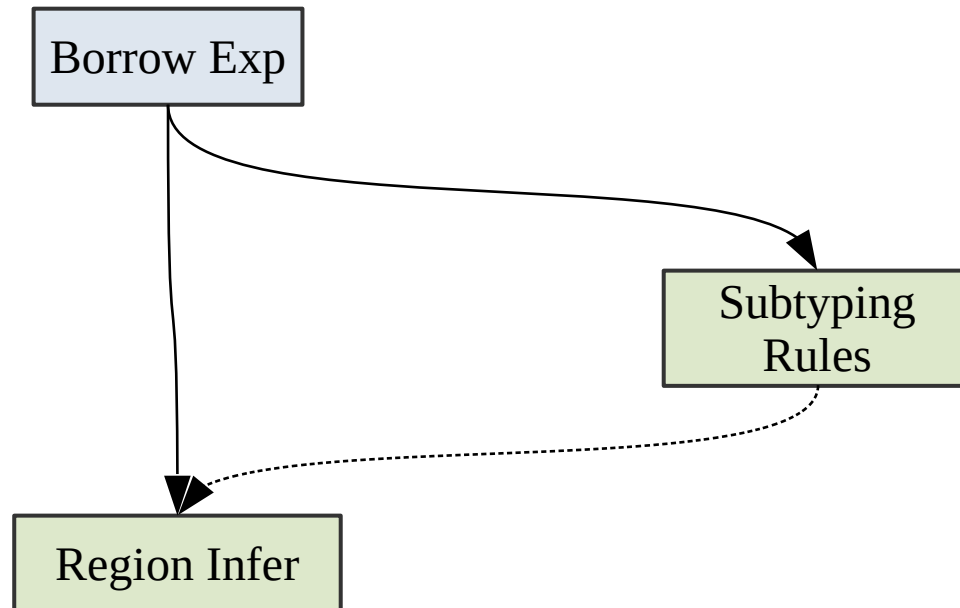
P_1 : $r = \&x$;

Borrow Exp

Borrow

P_0 : **let** r : $\&'0$ T

P_1 : $r = \&'1$ x ;



Borrow

$P_0 : \text{let } r : \&'0 \text{ T}$

$P_1 : r = \&'1 \text{ x};$

Borrow Exp

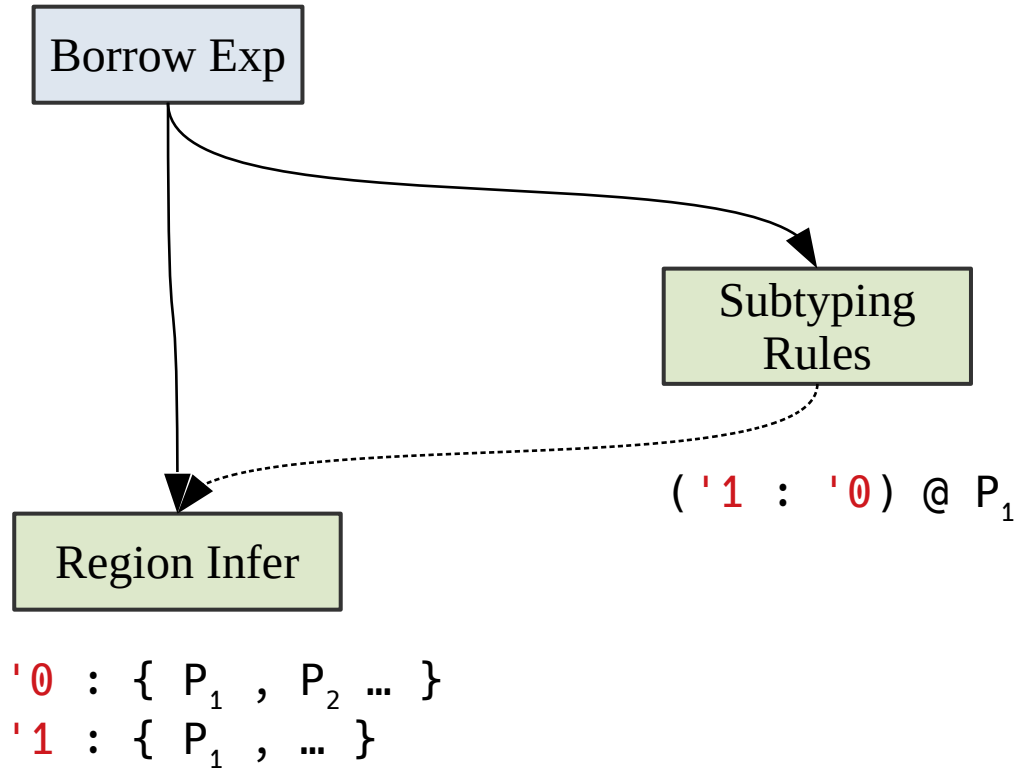
Subtyping
Rules

Region Infer

$('1 : '0) @ P_1$

$'0 : \{ P_1 , P_2 \dots \}$

$'1 : \{ P_1 , \dots \}$



Borrow

Each Borrow expression will
corresponding to each Loan

```
Loan L0 {
  point: P1,
  path: x,
  kind: shared
  region: '1 {
    P1 ...
  }
}
```

$P_0 : \text{let } r : \&'0 \text{ T}$

$P_1 : r = \&'1 x;$

Borrow Exp

Subtyping
Rules

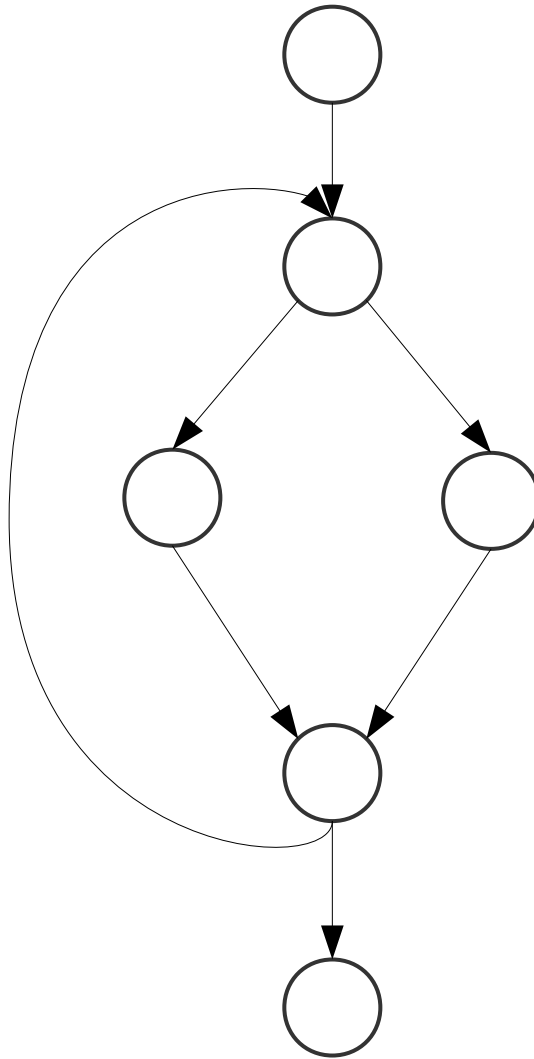
Region Infer

$('1 : '0) @ P_1$

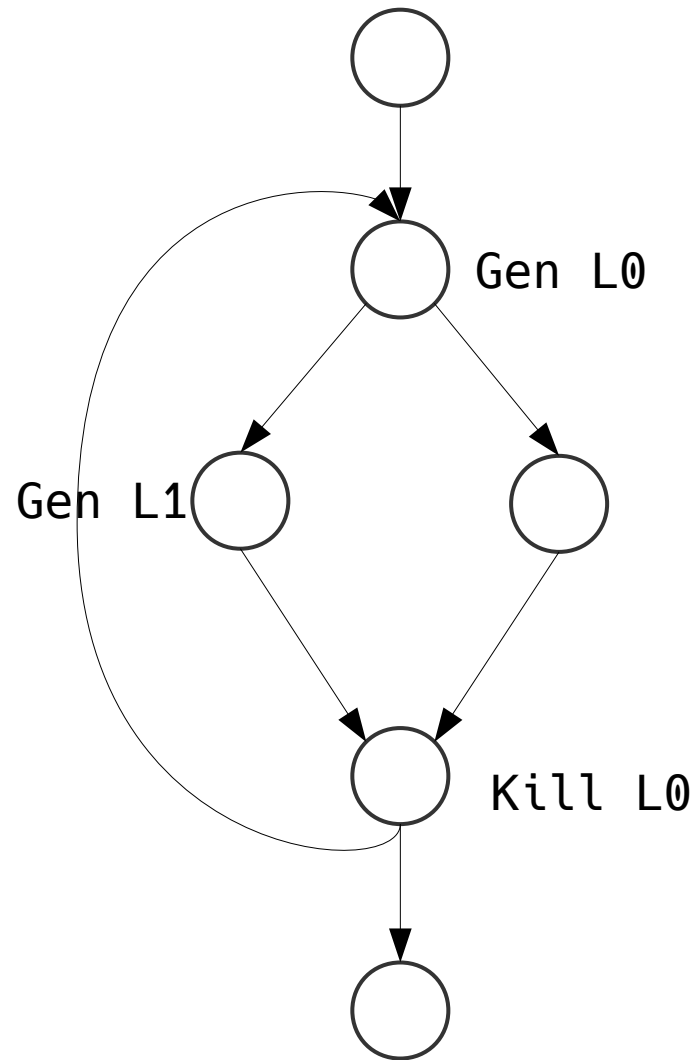
$'0 : \{ P_1 , P_2 \dots \}$

$'1 : \{ P_1 , \dots \}$

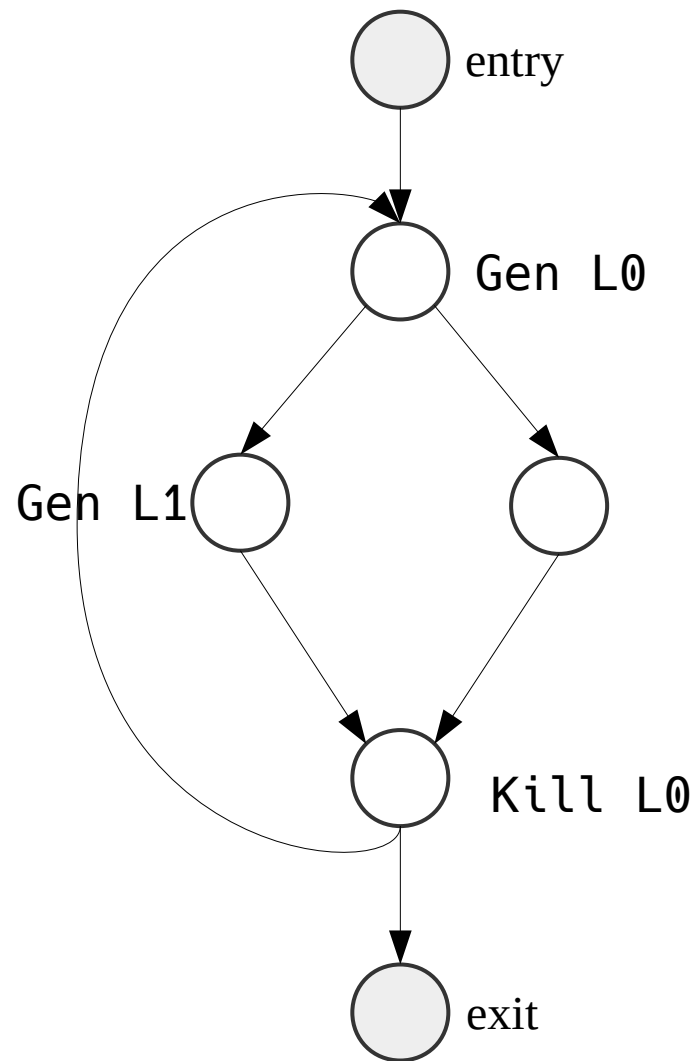
The Data Flow of the Loan



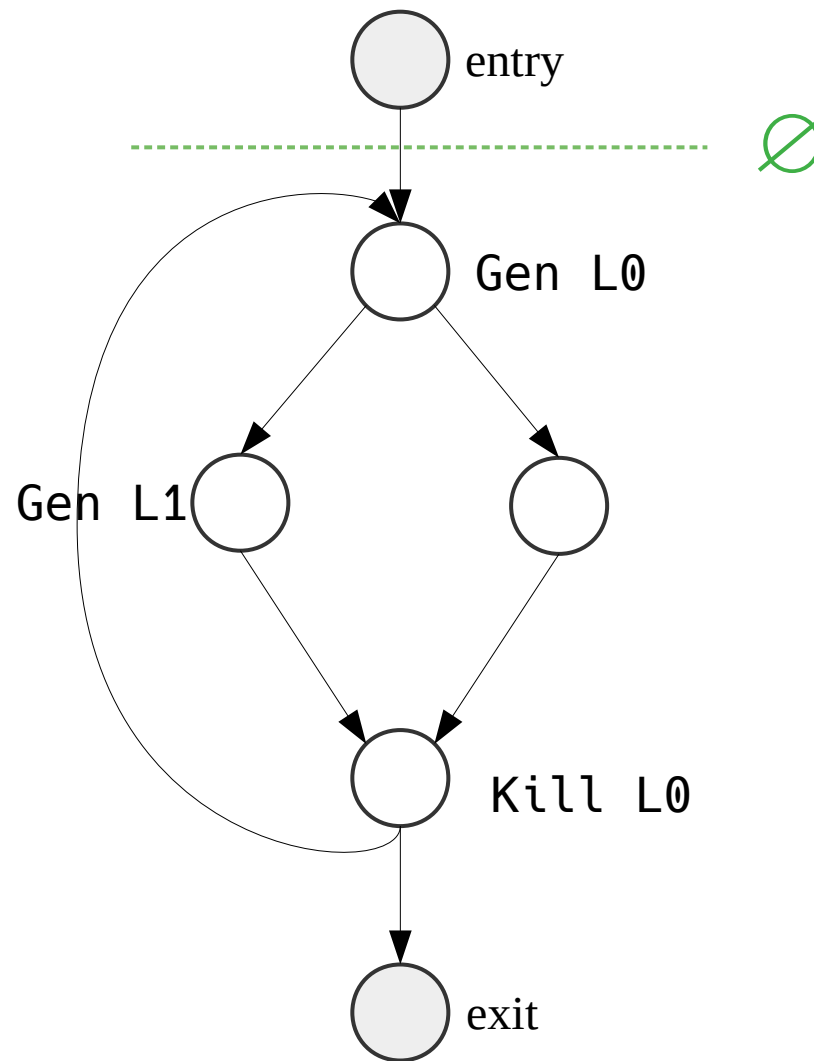
The Data Flow of the Loan



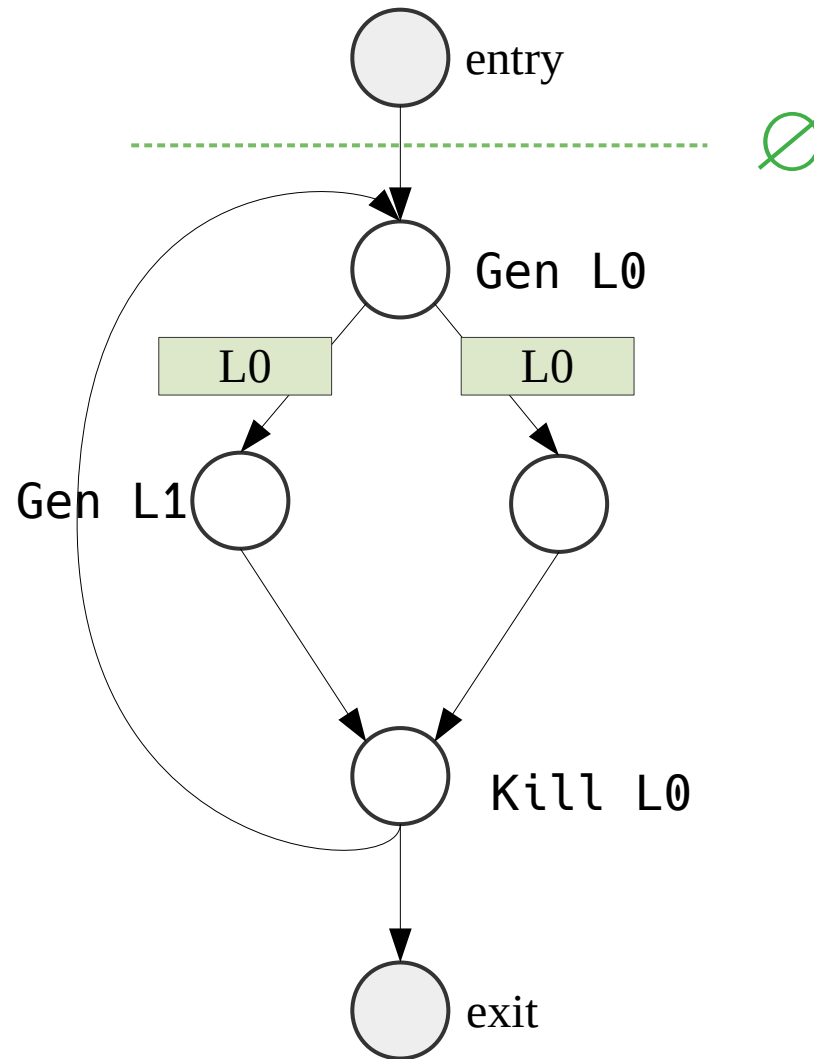
The Data Flow of the Loan



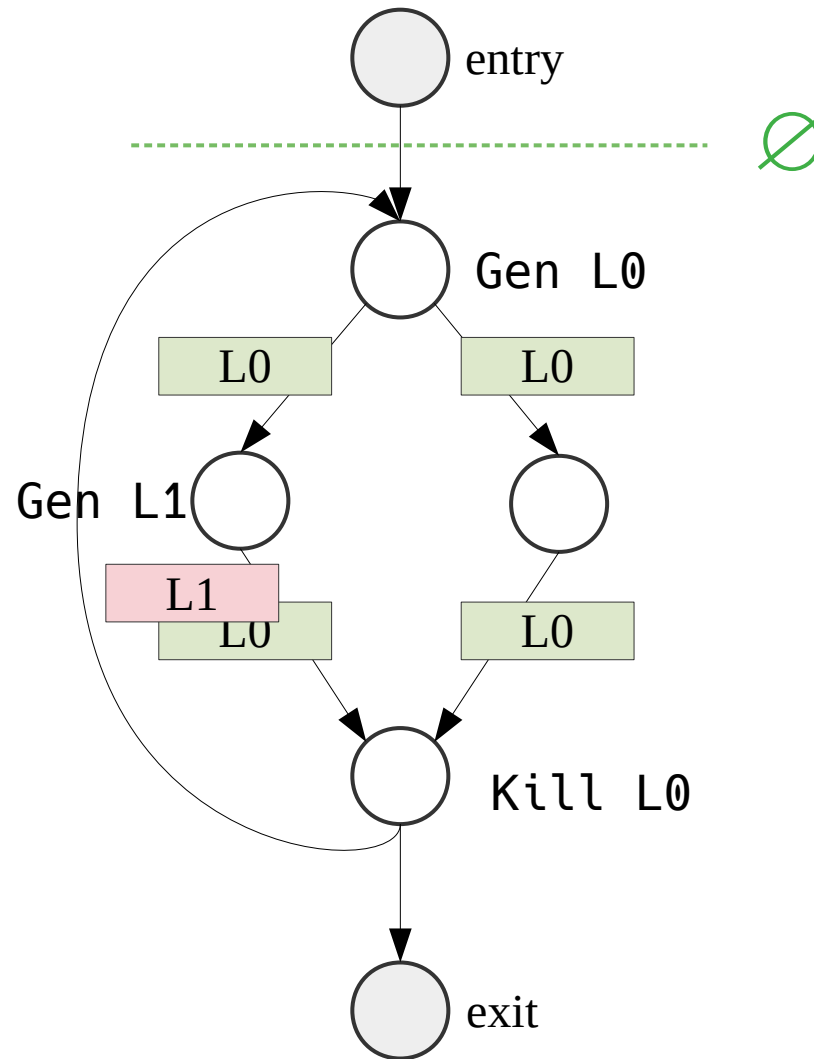
The Data Flow of the Loan



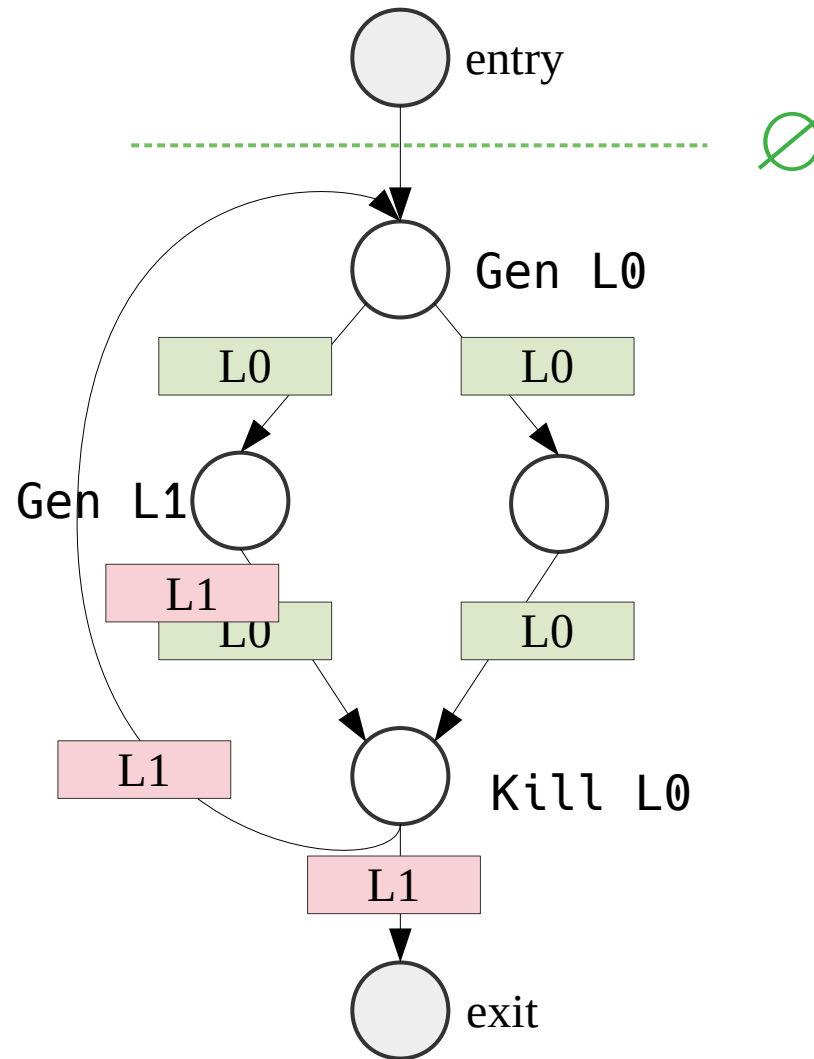
The Data Flow of the Loan



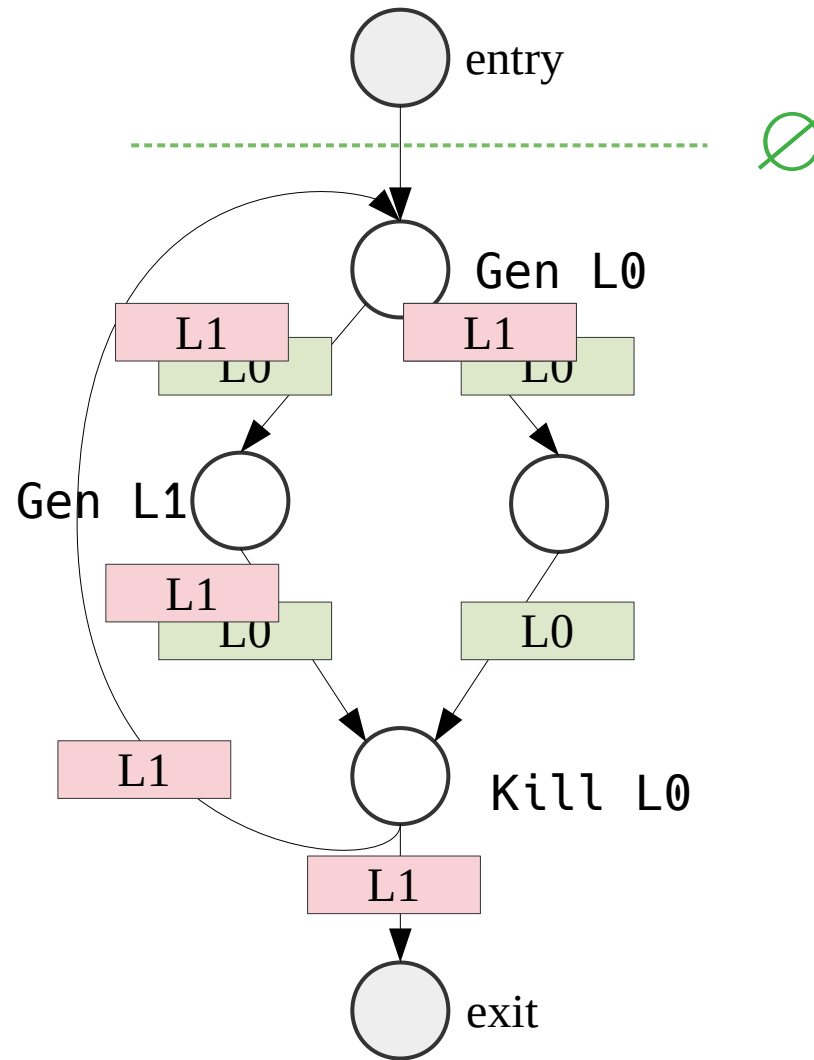
The Data Flow of the Loan



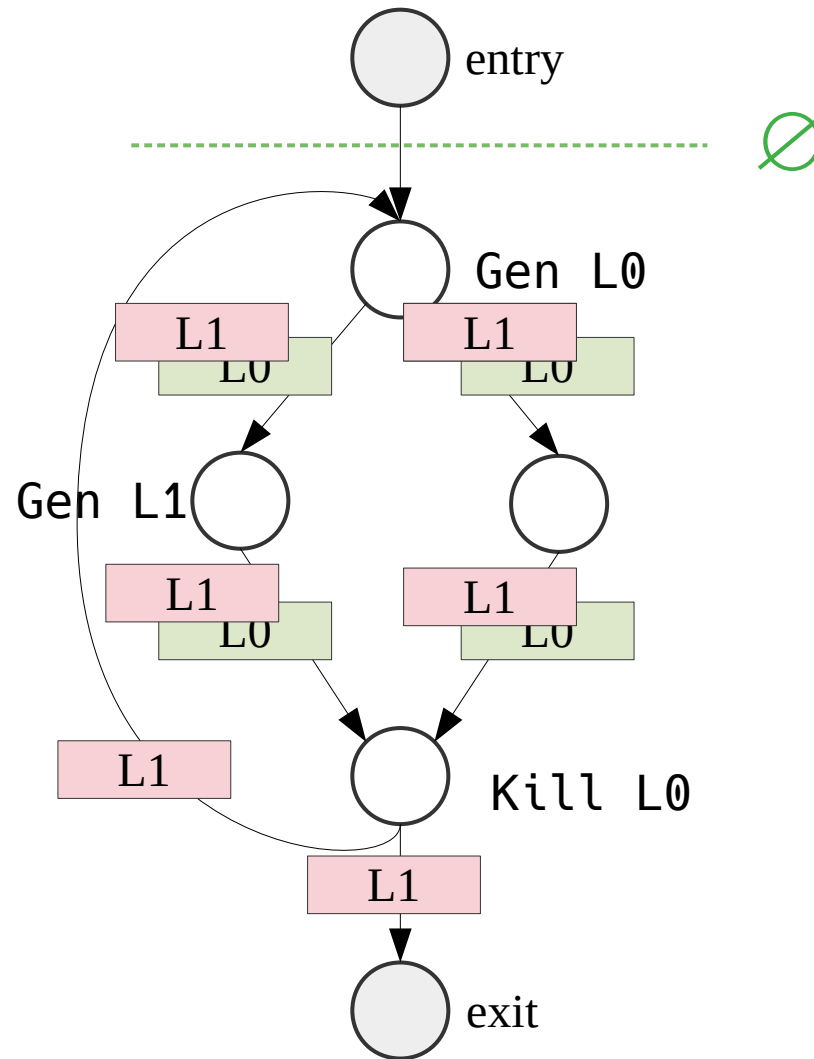
The Data Flow of the Loan



The Data Flow of the Loan

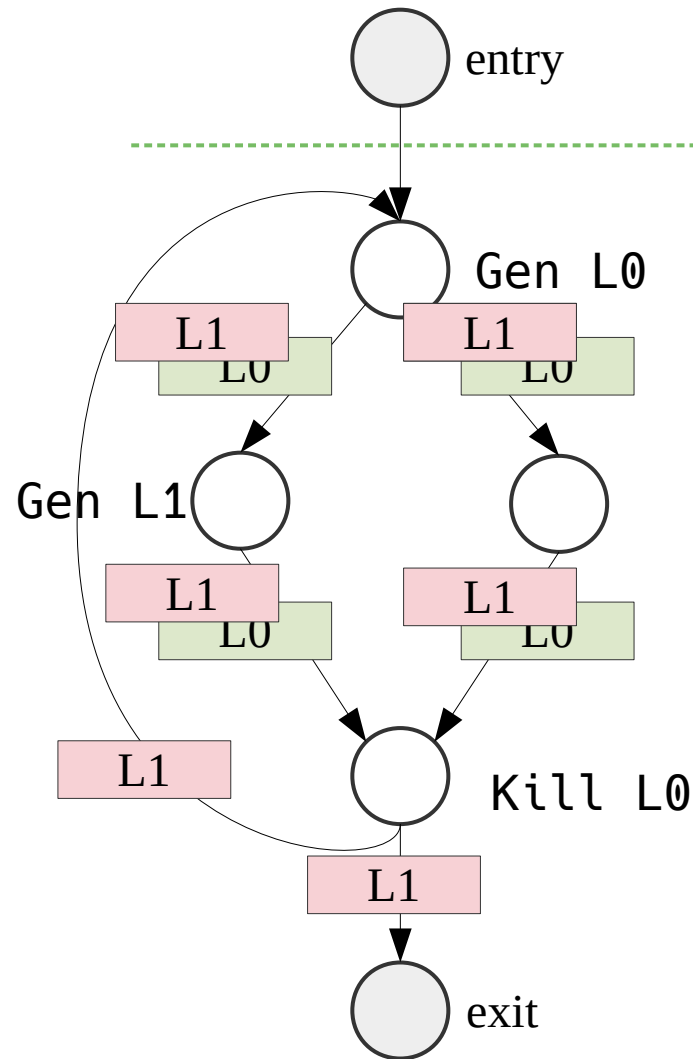


The Data Flow of the Loan



The Data Flow of the Loan

Key: which loan live at which points



When all the sets are stable, that's mean **the state is not changed anymore**, then the data flow computation is complete.

When to Gen, Kill

Gen Loan :

If it's a borrow expression, then gen a Loan

Kill Loan :

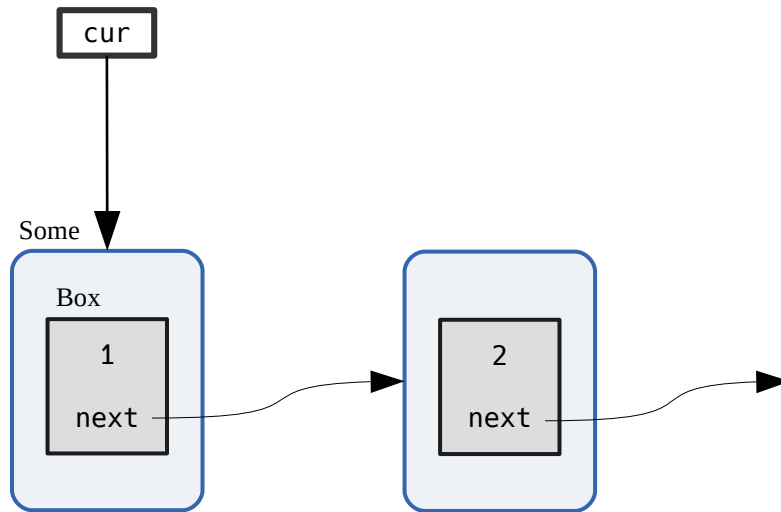
- 1) $LV = \text{Loan}_i . \text{path}$
- 2) $\text{point} \notin \text{Loan}_i . \text{region}$

Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```

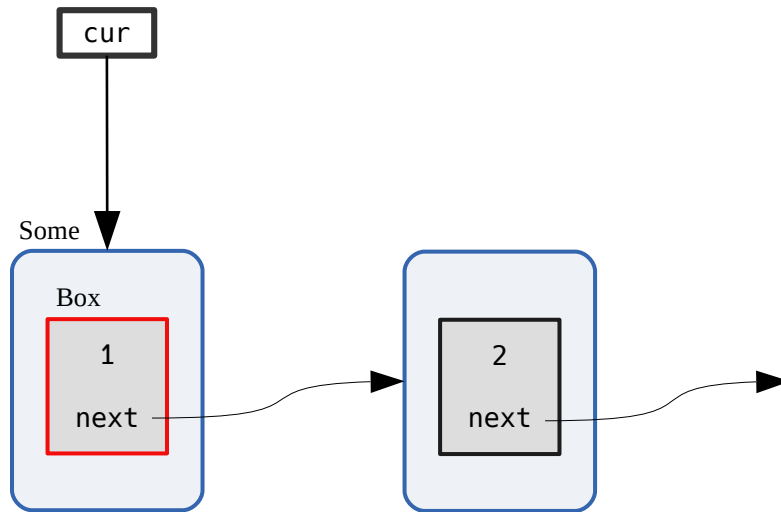
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



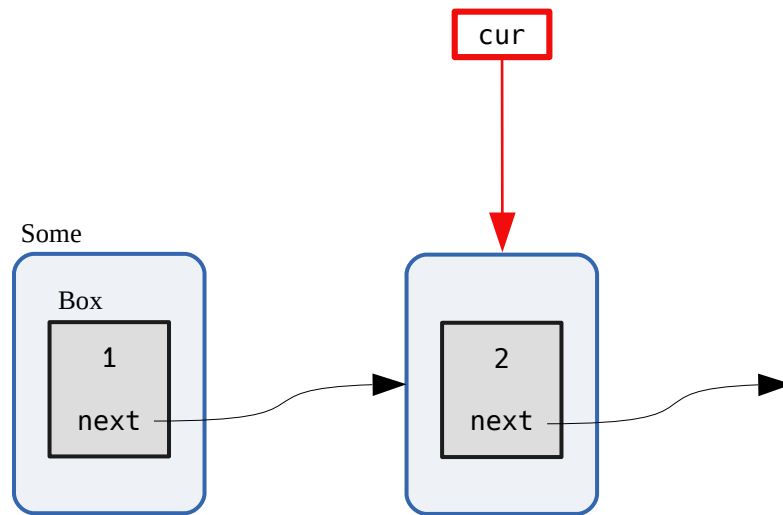
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



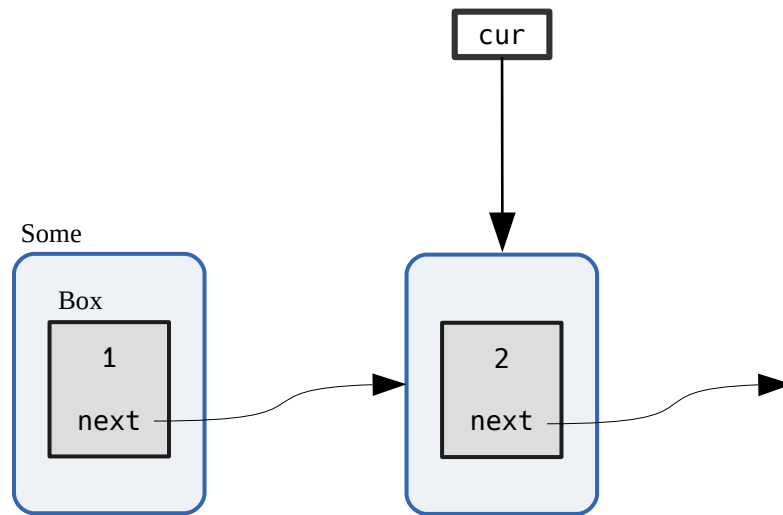
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



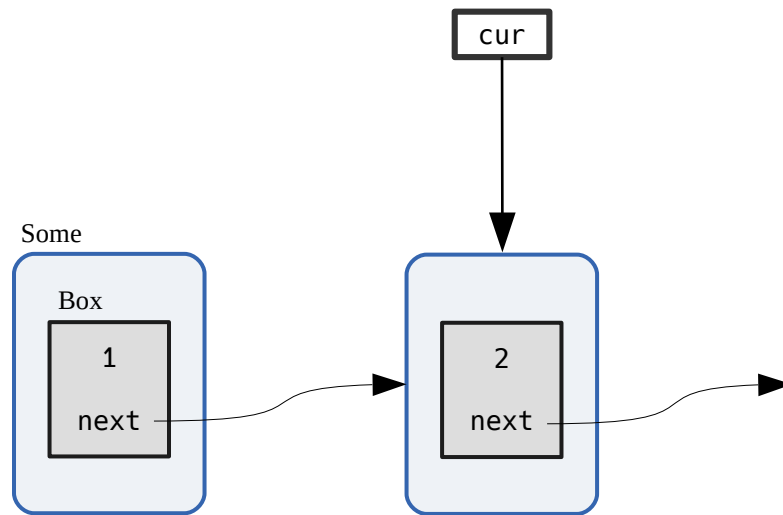
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



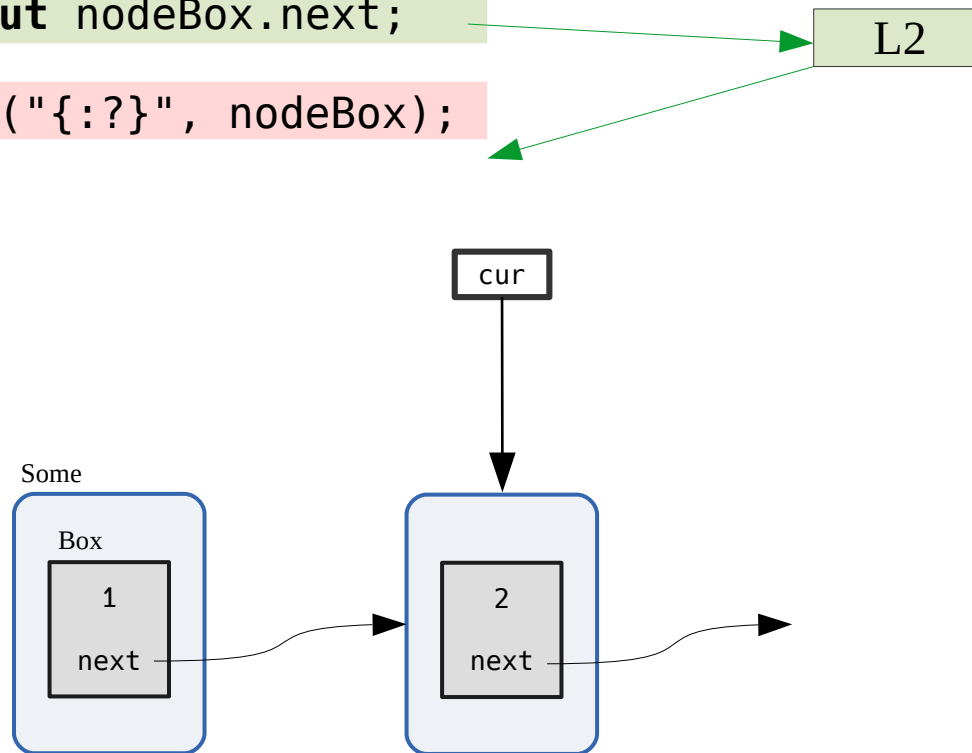
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



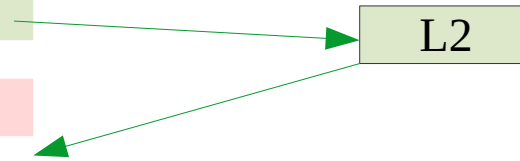
Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



Example

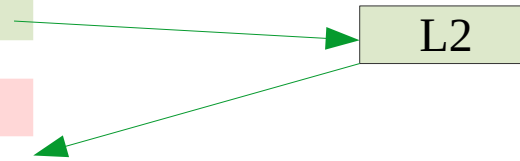
```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



Why `L2` live at this point ?

Example

```
fn do_something(mut head: Option<Box<ListNode>>)  
{  
    let mut cur = &mut head;  
  
    if let Some(nodeBox) = cur.as_mut() {  
        cur = &mut nodeBox.next;  
        println!("{:?}", nodeBox);  
    }  
}
```



Why **`L2'** live at this point ?

1. no assignment to { nodeBox, nodeBox.next }
2. **`cur'** will be used later

參考題目

Leetcode : remove linked list elements

Datafrog

The tool used in Rust's new borrow checker called
Polonius

Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢

Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢

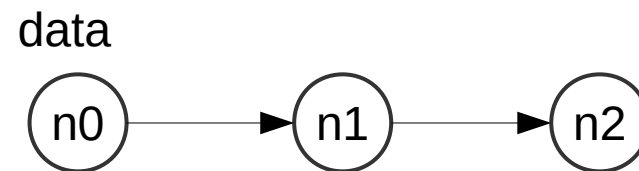
Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢
- ▶ 抽出每次推論的一小步 (Induction)

Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢
- ▶ 抽出每次推論的一小步 (Induction)

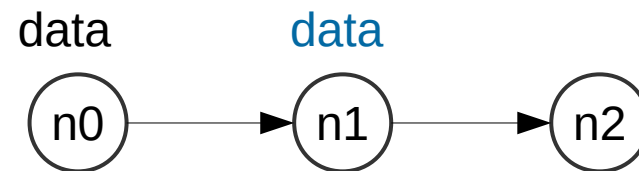
$$\frac{N(a, x) \cdot e(a, b)}{N(b, x)}$$



Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢
- ▶ 抽出每次推論的一小步 (Induction)

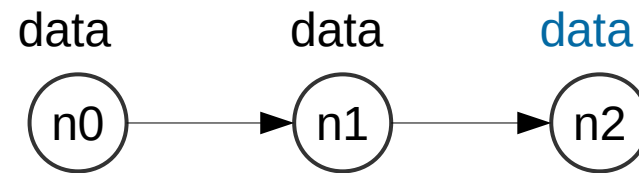
$$\frac{N(a, x) \cdot e(a, b)}{N(b, x)}$$



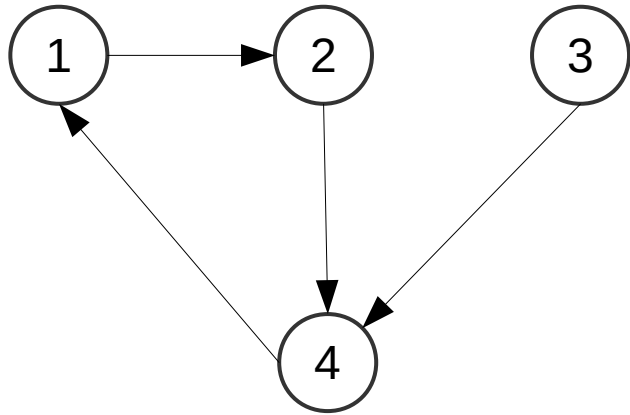
Idea

- ▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢
- ▶ 抽出每次推論的一小步 (Induction)

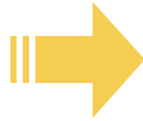
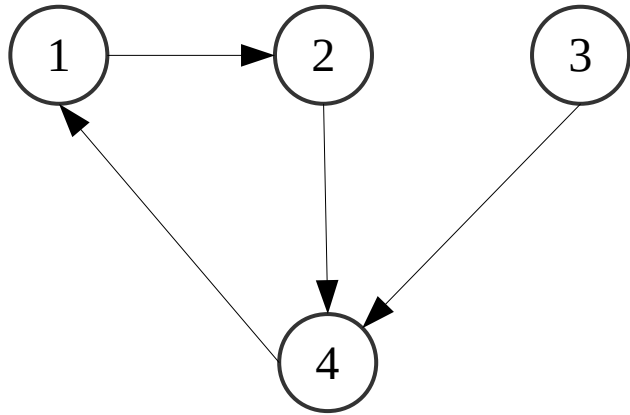
$$\frac{N(a, x) \cdot e(a, b)}{N(b, x)}$$



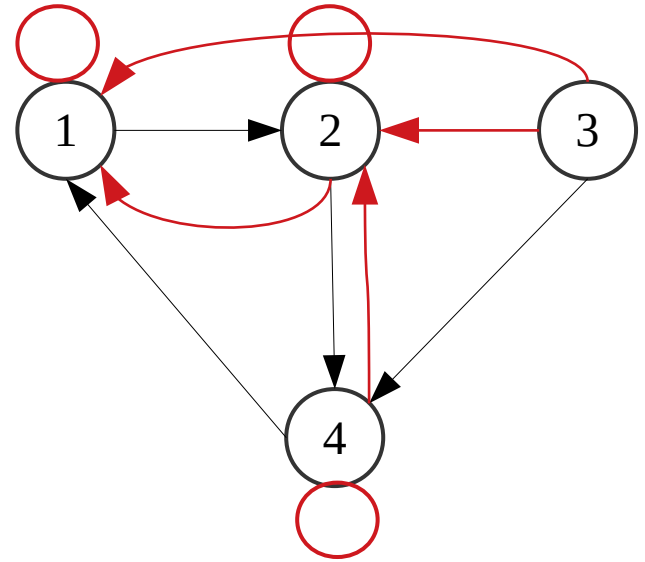
Example . Transitive Closure



Example . Transitive Closure



$$\frac{e(a,b) \quad e(b,c)}{e(a,c)}$$



Implementation – Initial

```
// create a iteration context
let mut iteration = Iteration::new();

// create some variables for later use
let v_edges  = iteration.variable::<(u32, u32)>("edges");
let v_redges = iteration.variable::<(u32, u32)>("reverse edges");

// load the initial variables
v_edges.insert(edges.into());

// start iteration
while iteration.changed() {

    ...

}


let result = v_edges.complete();
```

Implementation – Initial

```
// create a iteration context
let mut iteration = Iteration::new();

// create some variables for later use
let v_edges  = iteration.variable::<(u32, u32)>("edges");
let v_redges = iteration.variable::<(u32, u32)>("reverse edges");

// load the initial variables
v_edges.insert(edges.into());

// start iteration
while iteration.changed() {
     Writing Rules here
}

let result = v_edges.complete();
```

Implementation – Writing Rules

```
while iteration.changed() {  
    // reverse edges for mapping  
    v_redges.from_map(&v_edges, |&(a, b)| (b, a));  
  
    // e(a,c) <- e(a,b), e(b,c)  
    v_edges.from_join(&v_redges, &v_edges, |_b, &a, &c| (a, c));  
}
```

Implementation – Writing Rules

```
while iteration.changed() {  
    // reverse edges for mapping  
    v_redges.from_map(&v_edges, |&(a, b)| (b, a));  
  
    // e(a,c) <- e(a,b), e(b,c)  
    v_edges.from_join(&v_redges, &v_edges, |_b, &a, &c| (a, c));  
}
```

$$\frac{e(a,b) \quad e(b,c)}{e(a,c)}$$

Implementation – Writing Rules

```
while iteration.changed() {  
    // reverse edges for mapping  
    v_redges.from_map(&v_edges, |&(a, b)| (b, a));  
  
    // e(a,c) <- e(a,b), e(b,c)  
    v_edges.from_join(&v_redges, &v_edges, |_b, &a, &c| (a, c));  
}
```

$$\frac{e(a,b)}{e(b,c)} \frac{e(b,c)}{e(a,c)}$$



$$\frac{e(a,b)}{r(b,a)}$$

$$\frac{r(b,a)}{e(b,c)} \frac{e(b,c)}{e(a,c)}$$

Implementation – Writing Rules

```

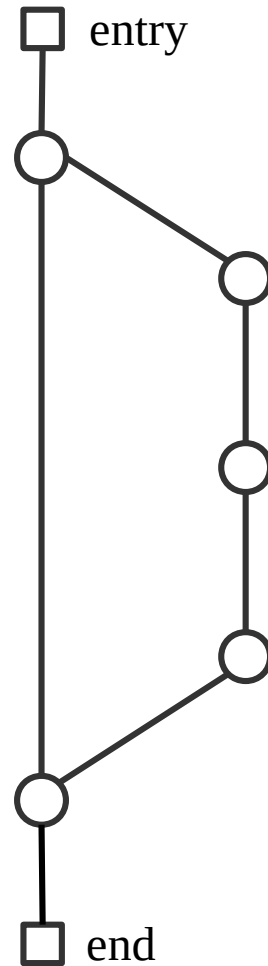
while iteration.changed() {
    // reverse edges for mapping
    v_redges.from_map(&v_edges, |&(a, b)| (b, a));

    // e(a,c) <- e(a,b), e(b,c)
    v_edges.from_join(&v_redges, &v_edges, |_b, &a, &c| (a, c));
}
    
```

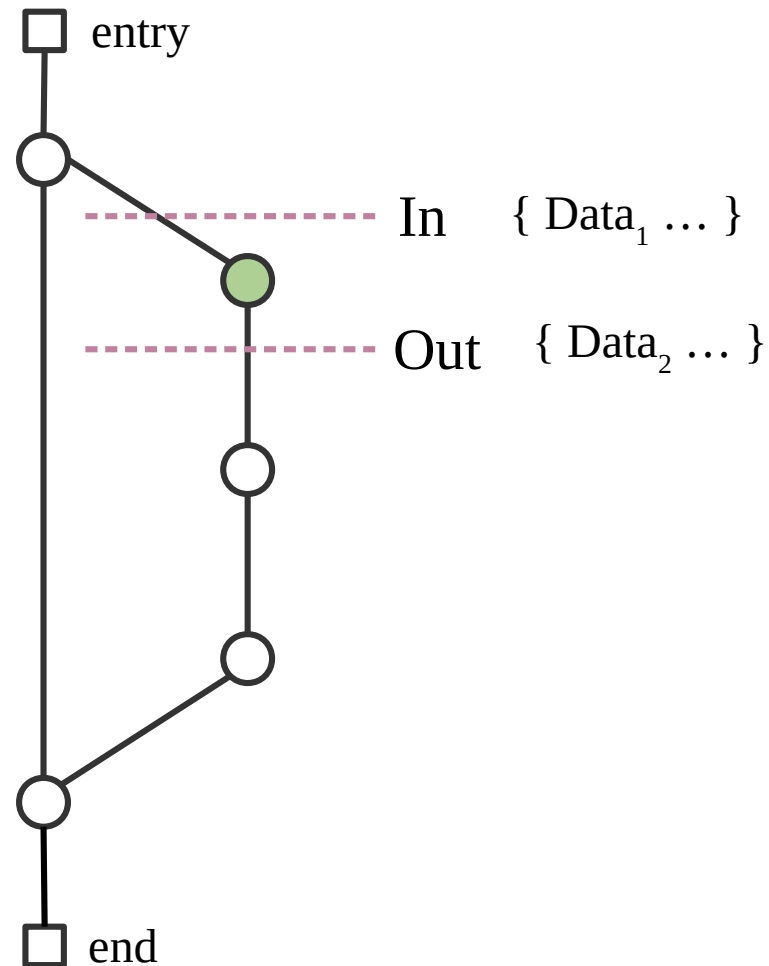
$$\frac{e(a,b)}{e(b,c)} \quad \Rightarrow \quad \frac{e(a,b)}{r(b,a)} \quad \frac{r(\textcolor{red}{b}, \textcolor{green}{a})}{e(\textcolor{red}{b}, \textcolor{blue}{c})} \quad \frac{e(a,c)}{e(a,c)}$$

QA

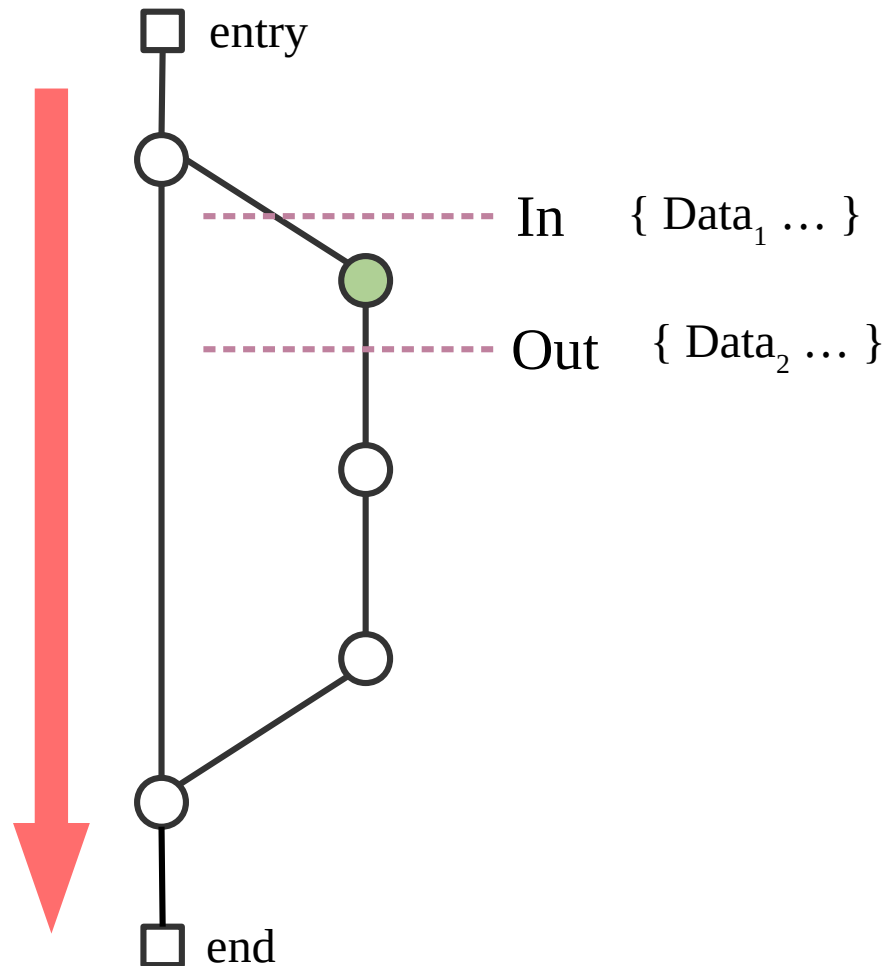
Data Flow Concepts $\langle D, V, \wedge, F \rangle$



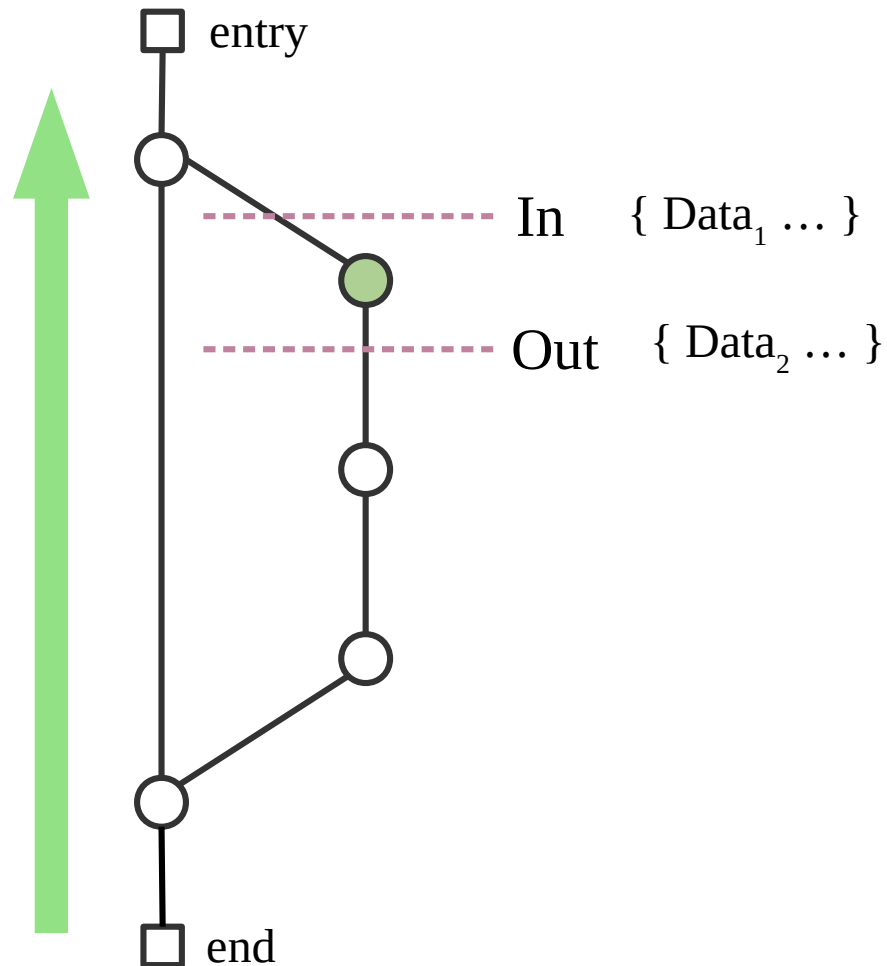
Data Flow Concepts $\langle D, \mathbf{V}, \Lambda, F \rangle$



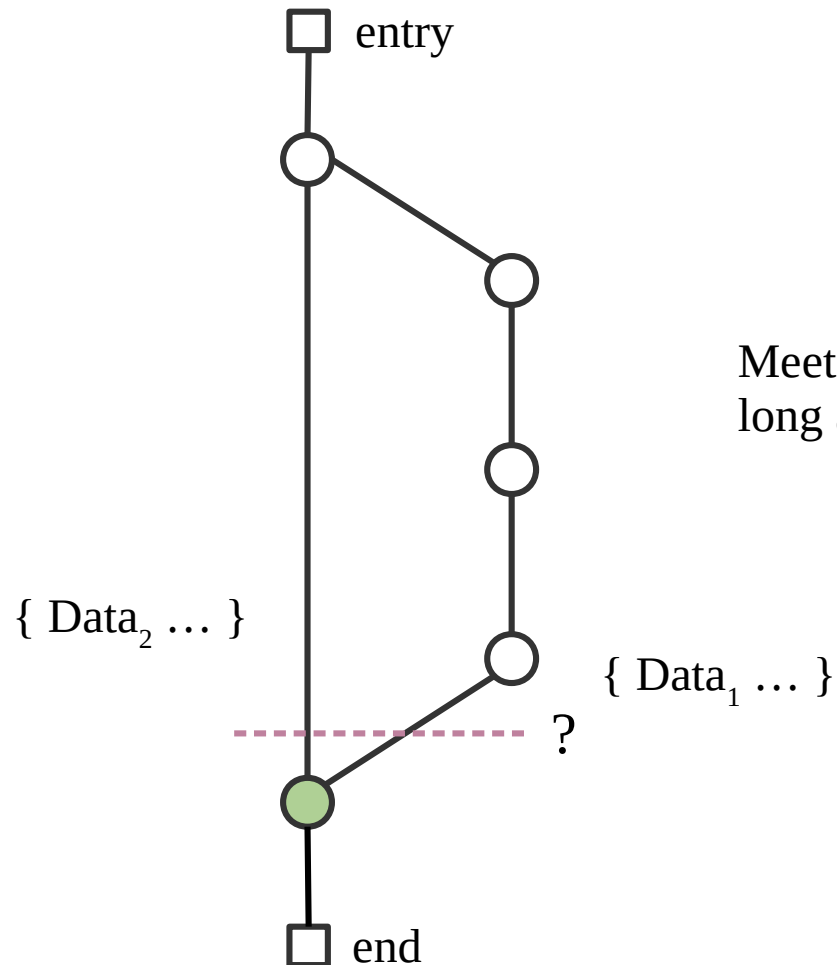
Data Flow Concepts $\langle \mathbf{D}, V, \Lambda, F \rangle$



Data Flow Concepts $\langle \mathbf{D}, V, \Lambda, F \rangle$

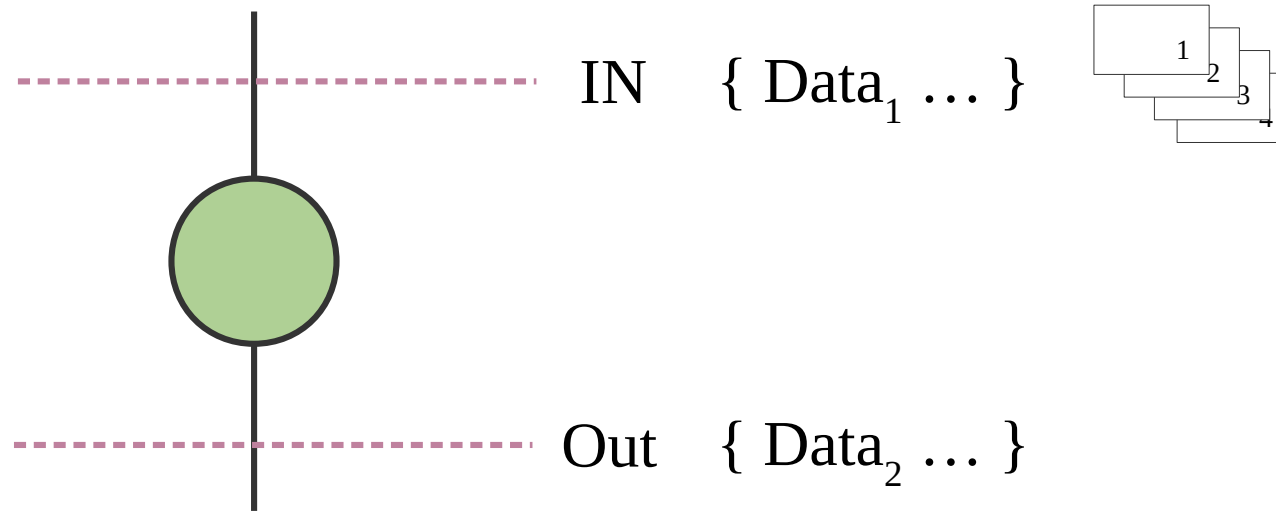


Data Flow Concepts $\langle D, V, \mathbf{\Lambda}, F \rangle$

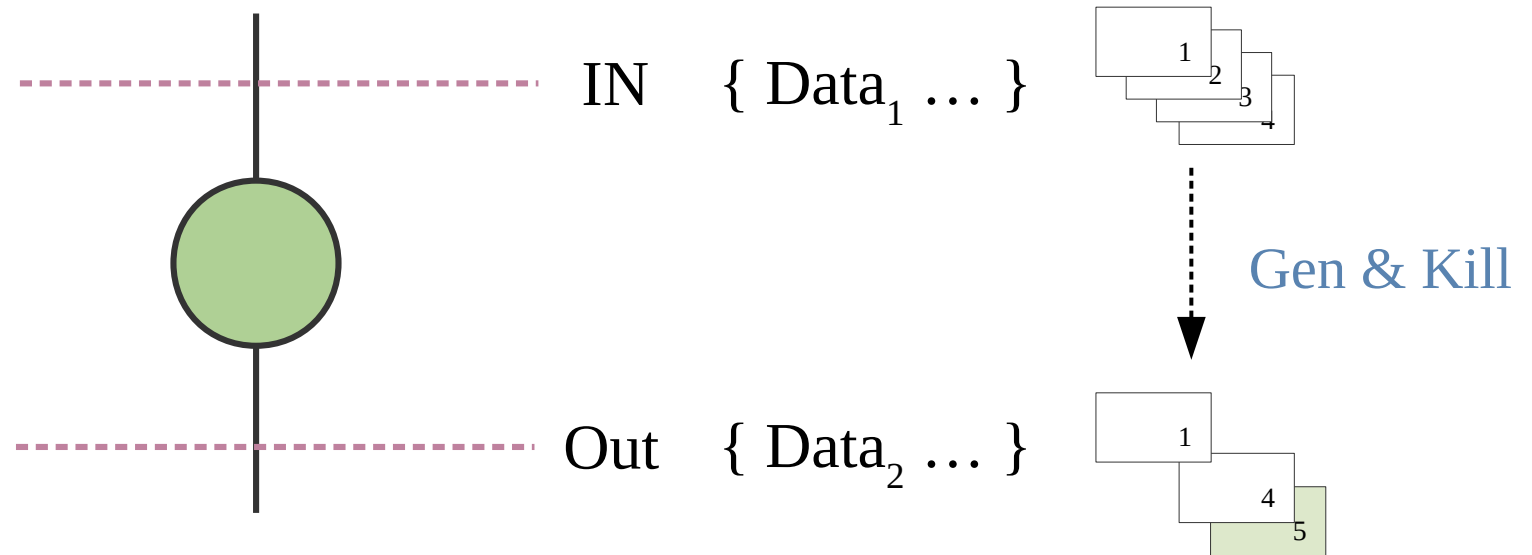


Meet operator can be union, intersection, as long as it can make $(V, \mathbf{\Lambda})$ semilattice.

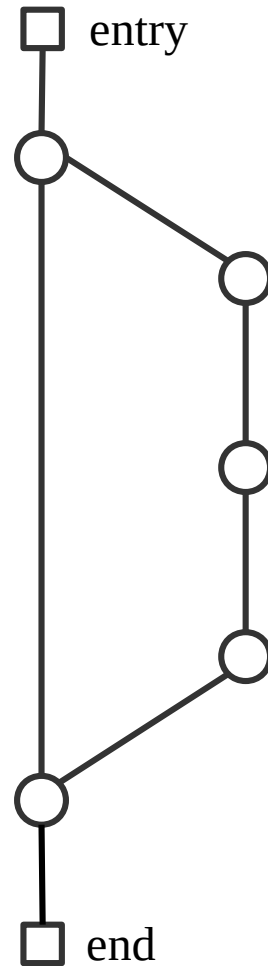
Data Flow Concepts $\langle D, V, \wedge, \mathbf{F} \rangle$



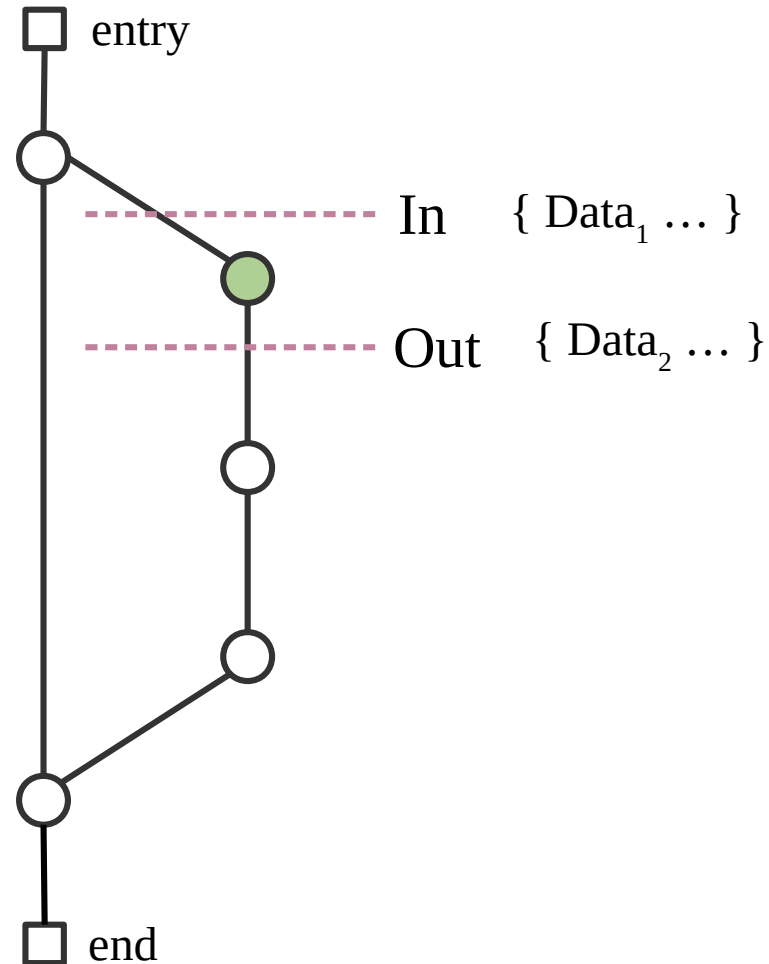
Data Flow Concepts $\langle D, V, \wedge, \mathbf{F} \rangle$



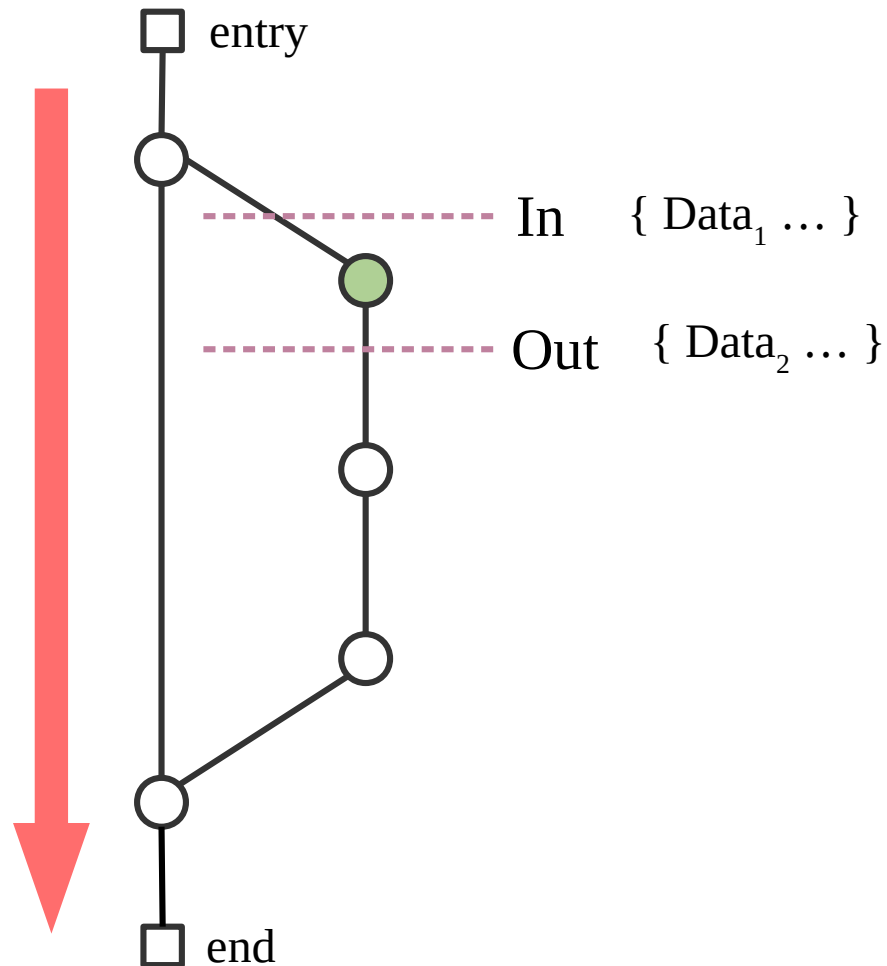
Data Flow Concepts $\langle D, V, \wedge, F \rangle$



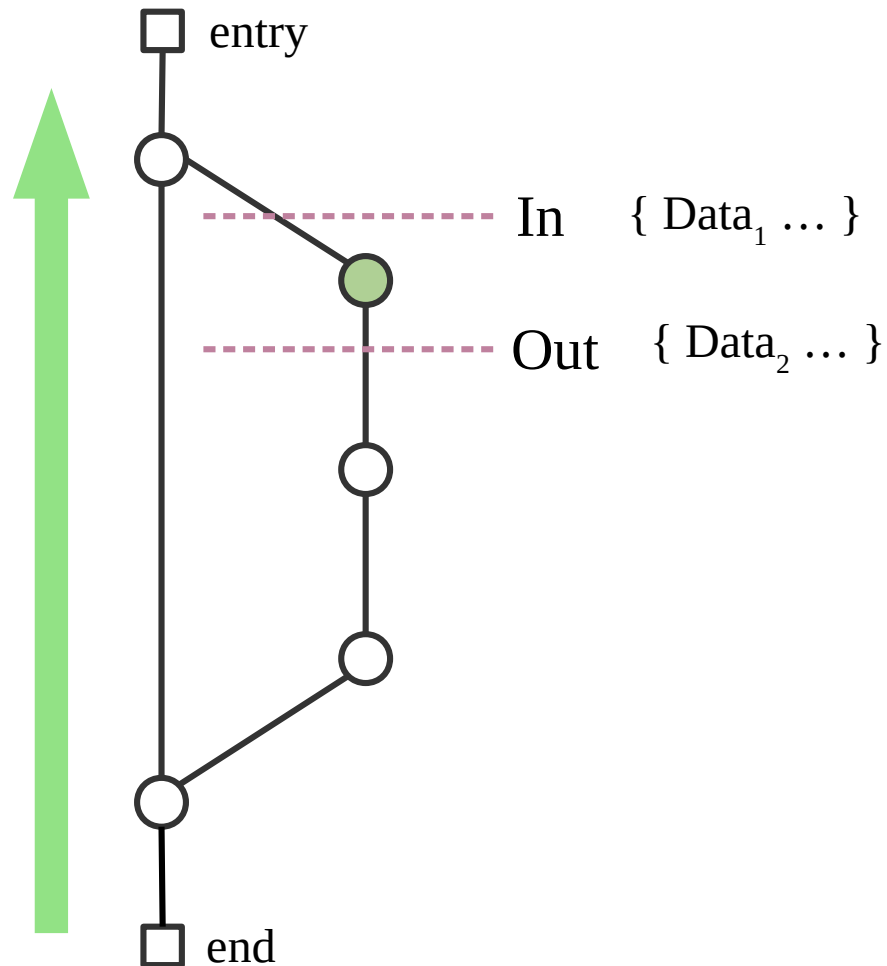
Data Flow Concepts $\langle D, \mathbf{V}, \Lambda, F \rangle$



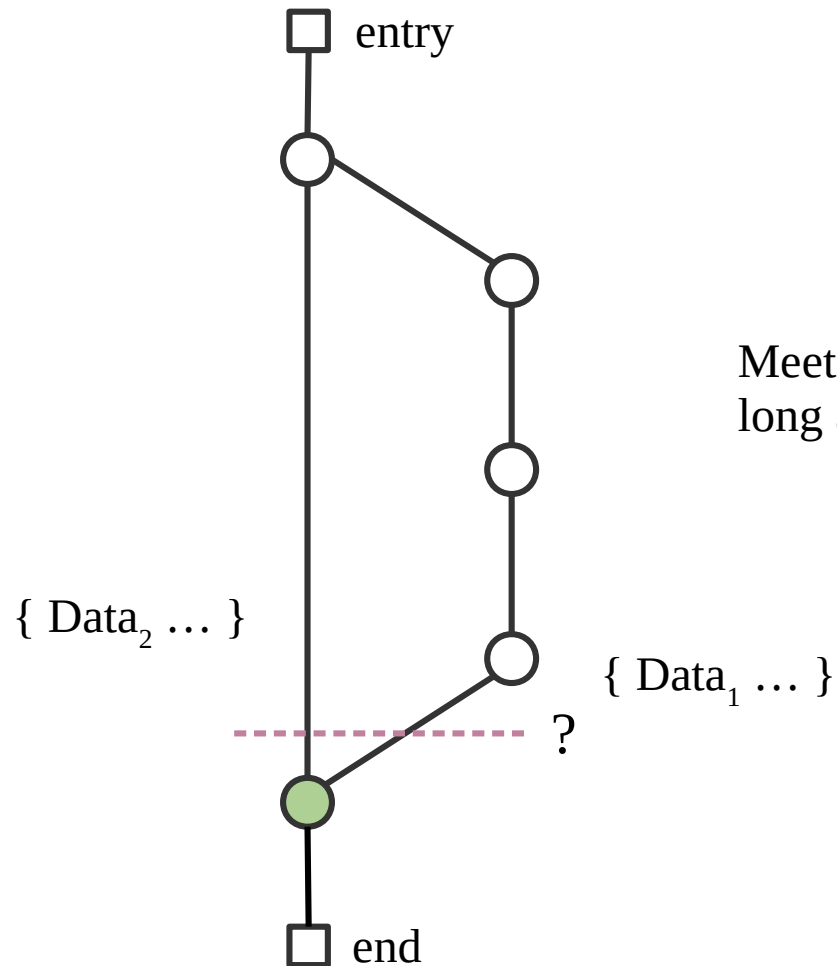
Data Flow Concepts $\langle \mathbf{D}, V, \Lambda, F \rangle$



Data Flow Concepts $\langle \mathbf{D}, V, \Lambda, F \rangle$

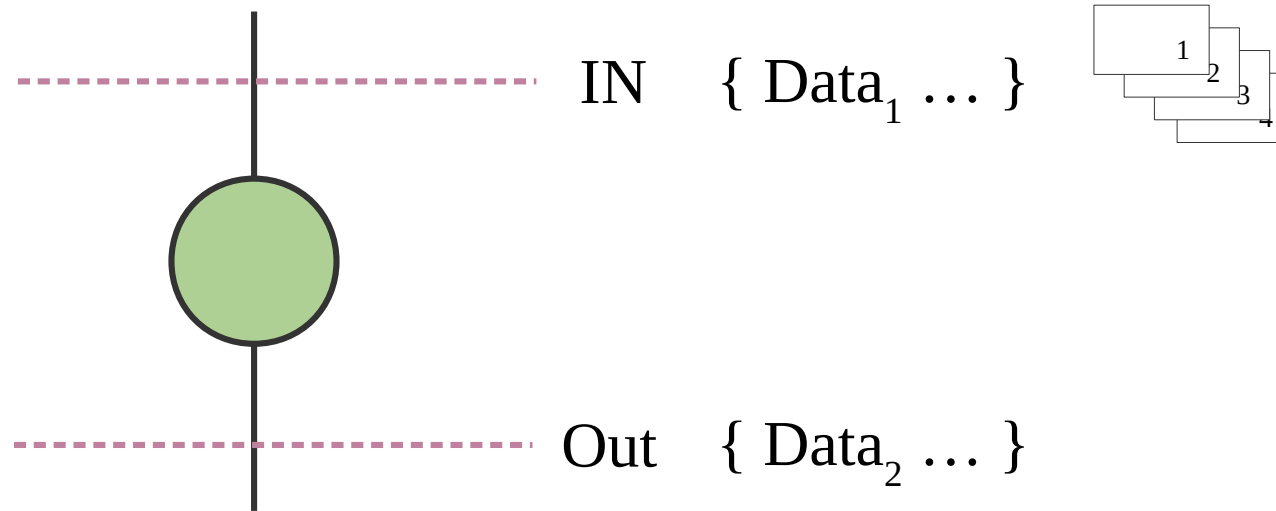


Data Flow Concepts $\langle D, V, \mathbf{\Lambda}, F \rangle$



Meet operator can be union, intersection, as long as it can make $(V, \mathbf{\Lambda})$ semilattice.

Data Flow Concepts $\langle D, V, \wedge, \mathbf{F} \rangle$



Data Flow Concepts $\langle D, V, \wedge, \mathbf{F} \rangle$

