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

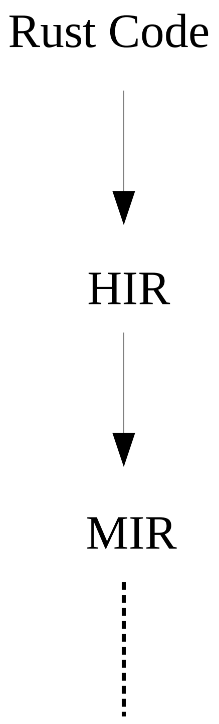
```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```

Compilation of Rust

$$\begin{tikzcd}\text{Rust Code} \arrow[d] \\ \text{HIR}\end{tikzcd}$$

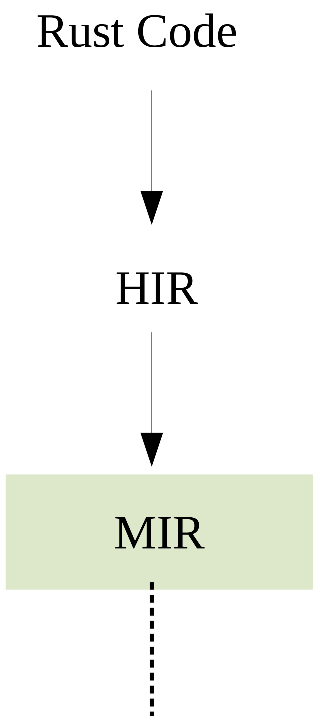
```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```

Compilation of Rust

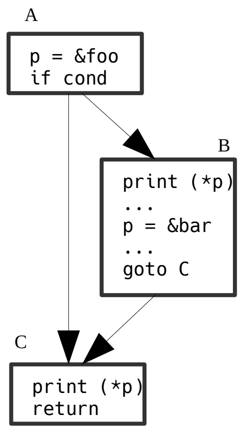


```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```

Compilation of Rust

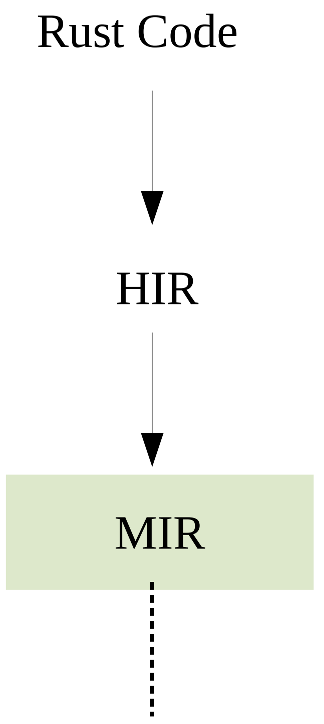


```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```

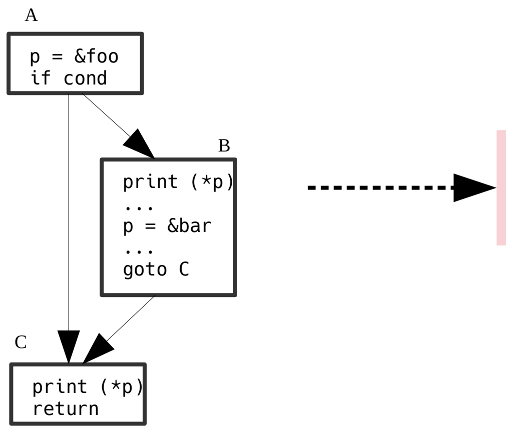


Control Flow Graph

Compilation of Rust



```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```



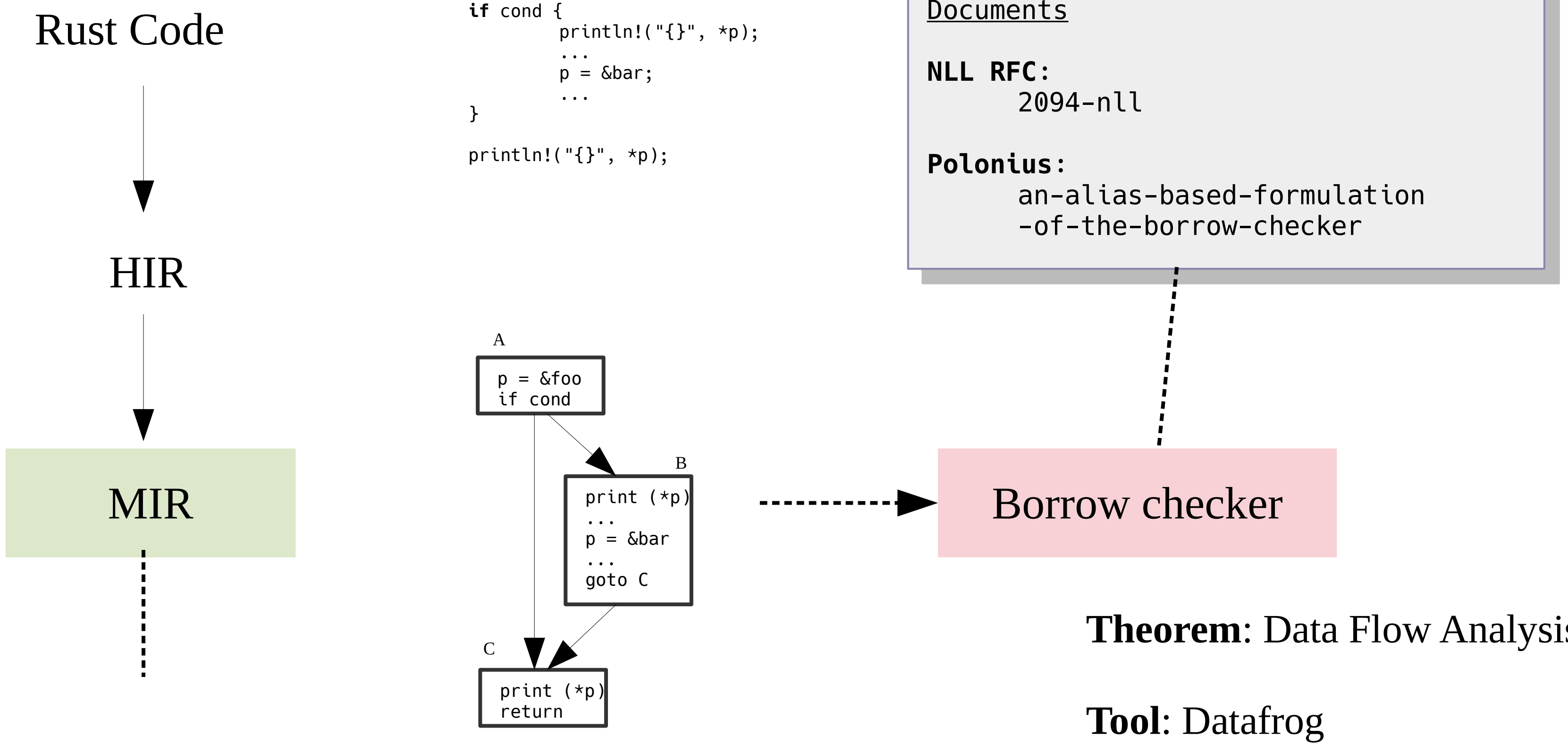
Control Flow Graph

Theorem: Data Flow Analysis

Tool: Datafrog

Compilation of Rust

```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

if cond {
        println!("{}", *p);
        ...
        p = &bar;
        ...
}

println!("{}", *p);
```



Control Flow Graph

<u>Documents</u>

NLL RFC:
    2094-nll

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

Theorem: Data Flow Analysis

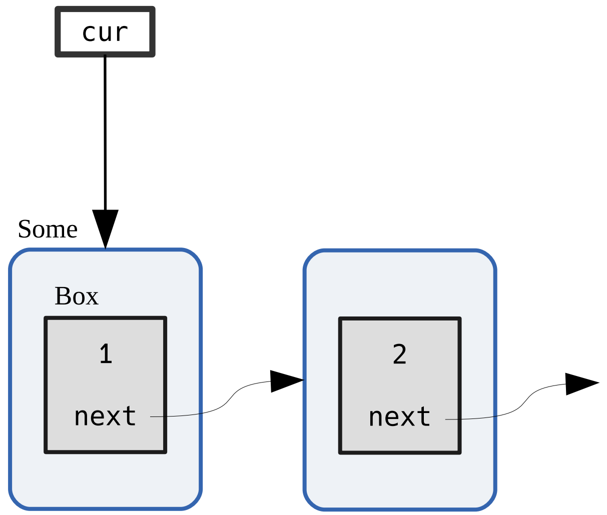
Tool: Datafrog

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

}
```

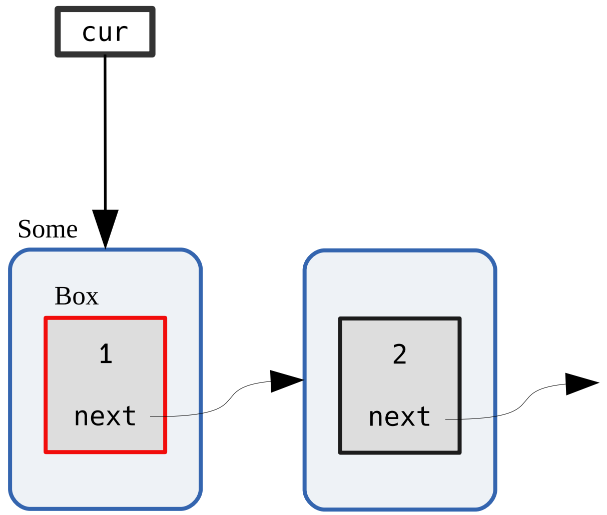
Example

```
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


}
```



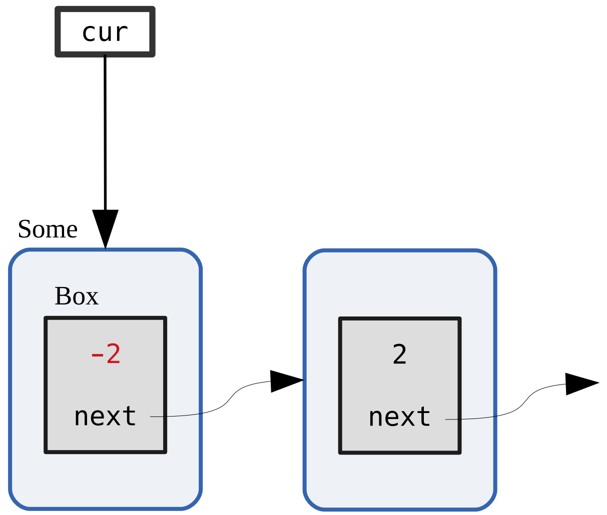
Example

```
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


}
```



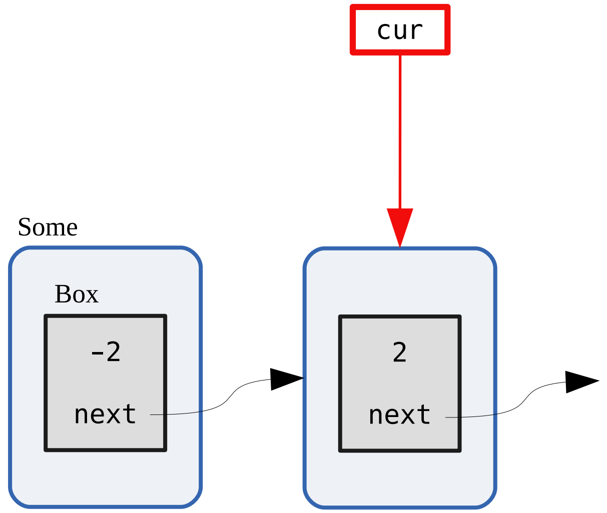
Example

```
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


}
```

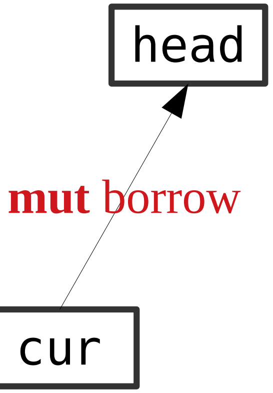


Using region

```
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


}
```

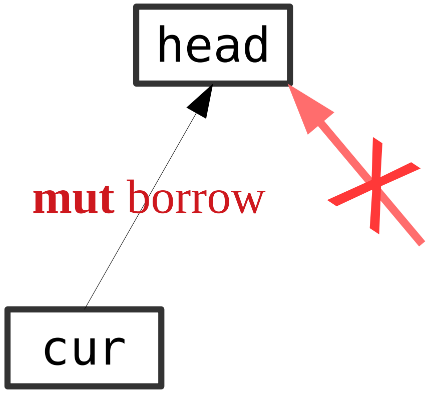
Using region

```
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

}
```



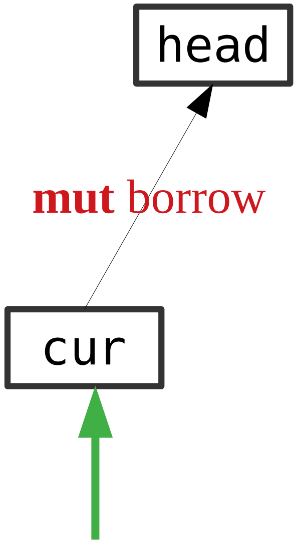
Using region

```
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

}
```



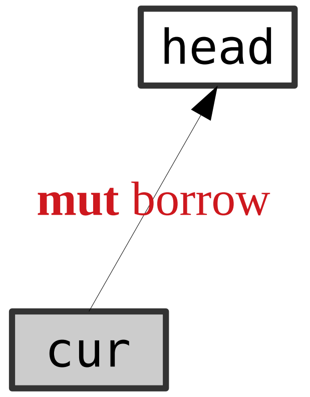
Using region

```
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


}
```



Using region

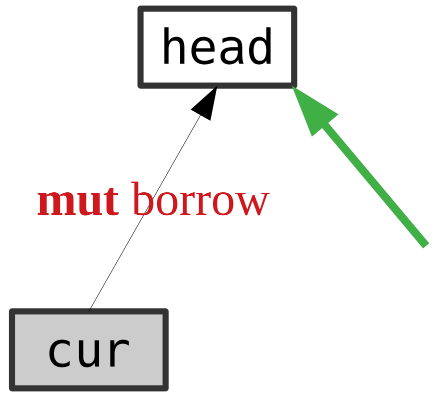
```
fn list_not(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>>
{
    let mut cur = &mut head;

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

    head

}
```



Not using anymore

Using region

```
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

}
```

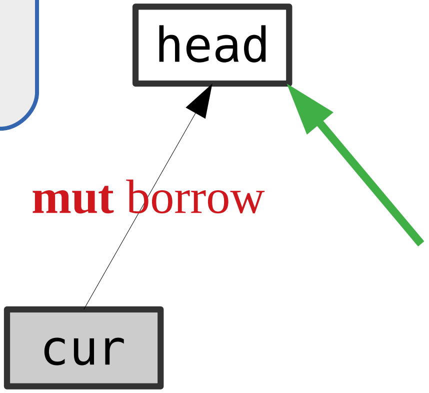


Not using anymore

Using region

```
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

}
```

`**cur**' only used here



Not using anymore

Borrow

```
P0 : let r: &T
P1 : r = &x;
```

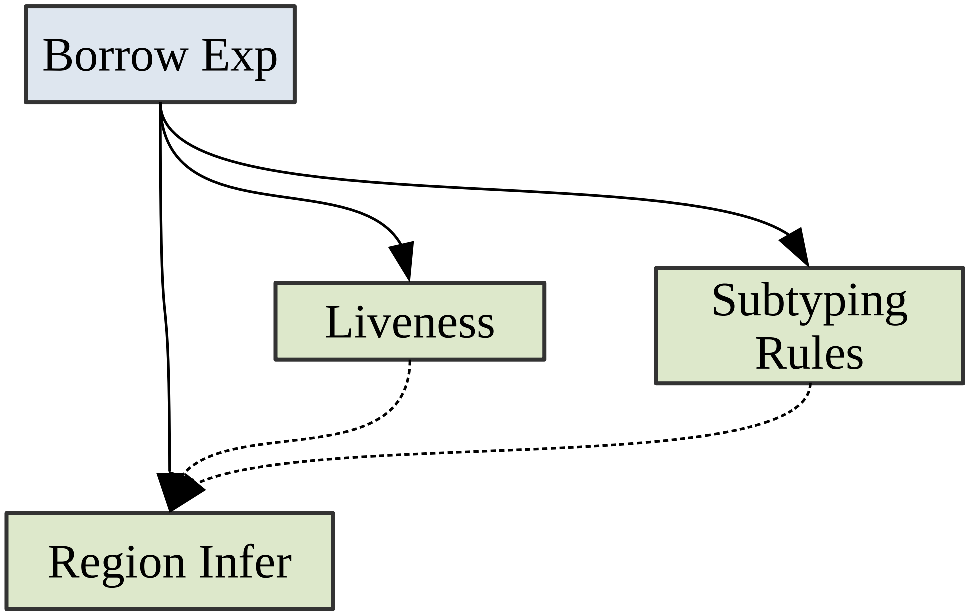
Borrow

```
P0 : let r: &'0 T
P1 : r = &'1 x;
```

Borrow Exp

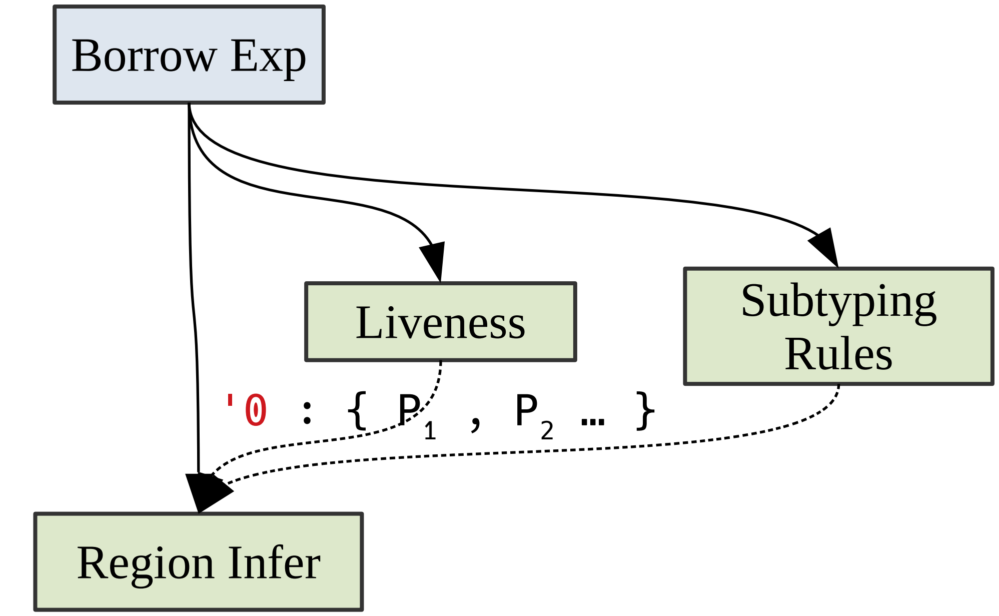
Borrow

```
P0 : let r: &'0 T
P1 : r = &'1 x;
```



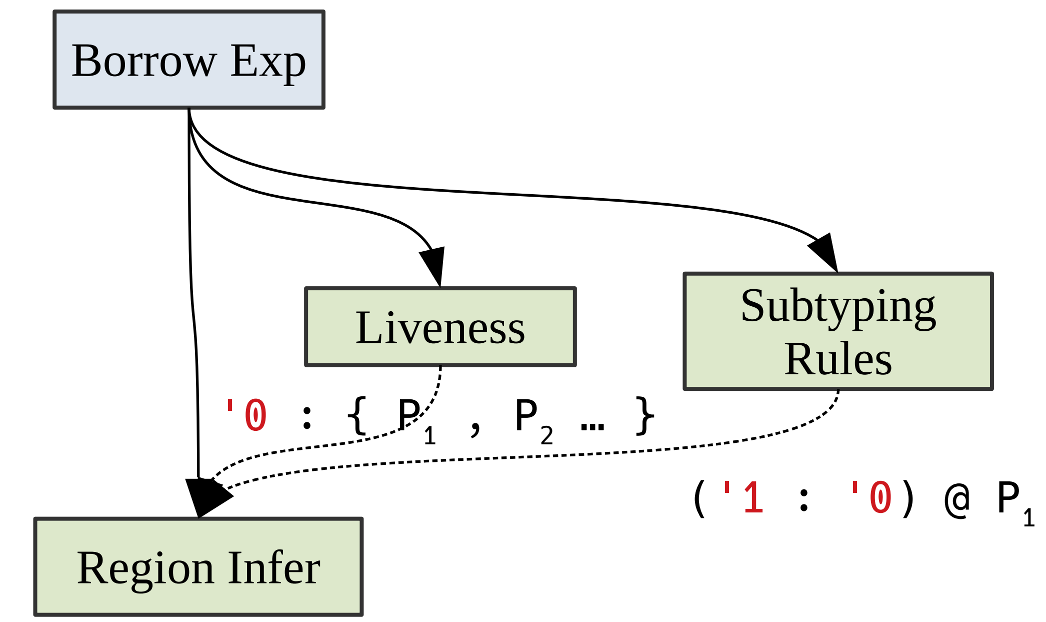
Borrow

```
P0 : let r: &'0 T
P1 : r = &'1 x;
```



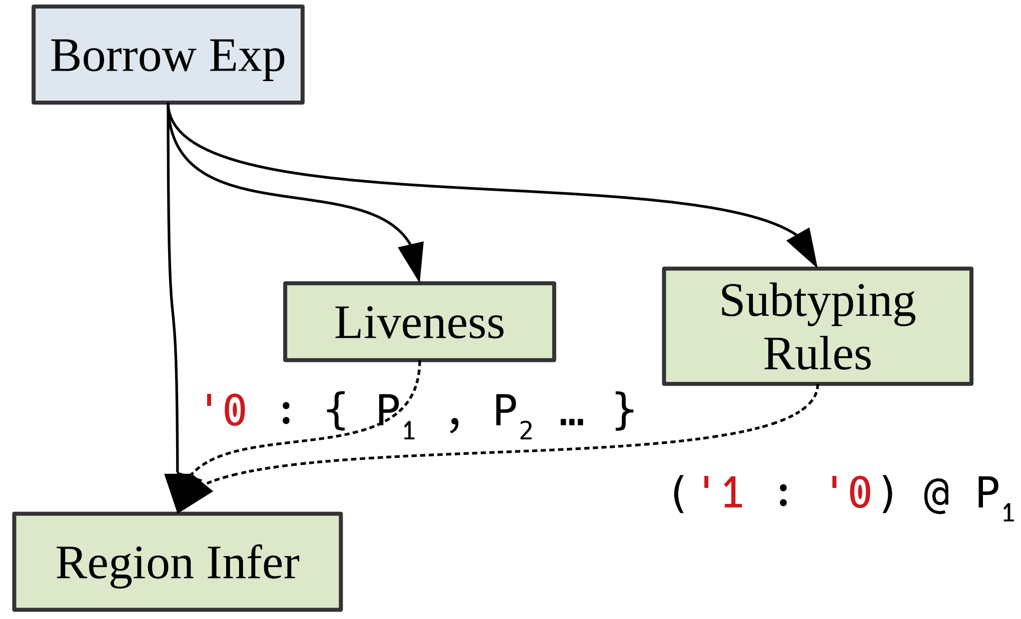
Borrow

```
P0 : let r: &'0 T
P1 : r = &'1 x;
```



Borrow

```
P₀ : let r: &'0 T
P₁ : r = &'1 x;
```



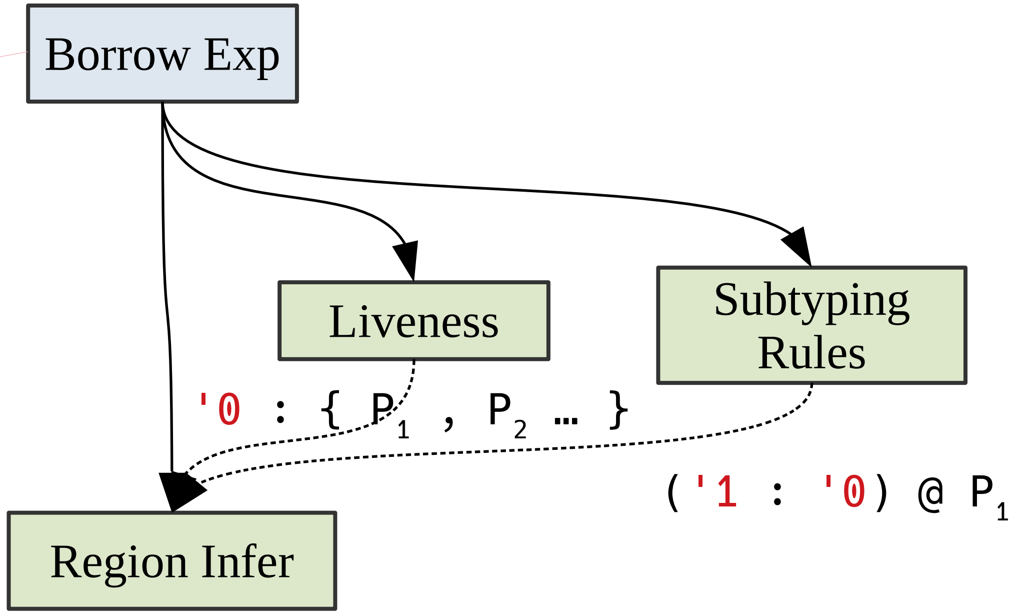
$'0 : \{ P_1 , P_2 \ldots \}$
$'1 : \{ P_1 , \ldots \}$

Borrow

Each Borrow expression will corresponding to each Loan

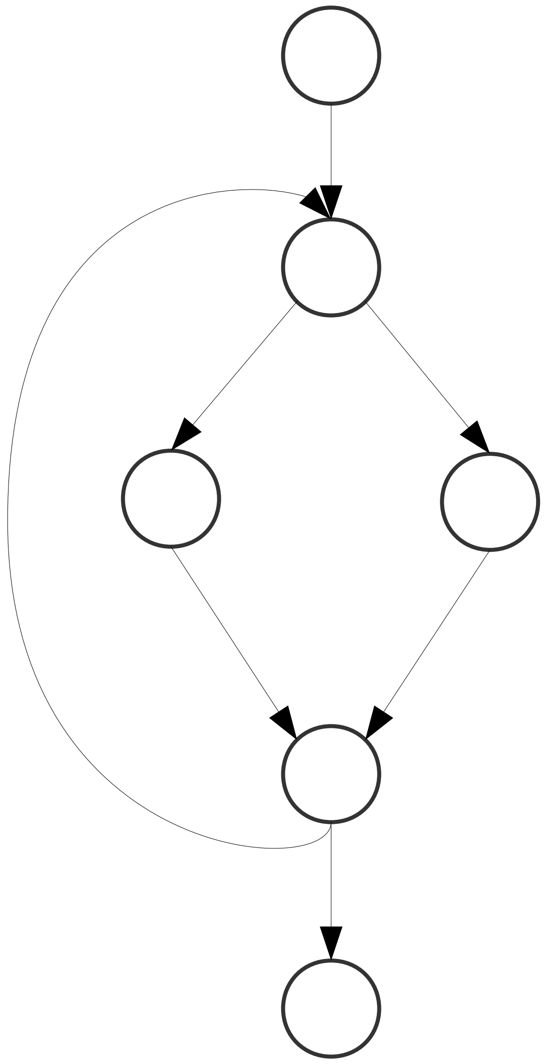
```
P₀ : let r: &'0 T
P₁ : r = &'1 x;
```

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

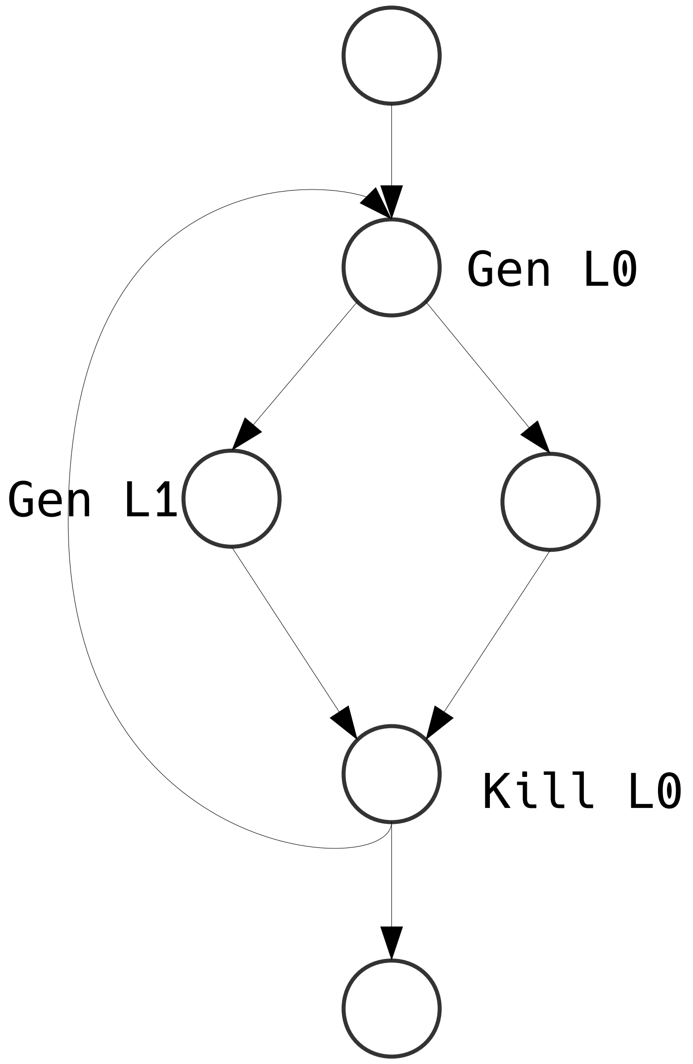


```
'0 : { P₁ , P₂ … }
'1 : { P₁ , … }
```

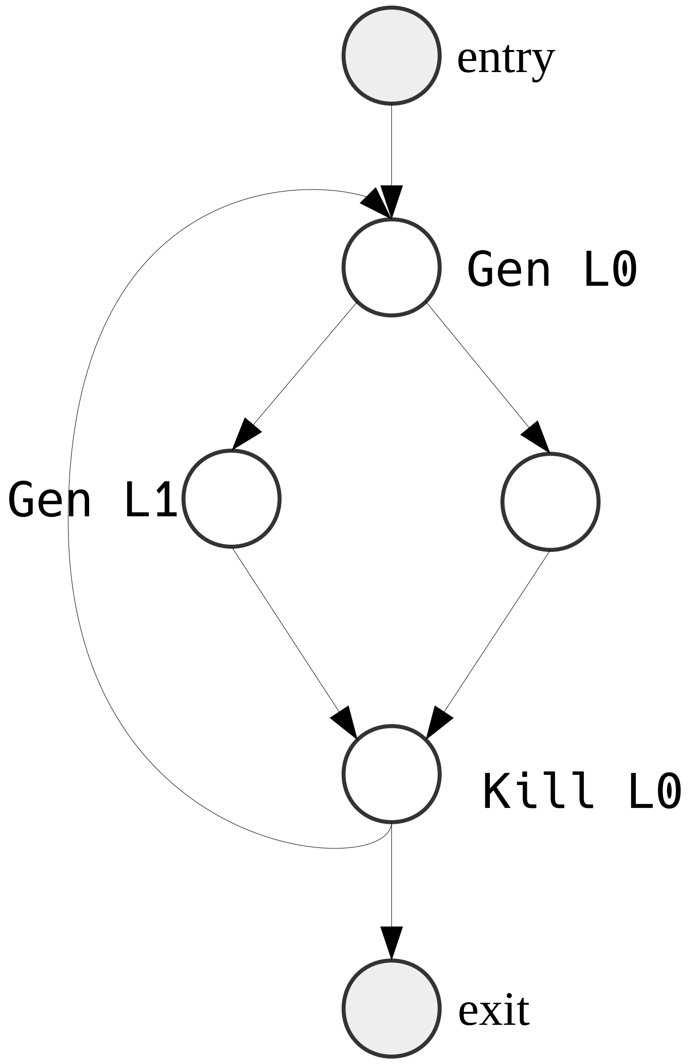
The Data Flow of the Loan



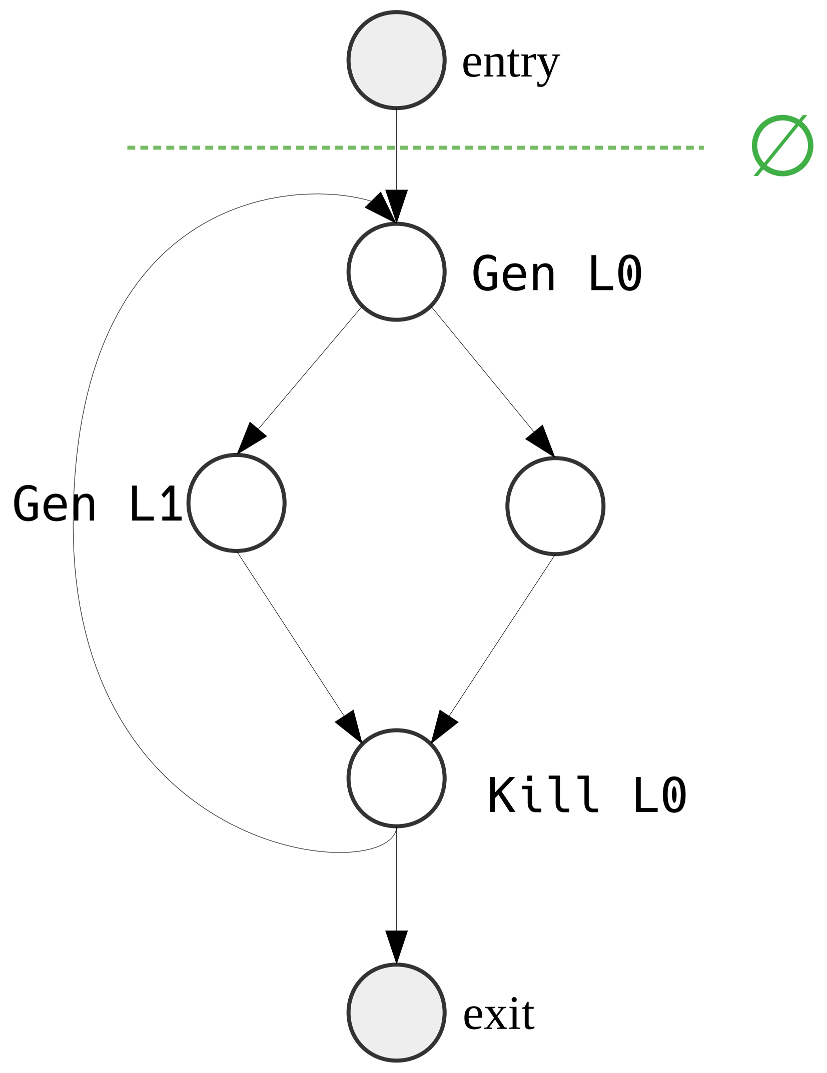
The Data Flow of the Loan



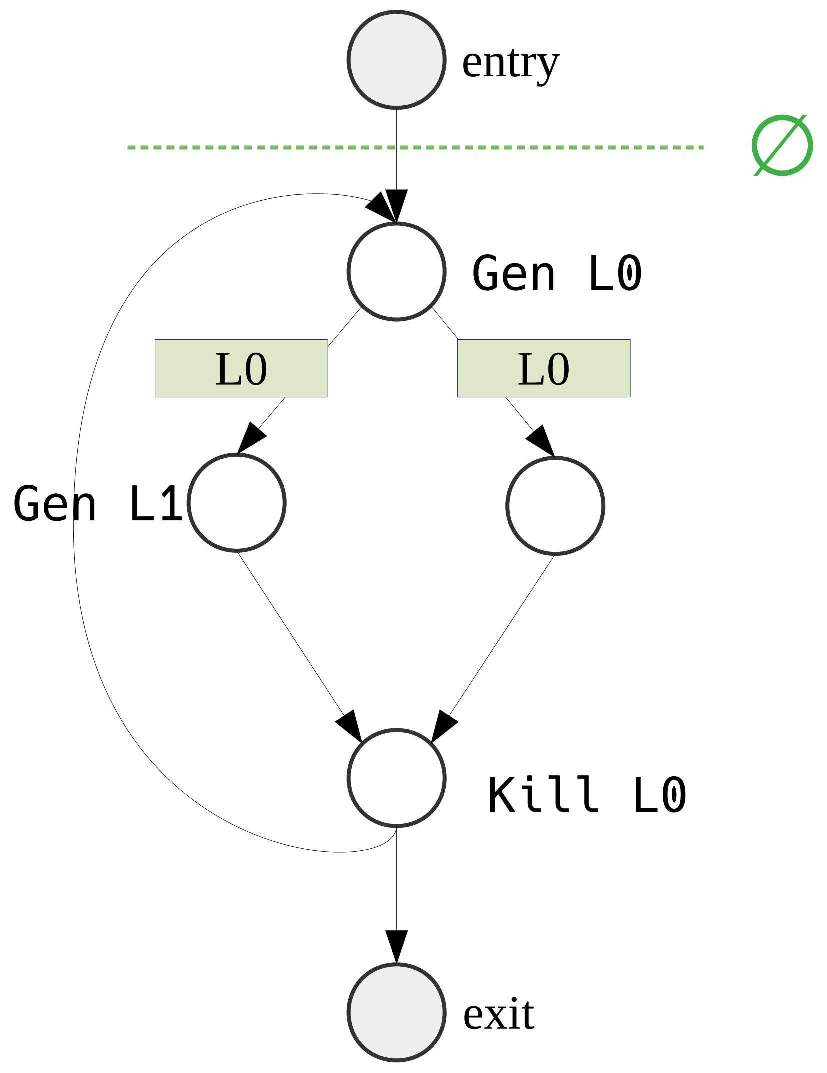
The Data Flow of the Loan



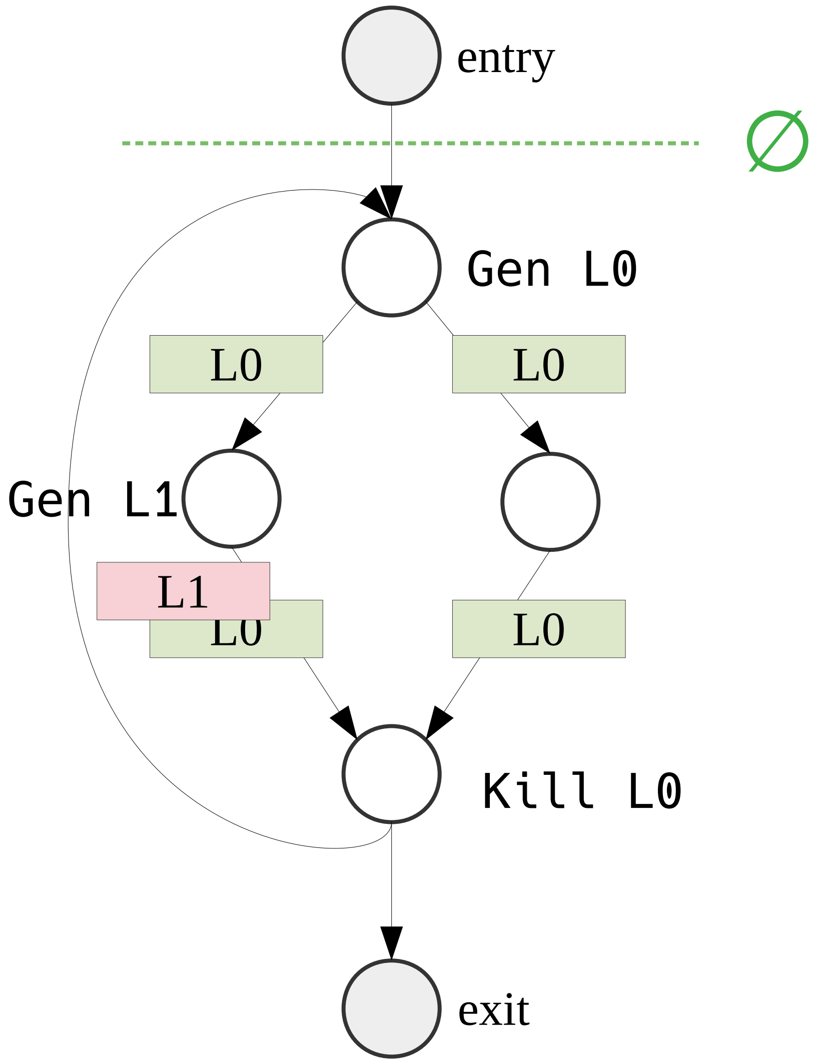
The Data Flow of the Loan



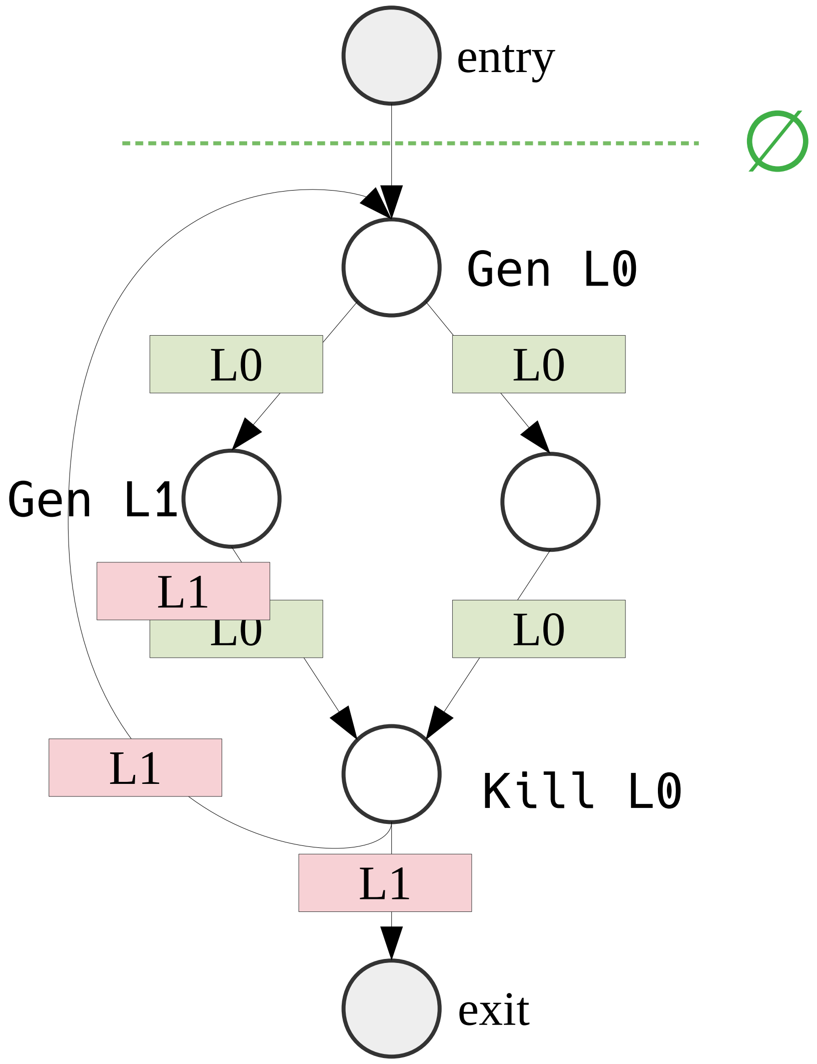
The Data Flow of the Loan



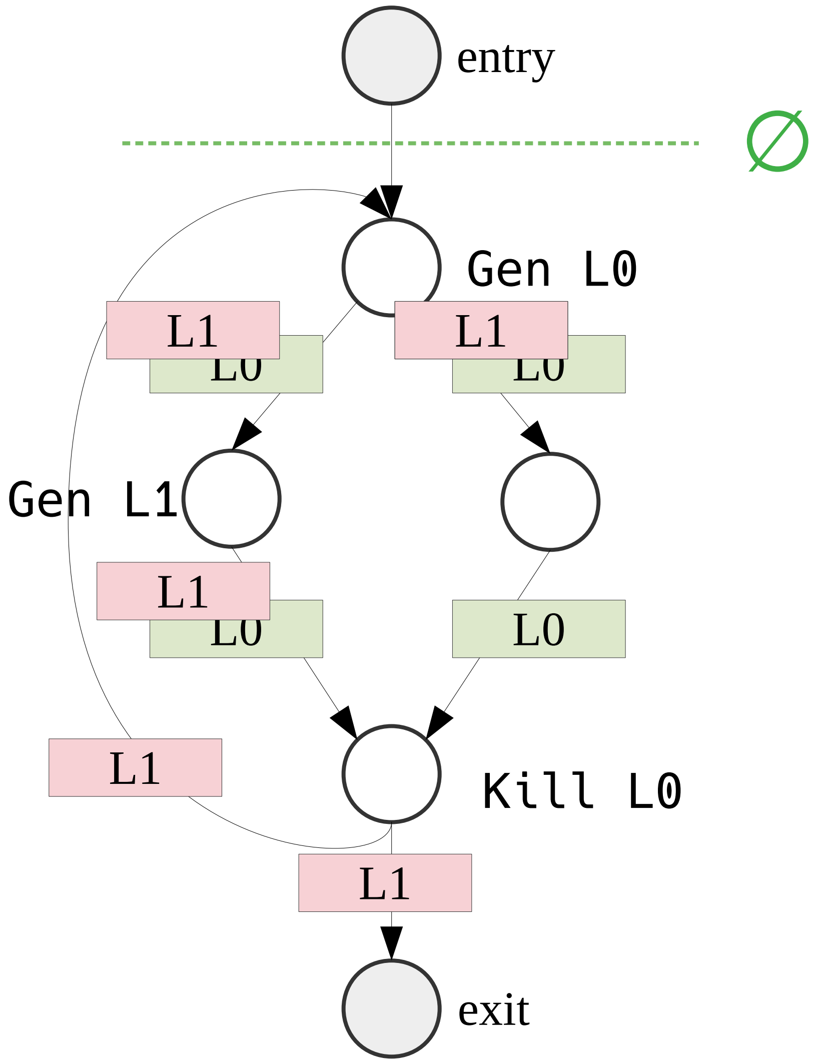
The Data Flow of the Loan



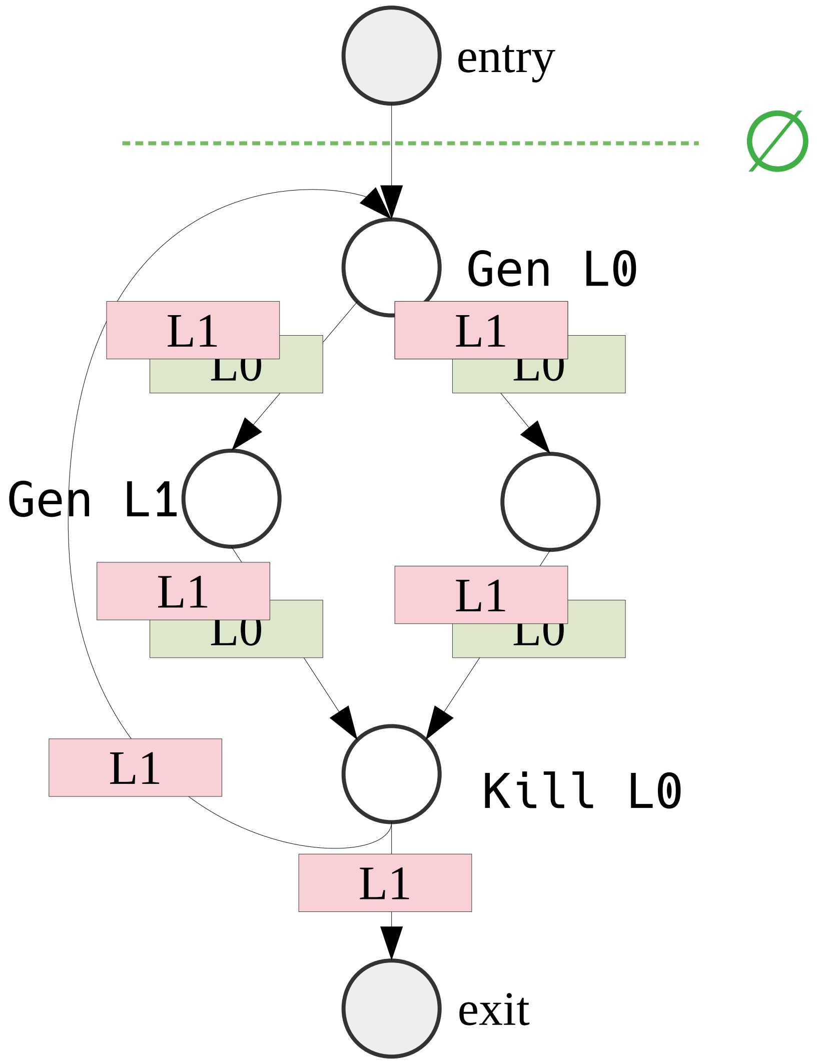
The Data Flow of the Loan



The Data Flow of the Loan

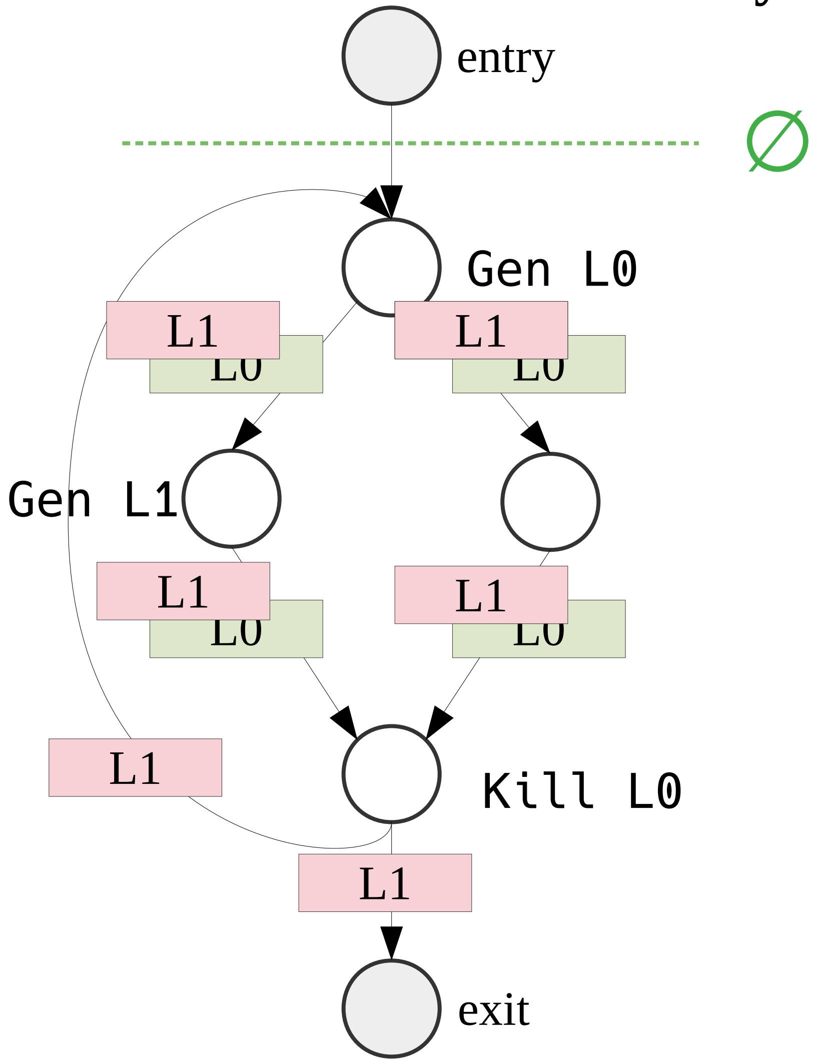


The Data Flow of the Loan



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 :

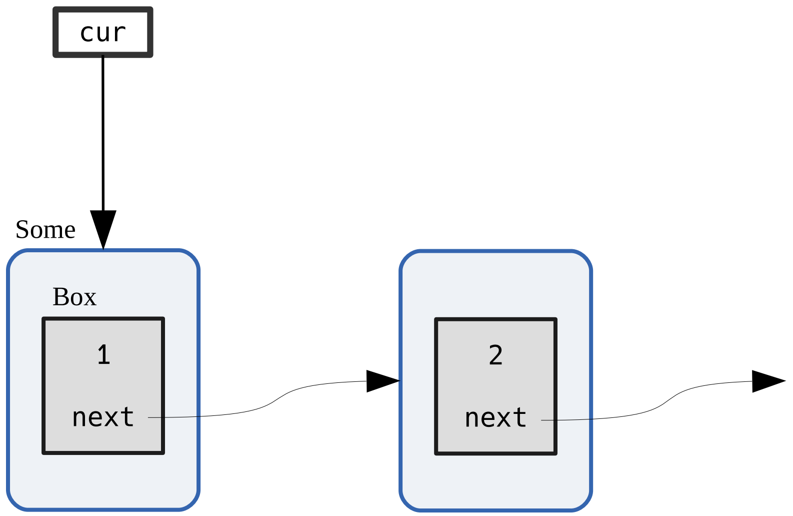
1) $\texttt{LV = Loan}_i\texttt{ . path}$
2) $\texttt{point} \notin \texttt{Loan}_i\texttt{ . region}$

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);
    }

}
```

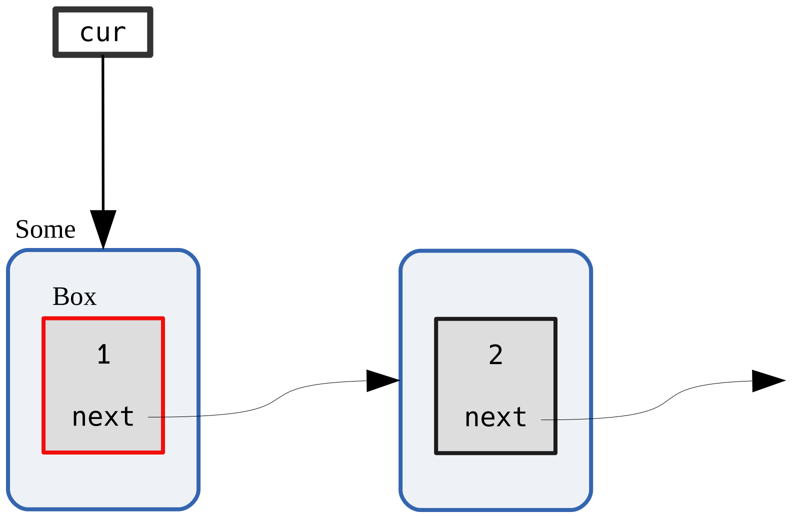
Example

```
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);
    }

}
```



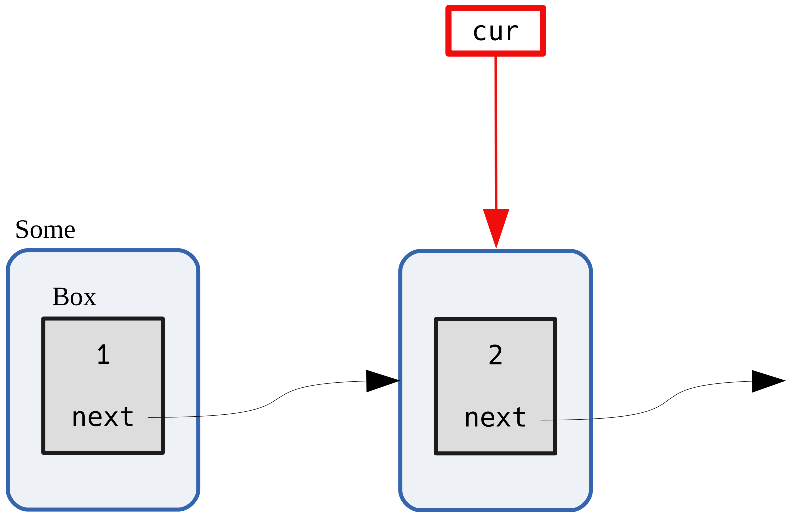
Example

```
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);
    }

}
```



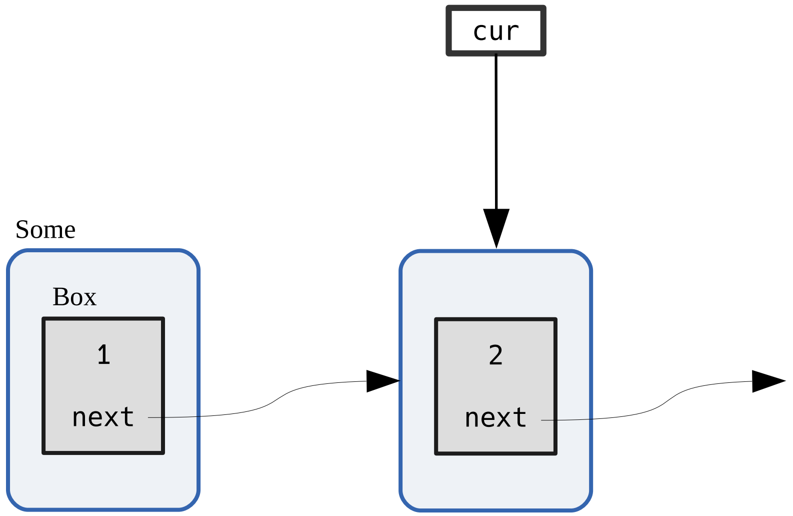
Example

```
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

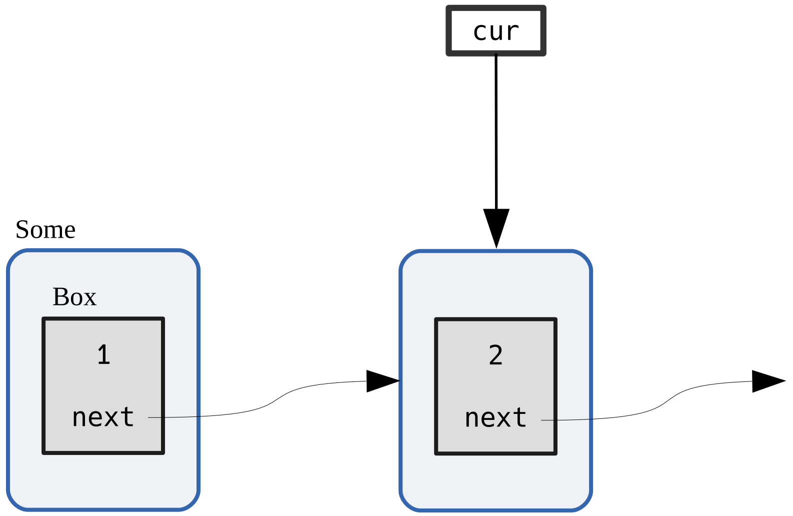
```
fn do_something(mut head: Option<Box<ListNode>>)
{
    let mut cur = &mut head;

    if let Some(nodeBox) = cur.as_mut() {

        cur = &mut nodeBox.next;

        println!("{:?}", nodeBox);
    }

}
```



Example

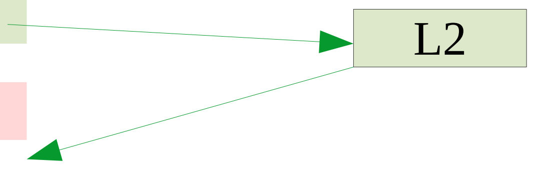
```
fn do_something(mut head: Option<Box<ListNode>>)
{
    let mut cur = &mut head;

    if let Some(nodeBox) = cur.as_mut() {

        cur = &mut nodeBox.next;

        println!("{:?}", nodeBox);
    }

}
```

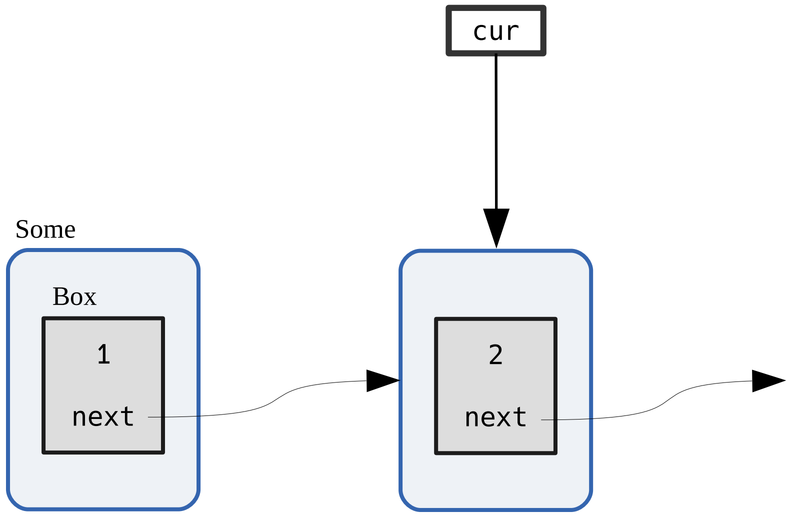
L2



Example

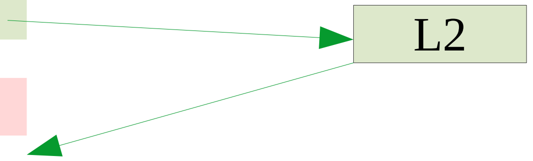
```
fn do_something(mut head: Option<Box<ListNode>>)
{
    let mut cur = &mut head;

    if let Some(nodeBox) = cur.as_mut() {

        cur = &mut nodeBox.next;

        println!("{:?}", nodeBox);
    }

}
```





Example

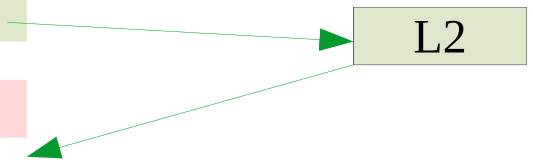
```
fn do_something(mut head: Option<Box<ListNode>>)
{
    let mut cur = &mut head;

    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);
    }

}
```

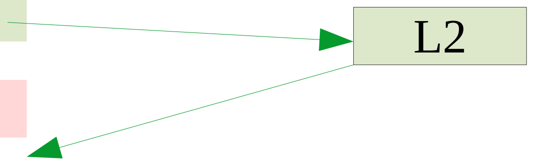


Why `**L2'** live at this point ?

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

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

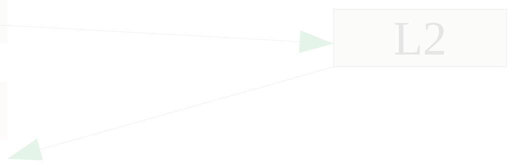
It is rejected in the current borow checker, but it is accepted by the Polonius borrow checker in the future.

Example

參考題目

Leetcode : remove linked list elements

```
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

It is rejected in the current borow checker, but it is accepted by the Polonius borrow checker in the future.

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

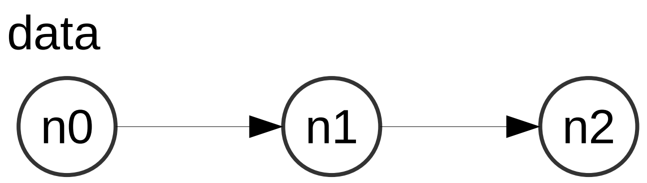
- 每次都往前推論一步，直到每個節點都達到穩態即推論完畢
- 抽出每次推論的一小步 (Induction)

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)

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

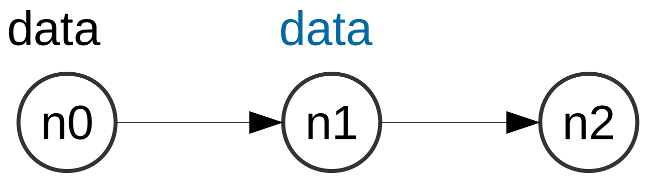


Idea

- 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)

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

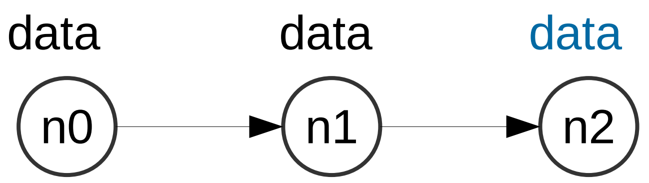


Idea

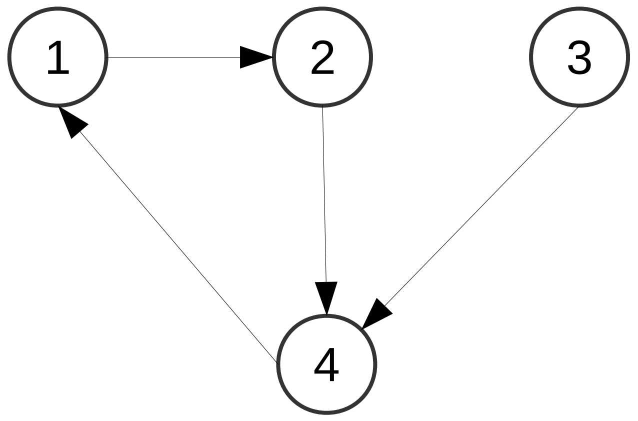
- 每次都往前推論一步，直到每個節點都達到穩態即推論完畢

- 抽出每次推論的一小步 (Induction)

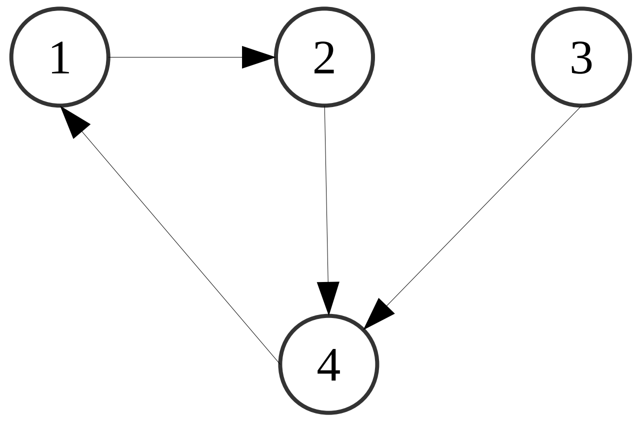
$$\frac{N(a,x) \quad e(a,b)}{N(b,x)}$$

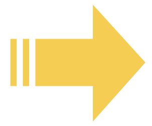


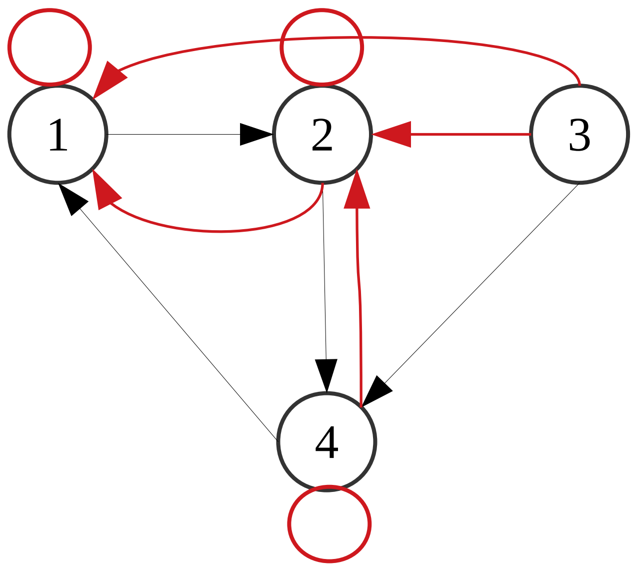
Example . Transitive Closure



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 iteartion.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 iteartion.changed() {

   ...

}


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

Writing Rules here

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));
}
```

$$\frac{\begin{array}{c} e(a,b) \\ e(b,c) \end{array}}{e(a,c)} \Rightarrow \boxed{\frac{e(a,b)}{r(b,a)}} \quad \boxed{\frac{\begin{array}{c} r(b,a) \\ e(b,c) \end{array}}{e(a,c)}}$$

Implementation – Writing Rules

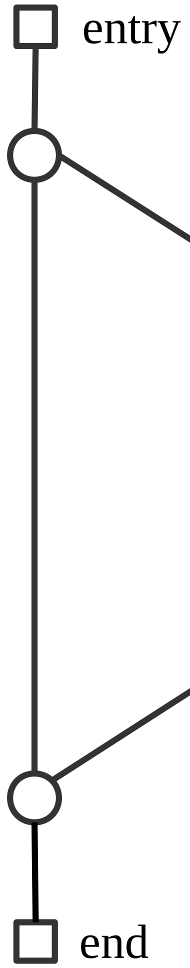
```
while iteration.changed() {
    // reverse edges for mapping
    v_redges.from_map(&v_edges, |&(a, b)| (b, a));

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

$$\frac{\begin{array}{c} e(a,b) \\ e(b,c) \end{array}}{e(a,c)} \quad \Longrightarrow \quad \frac{e(a,b)}{r(b,a)} \qquad \frac{\begin{array}{c} r(b,a) \\ e(b,c) \end{array}}{e(a,c)}$$

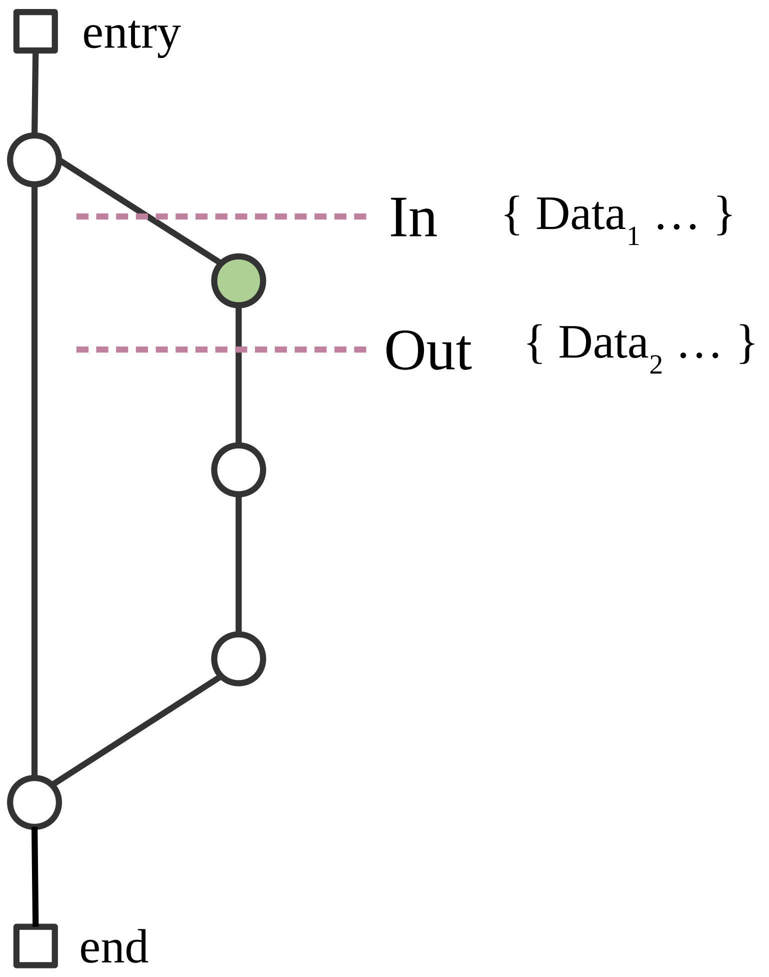
In Polonius

$$\frac{\begin{array}{l}\text{Require}(R, B, P)\\ \text{cfg-edges}(P, Q)\\ \text{!killed}(B, P)\\ \text{region\_live\_at}(R, Q)\end{array}}{\text{Require}(R, B, Q)}$$

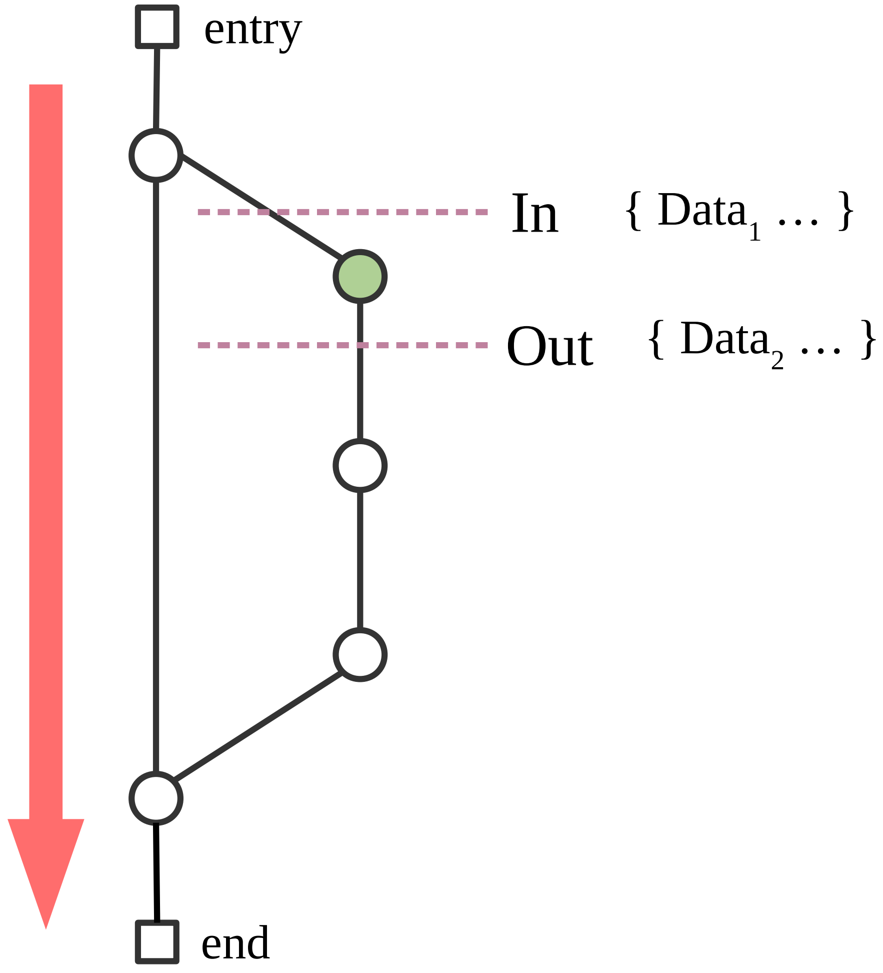
Data Flow Concepts <D, V, ∧, F>



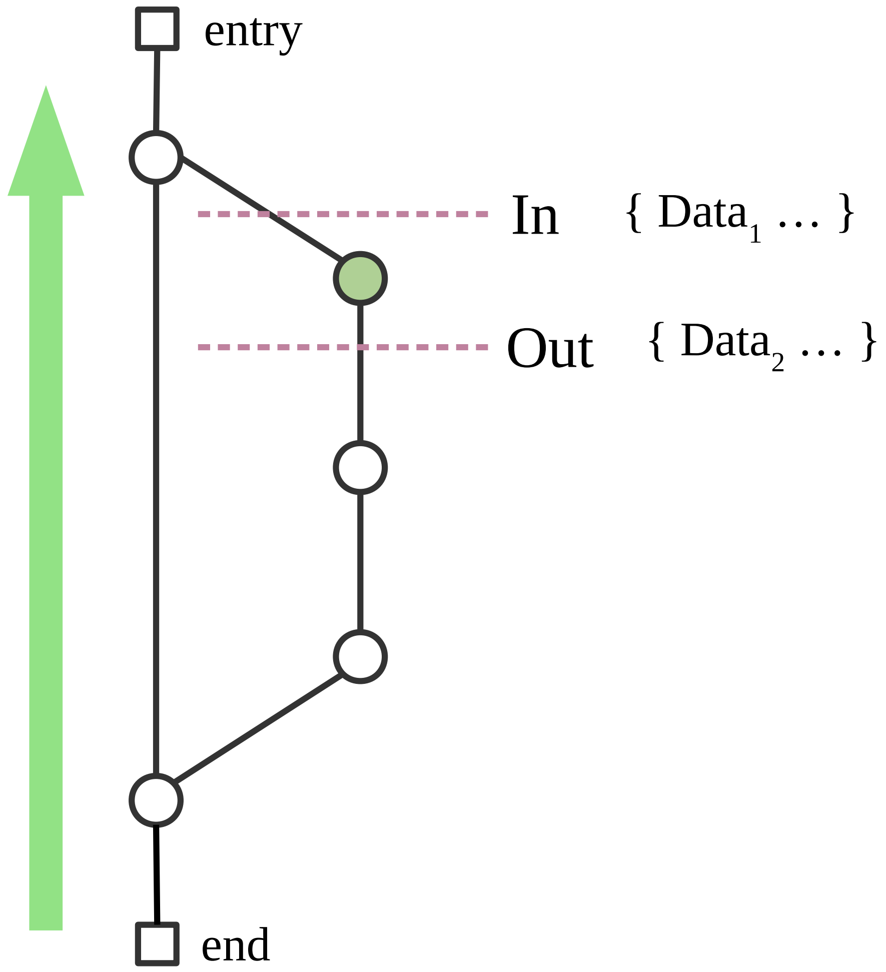
Data Flow Concepts <D, V, ∧, F>



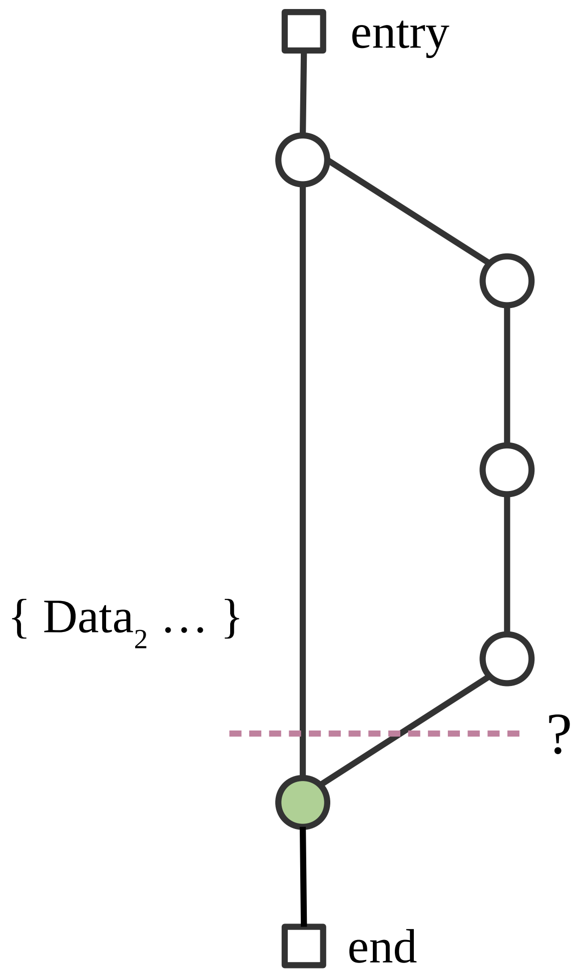
Data Flow Concepts <D, V, ∧, F>



Data Flow Concepts <**D**, V, ∧, F>

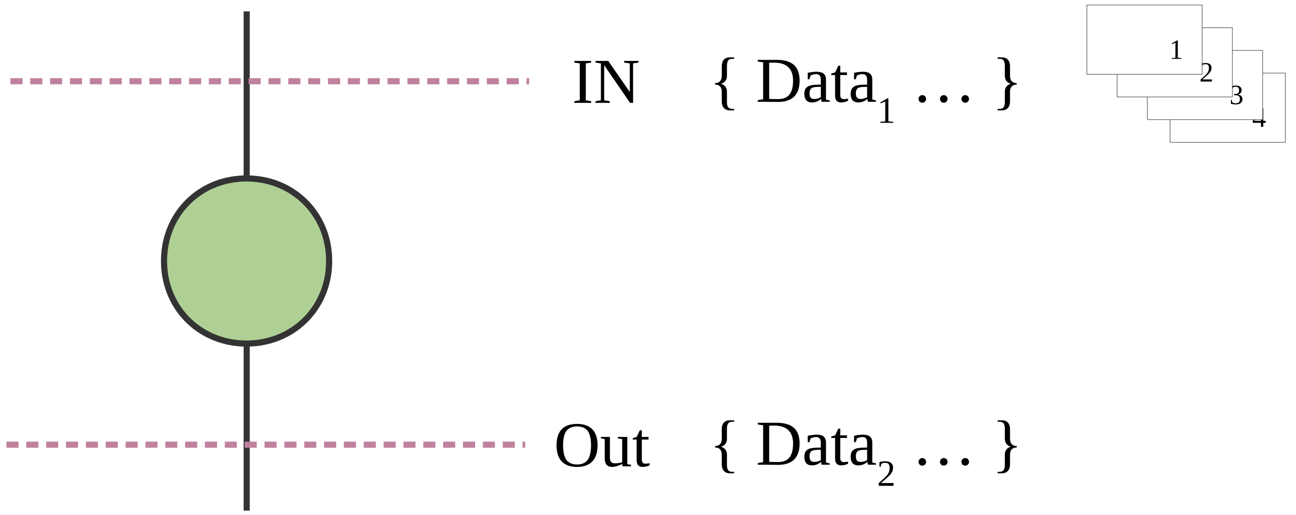


Data Flow Concepts <D, V, ∧, F>



Meet operator can be union, intersenction, as long as it can make (V, ∧) semilattice.

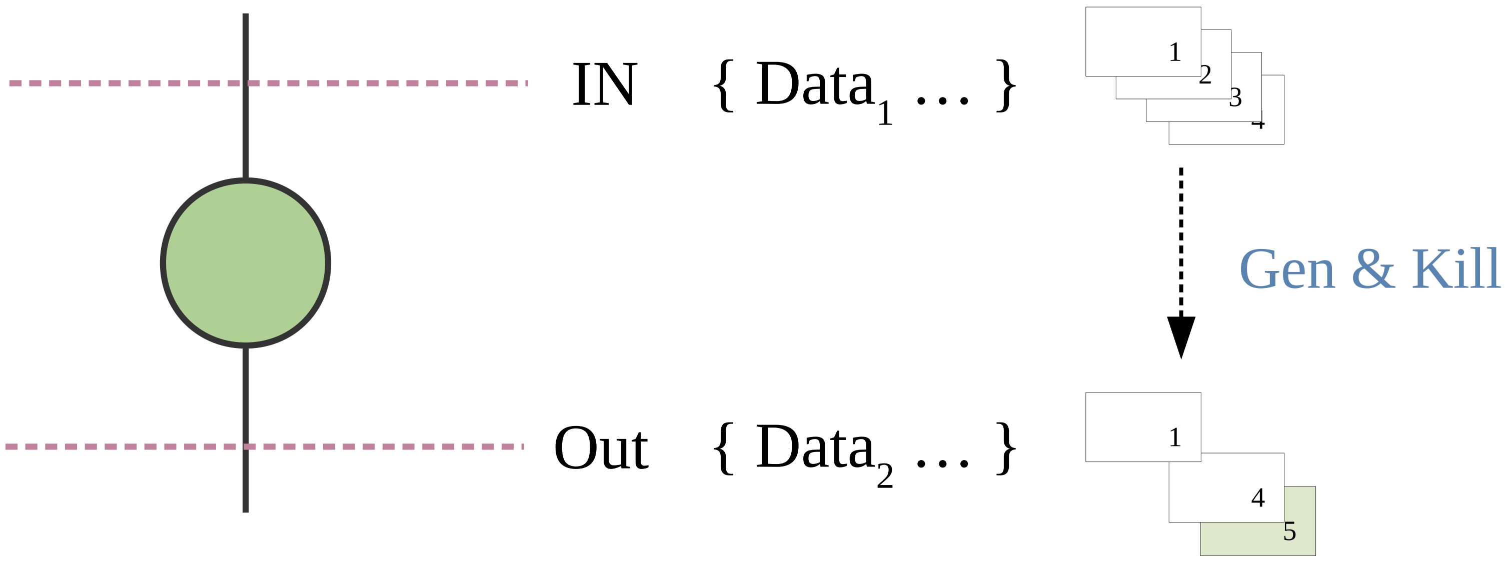
Data Flow Concepts <D, V, ∧, **F**>



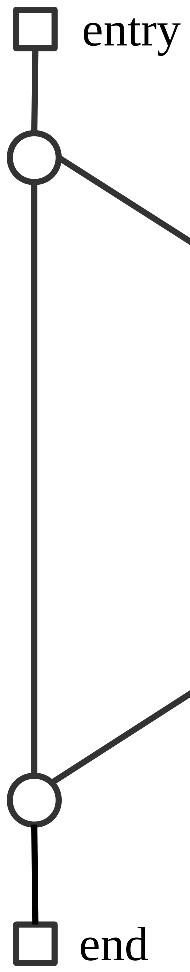
IN $\{ \text{Data}_1 \ldots \}$

Out $\{ \text{Data}_2 \ldots \}$

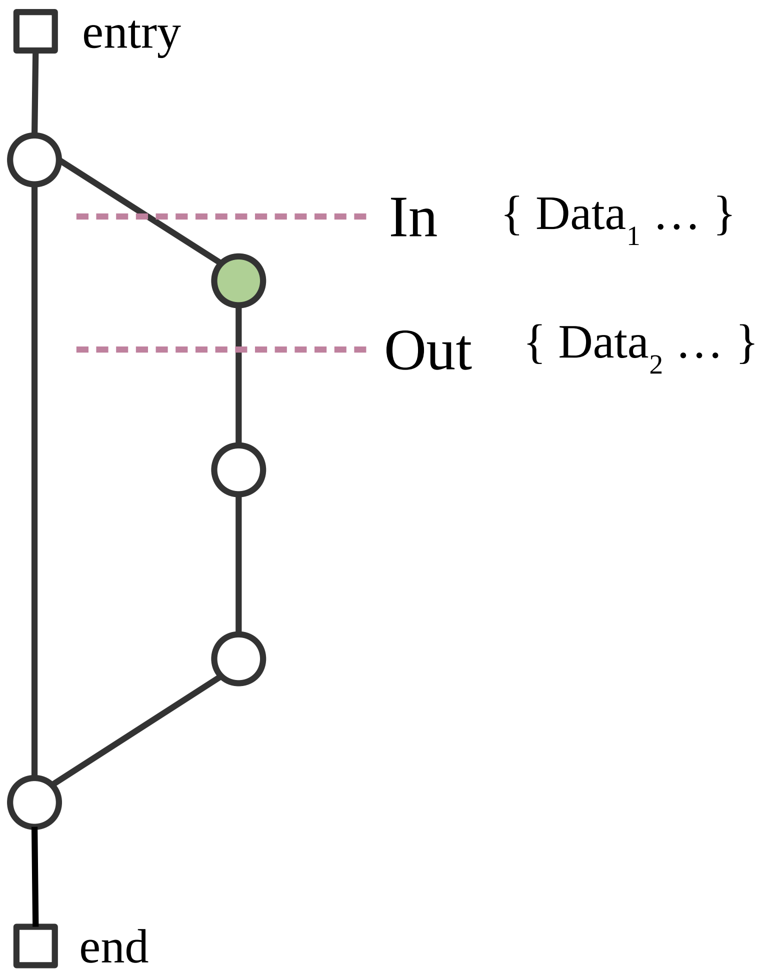
Data Flow Concepts <D, V, ∧, **F**>



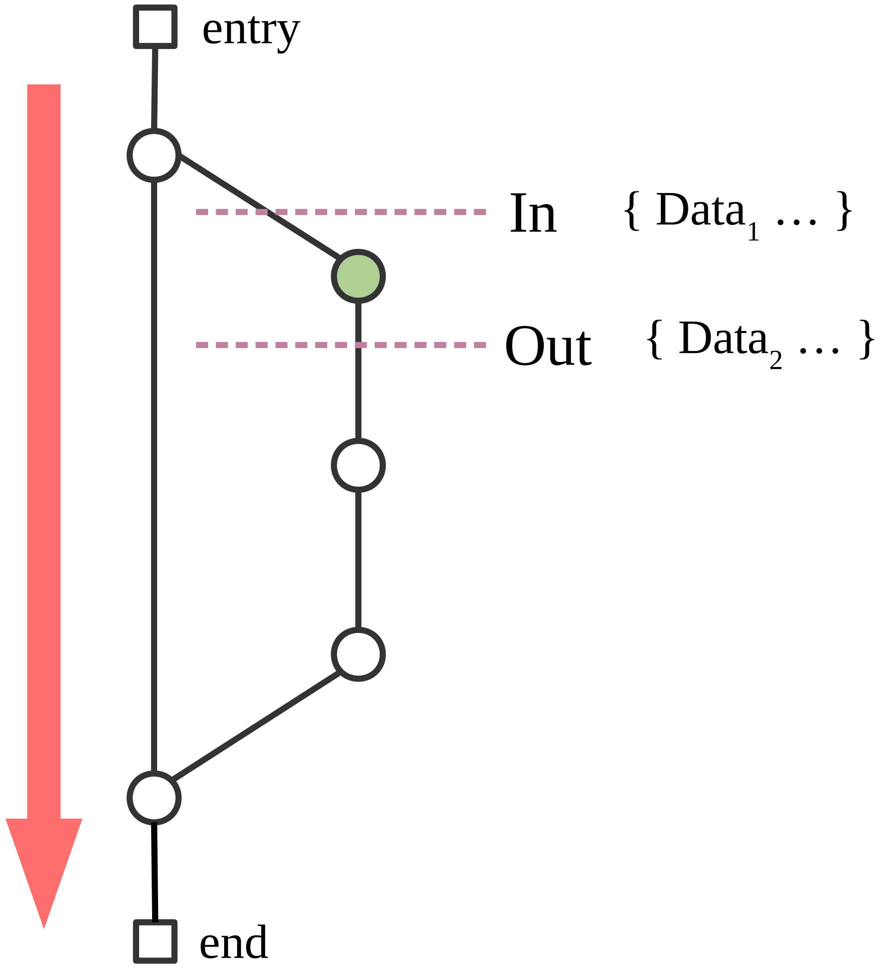
Data Flow Concepts <D, V, ∧, F>



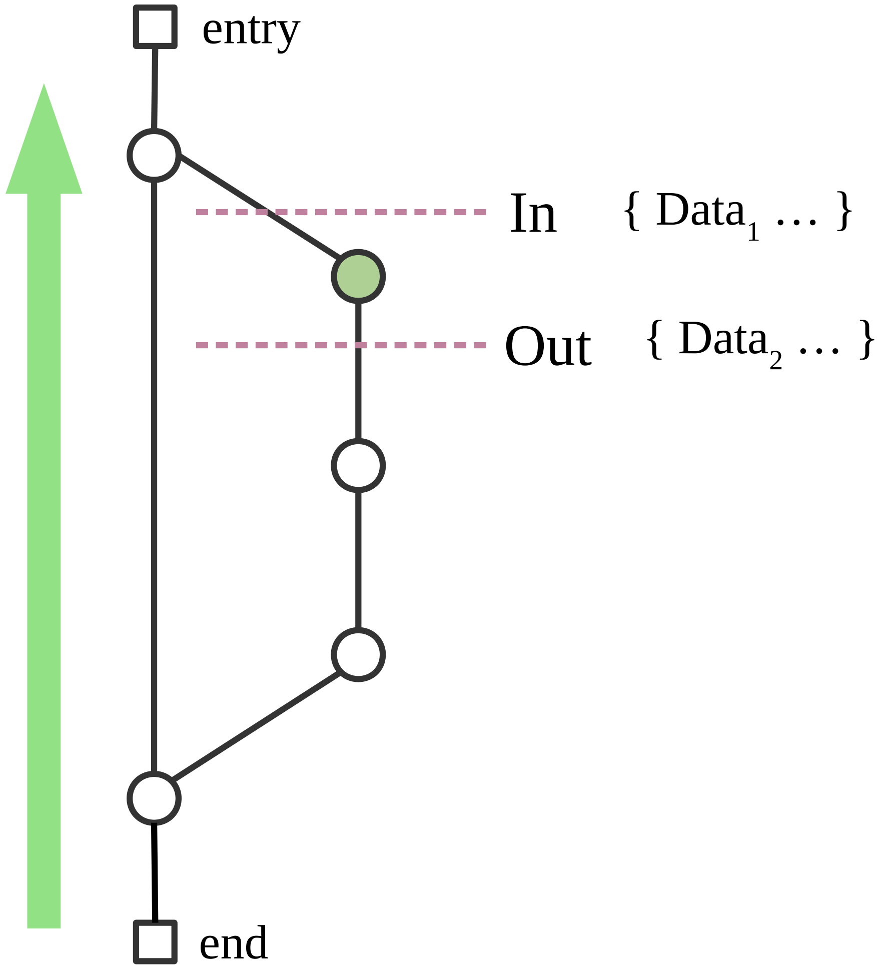
Data Flow Concepts <D, V, ∧, F>



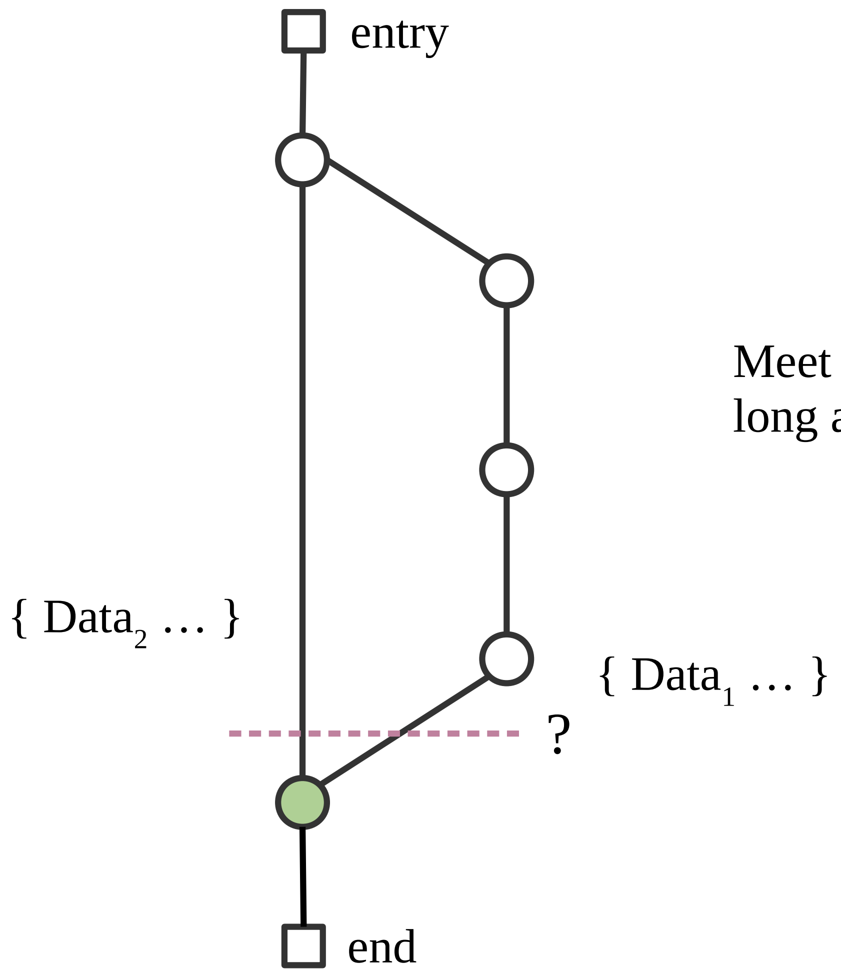
Data Flow Concepts <**D**, V, ∧, F>



Data Flow Concepts <**D**, V, ∧, F>

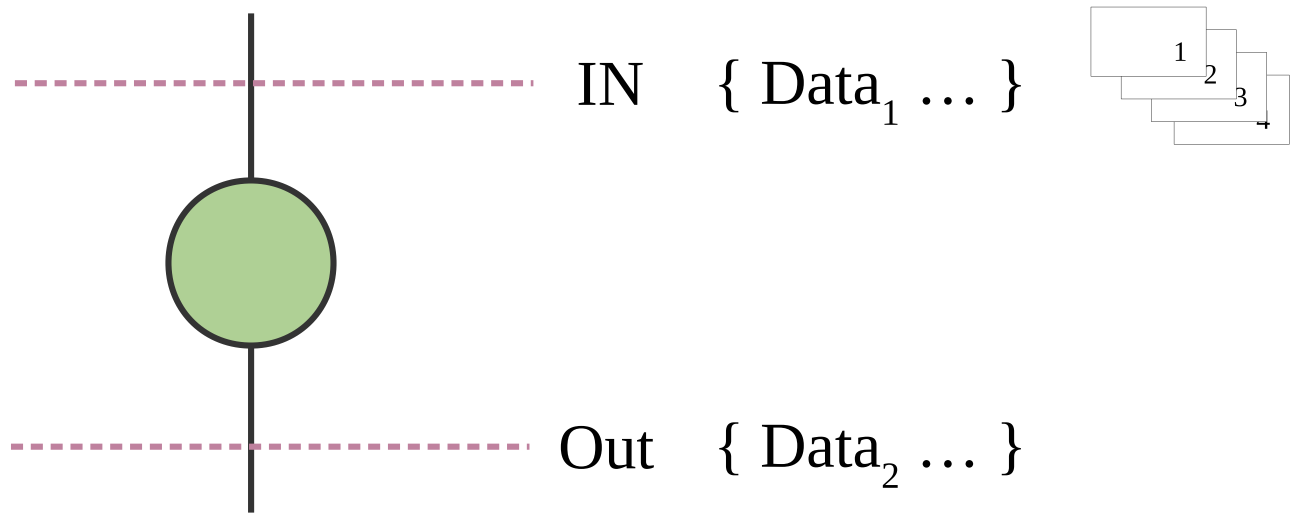


Data Flow Concepts <D, V, ∧, F>

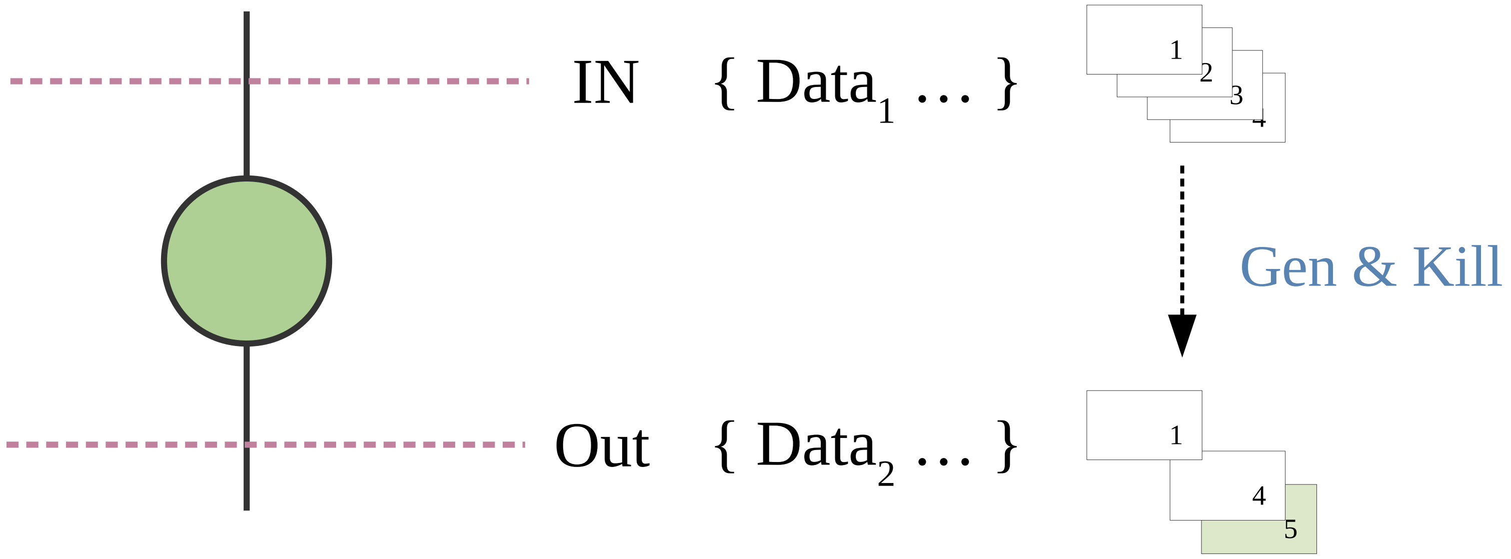


Meet operator can be union, intersenction, as long as it can make (V, ∧) semilattice.

Data Flow Concepts <D, V, ∧, **F**>



Data Flow Concepts <D, V, ∧, **F**>


IN
{ Data$_1$ … }
1
2
3
4
Gen & Kill
Out
{ Data$_2$ … }
1
4
5