# 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)
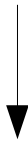
# Compilation of Rust

Rust Code

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

let mut p = &foo;

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

println!("{}", *p);
```
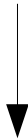
# Compilation of Rust
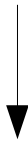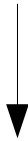
Rust Code

HIR

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

let mut p = &foo;

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

println!("{}", *p);
```
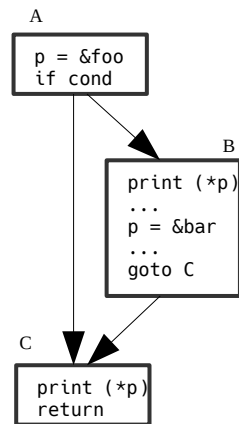
# Compilation of Rust

Rust Code

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

let mut p = &foo;

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

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

↓

HIR

↓

MIR

# 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);
```

HIR

MIR

A
```
p = &foo
if cond
```

B
```
print (*p)
...
p = &bar
...
goto C
```

C
```
print (*p)
return
```

Control Flow Graph

# Compilation of Rust



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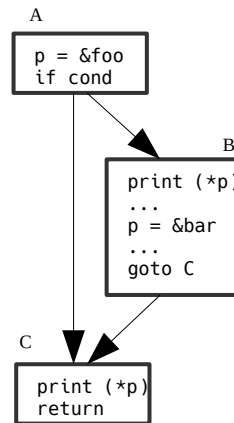
```
let foo: T = Foo {};
let bar: T = Bar {};

let mut p = &foo;

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

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

HIR

MIR

A
```
p = &foo
if cond
```

B
```
print (*p)
...
p = &bar
...
goto C
```

C
```
print (*p)
return
```
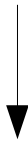
Control Flow Graph

Borrow checker
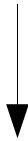
**Theorem**: Data Flow Analysis

**Tool**: Datafrog

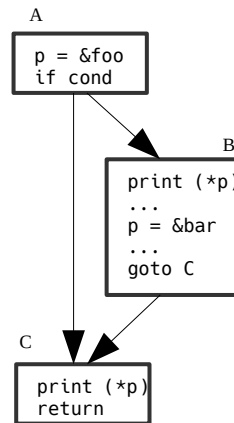# Compilation of Rust

Rust Code

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

let mut p = &foo;

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

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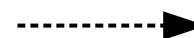
HIR

MIR

```
Documents

NLL RFC:
    2094-nll

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



Control Flow Graph

Borrow checker

**Theorem**: Data Flow Analysis

**Tool**: Datafrog

# Example

```rust
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

}
```
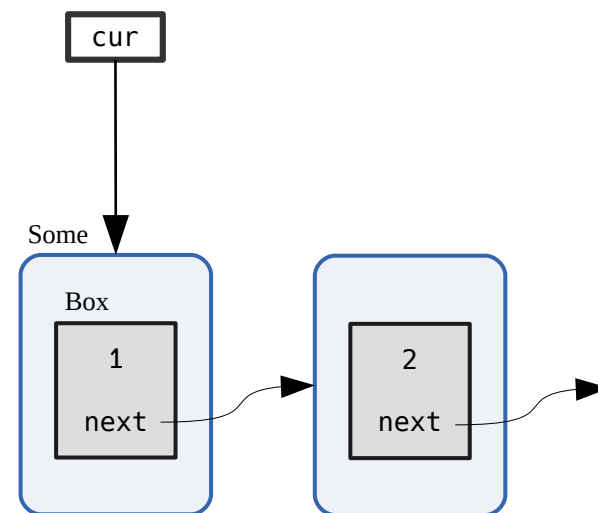
# Example

```rust
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

}
```
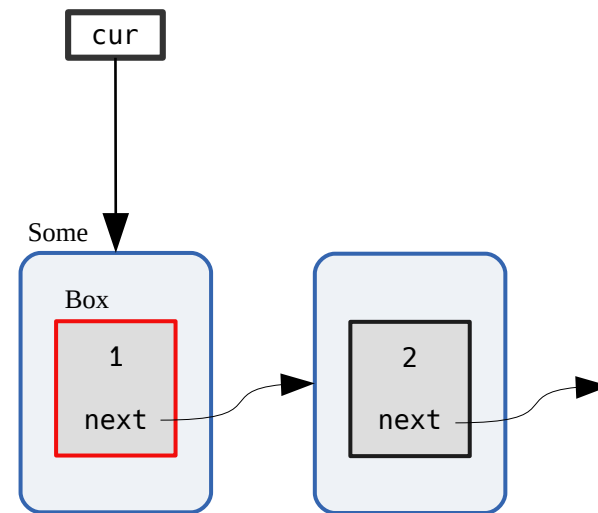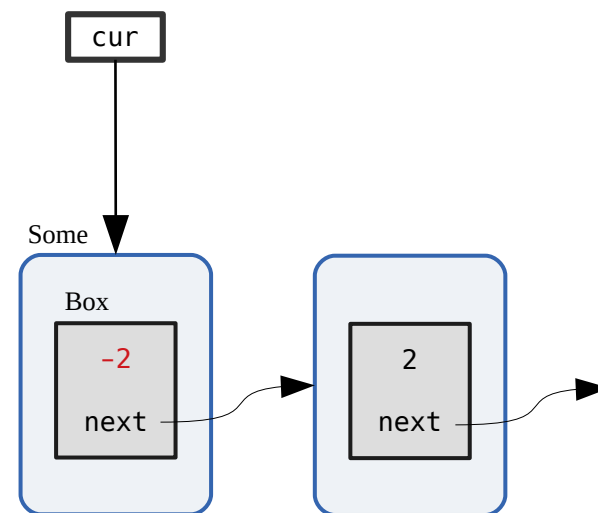
# Example

```
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


}
```
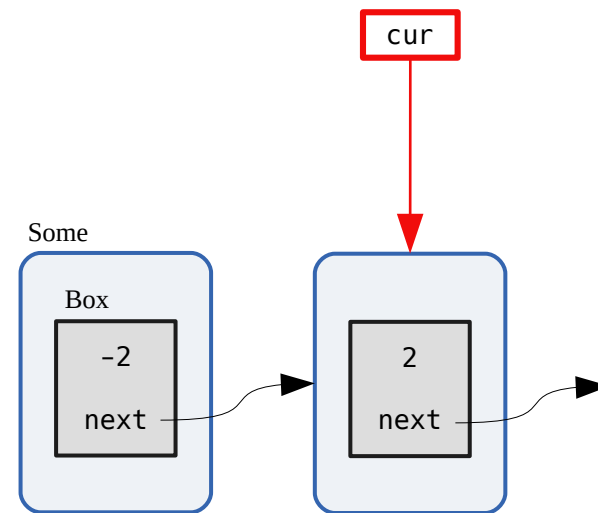
# Example

```
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

```rust
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

}
```
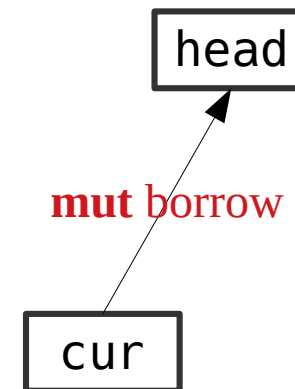
# 依照使用區間判斷

```rust
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


}
```
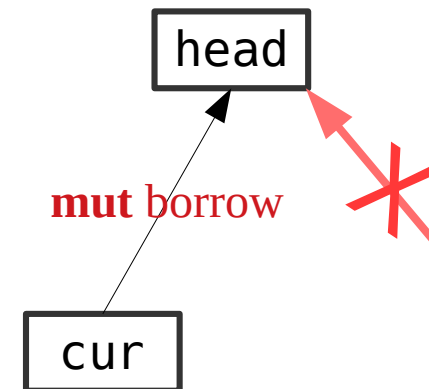
head

**mut** borrow

cur
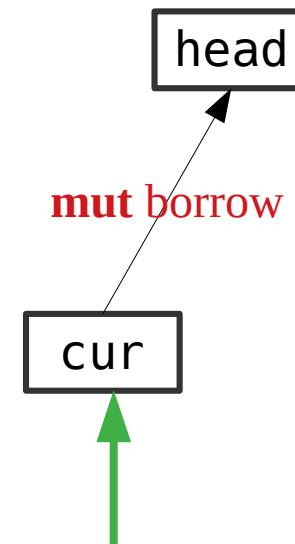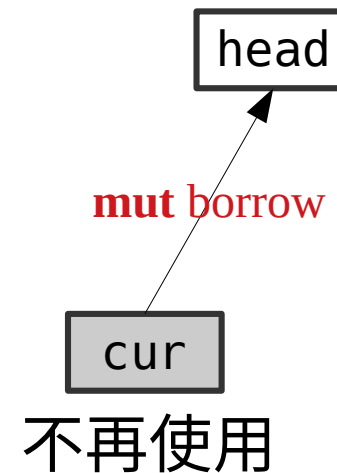
# 依照使用區間判斷

```
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

}
```



**mut** borrow

# 依照使用區間判斷

```rust
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


}
```

head

**mut** borrow

cur
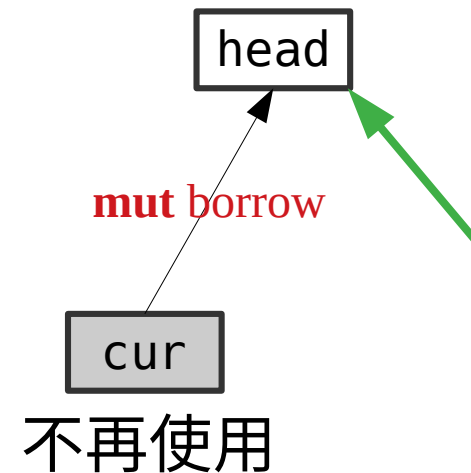
# 依照使用區間判斷

```
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

}
```

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

}
```

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;
```

`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

# Borrow

$$P_0 : \textbf{let}\ r:\ \&T$$
$$P_1 :\ r\ =\ \&x;$$

Borrow Exp

# Borrow

$P_0$ : **let** r: &$'0$ T
$P_1$ : r = &$'1$ x;

# Borrow

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

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

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



Borrow Exp

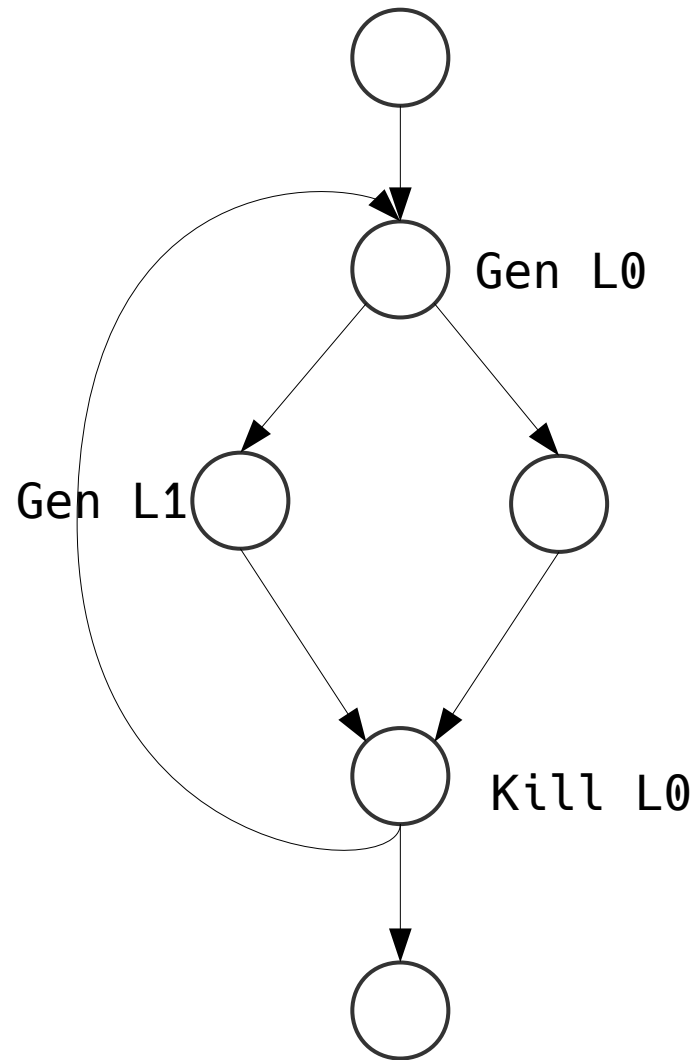Liveness
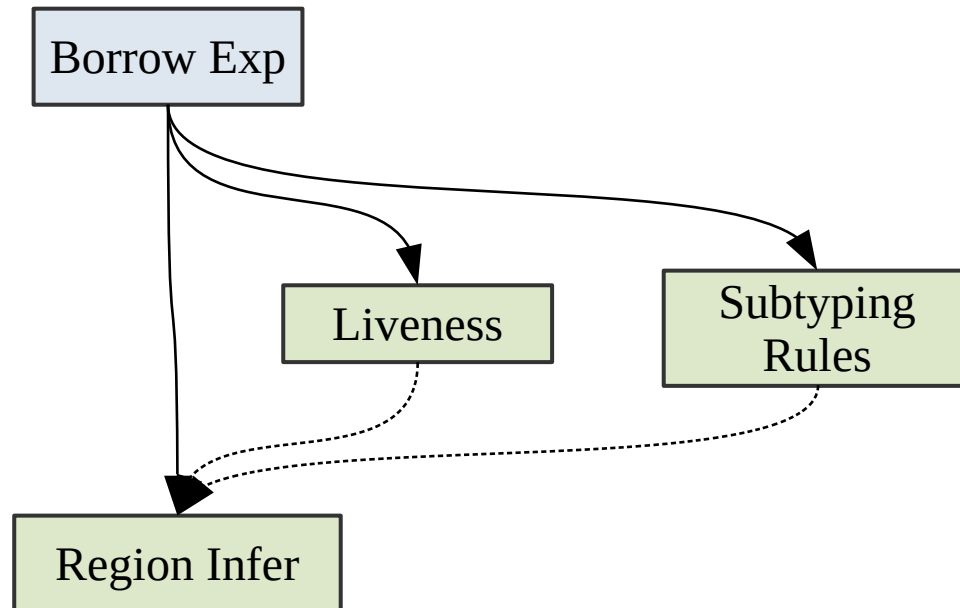
Subtyping Rules
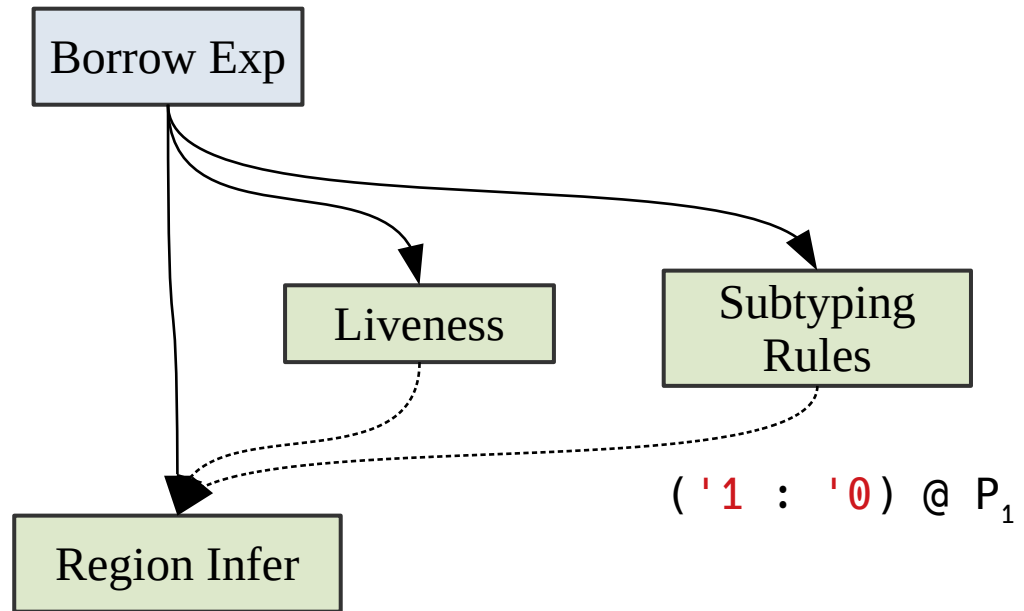
Region Infer

$('1 : '0)\ @\ P_1$

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

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

# Borrow

Each Borrow expression will corresponding to each Loan

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

$$P_0 : \textbf{let}\ r: \&'0\ T$$
$$P_1 : r = \&'1\ x;$$



Borrow Exp

Liveness

Subtyping Rules

Region Infer

$('1 : '0)\ @\ P_1$

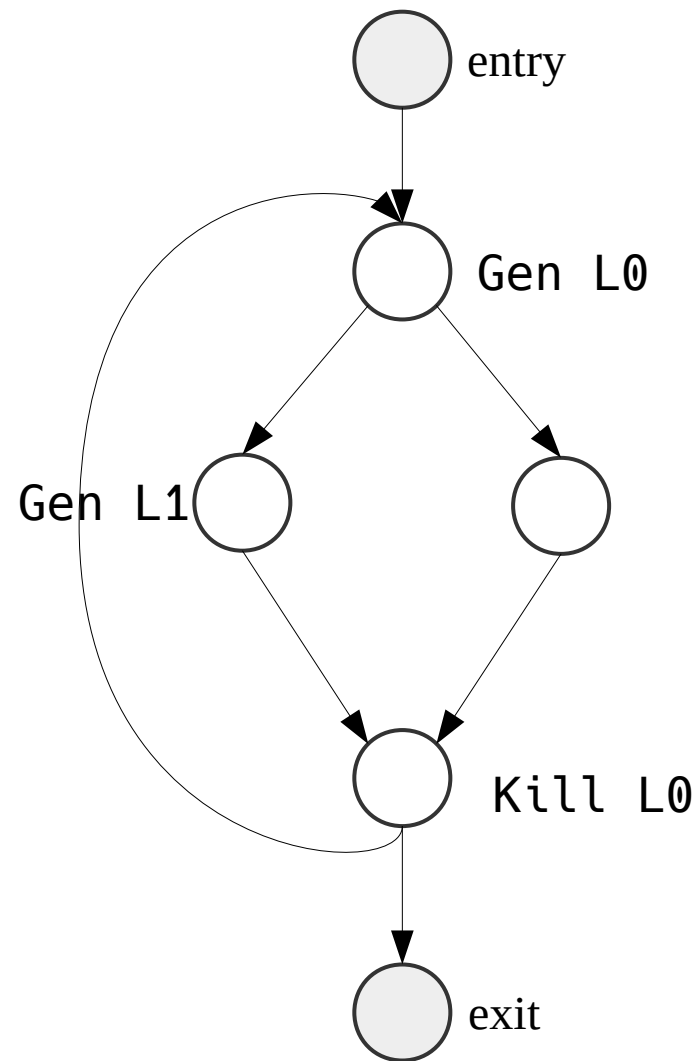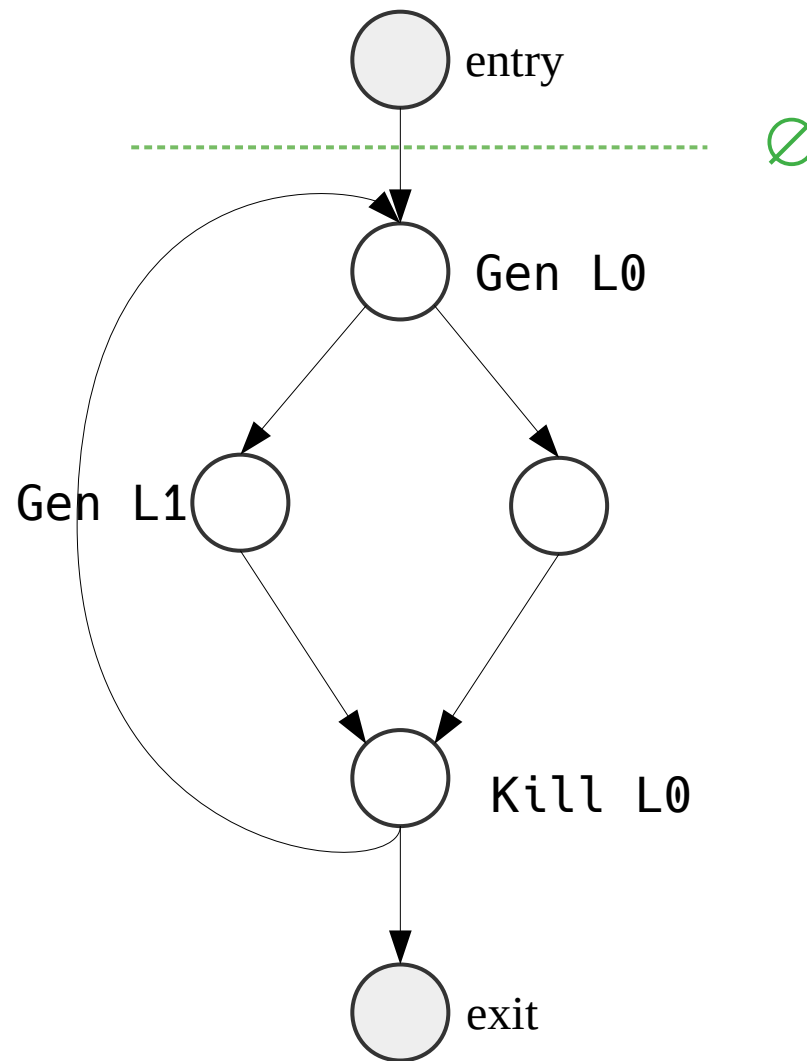$'0 : \{\ P_1\ ,\ P_2\ …\ \}$
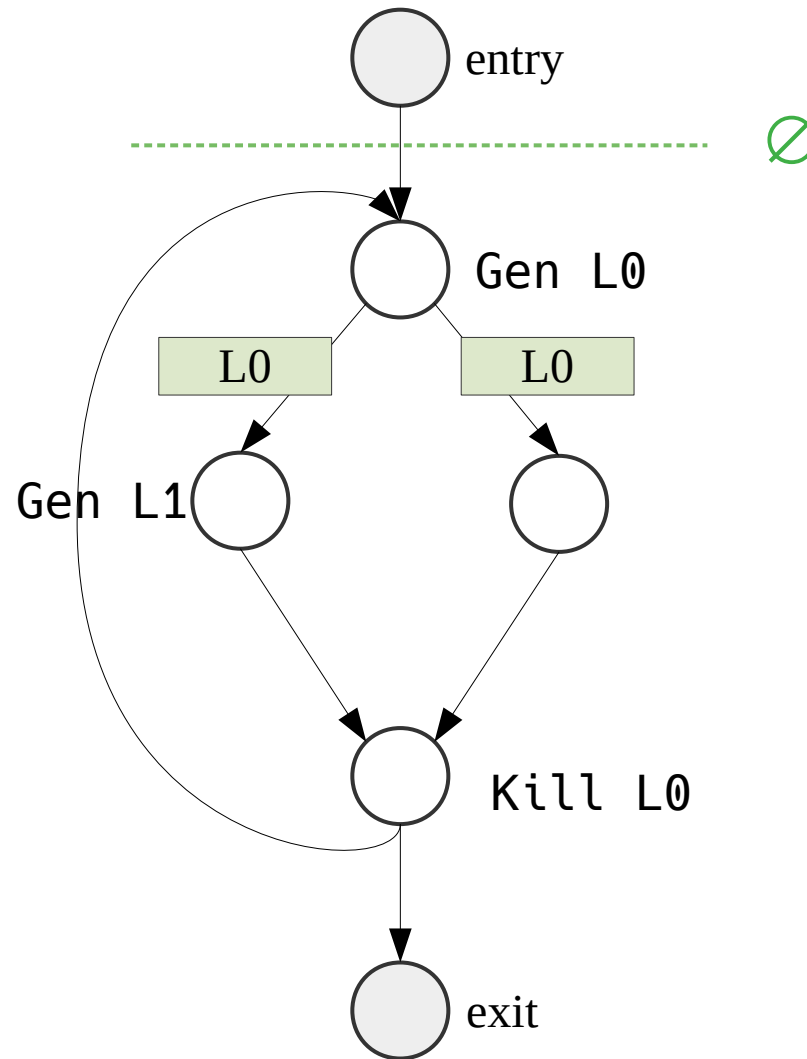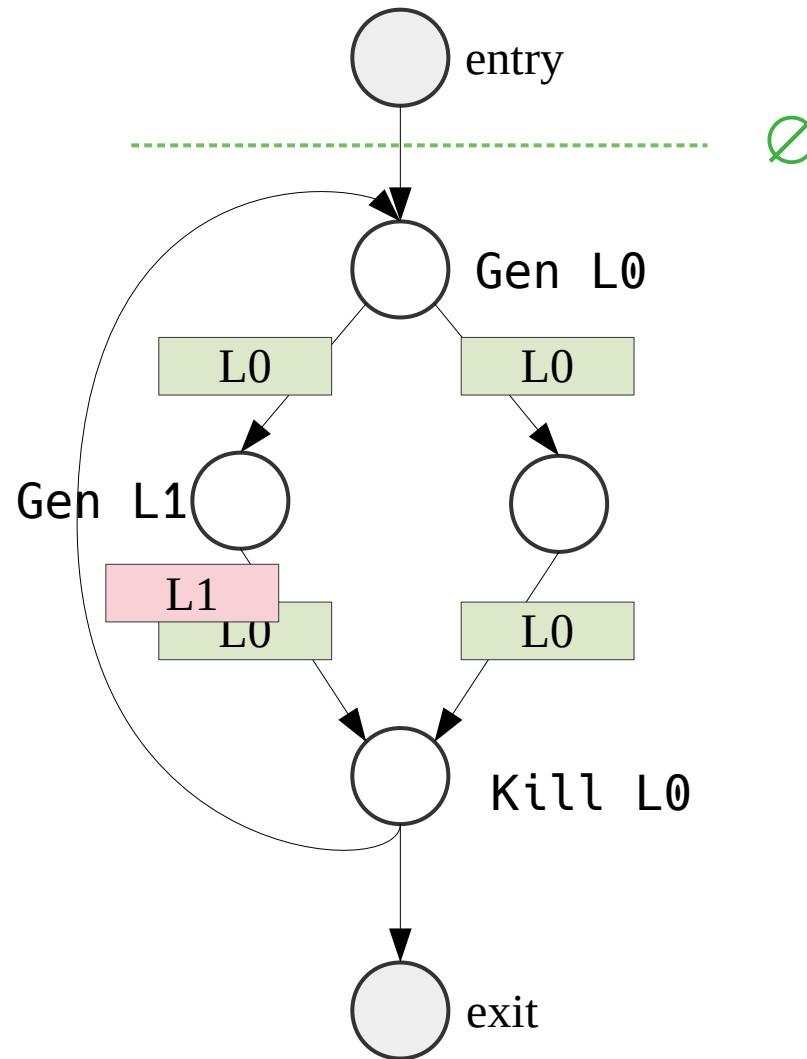$'1 : \{\ P_1\ ,\ …\ \}$

# The Data Flow of the Loan
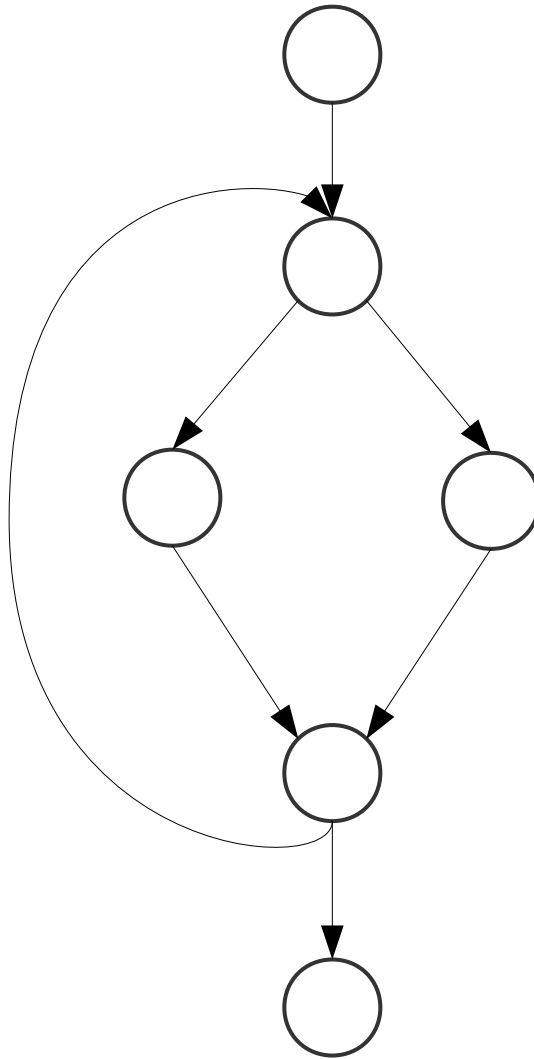
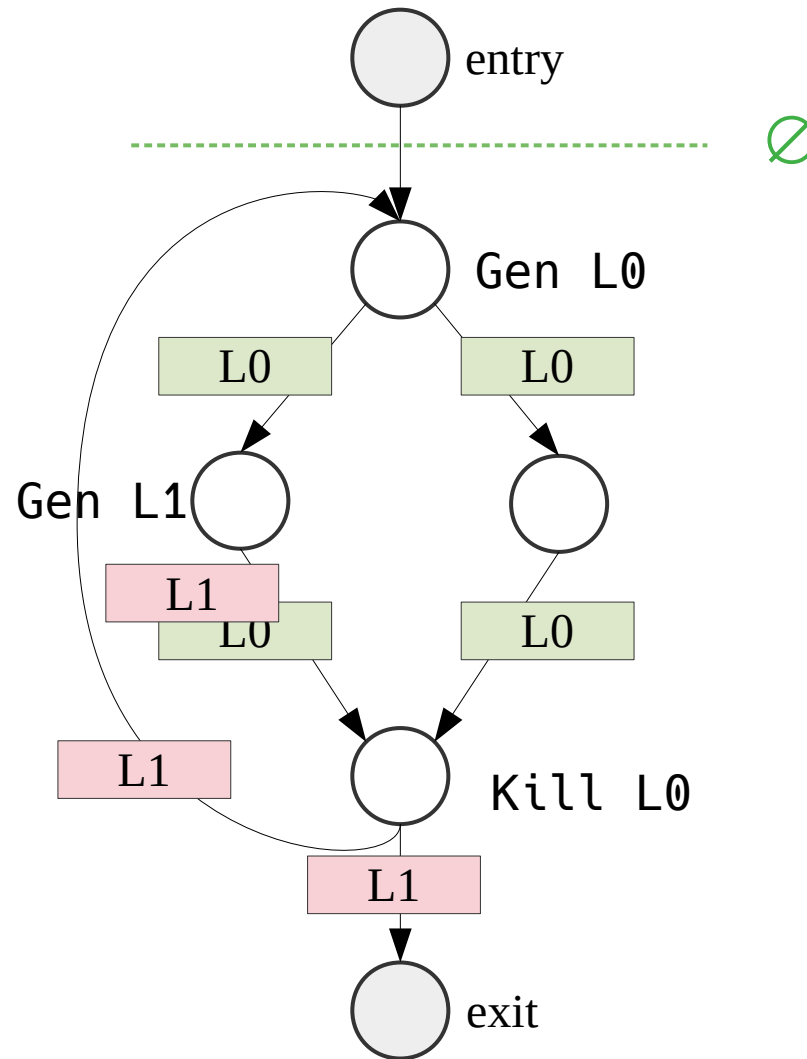# The Data Flow of the Loan

# 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
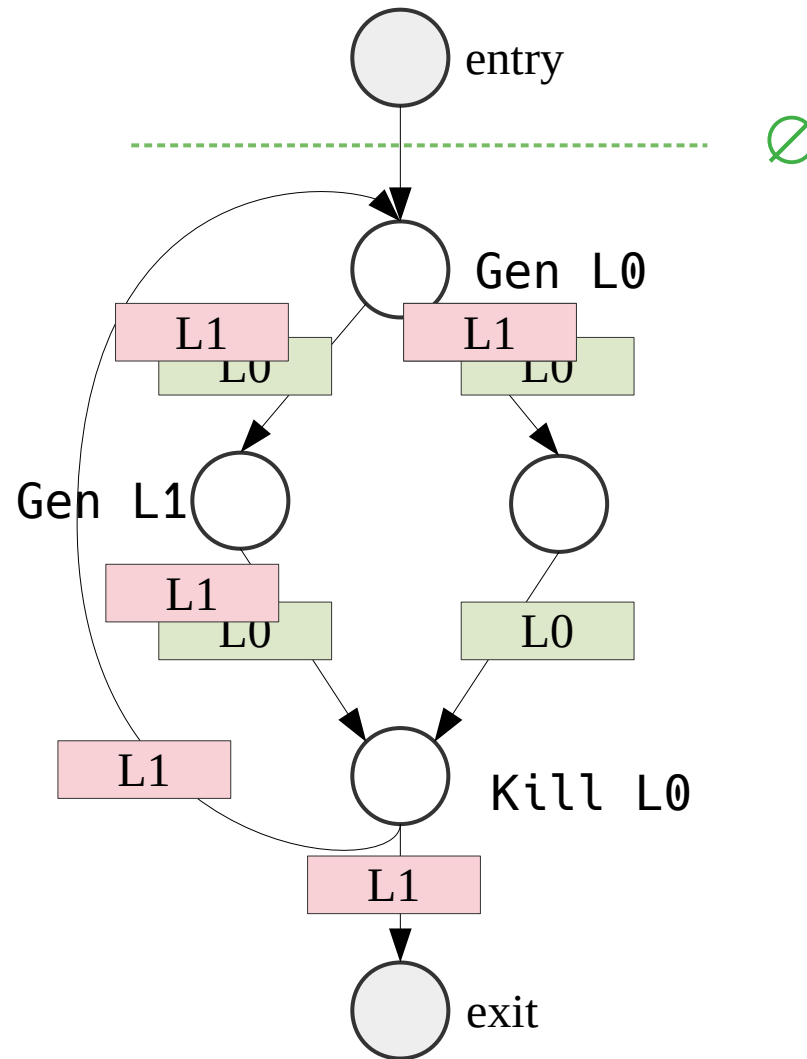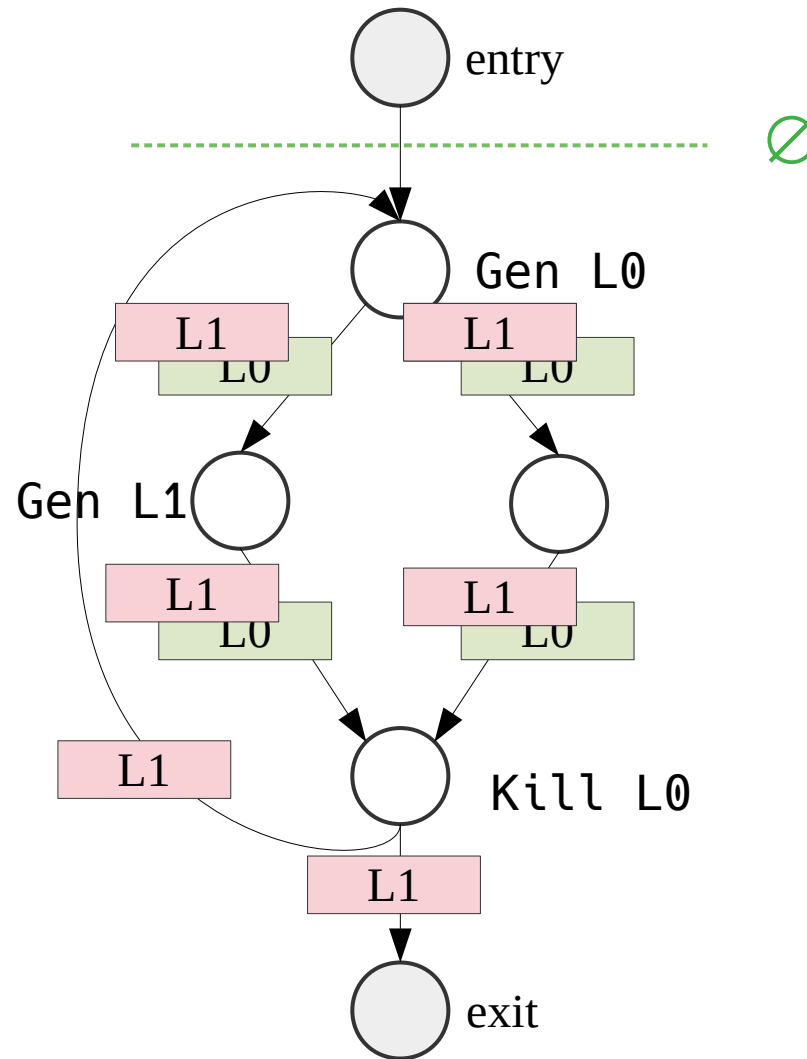
# The Data Flow of the Loan

**Key**: which loan live at which points



entry

$\varnothing$

Gen L0

| L1 |
| L0 |

| L1 |
| L0 |

Gen L1

| L1 |
| L0 |

| L1 |
| L0 |

| L1 |

Kill L0

| L1 |

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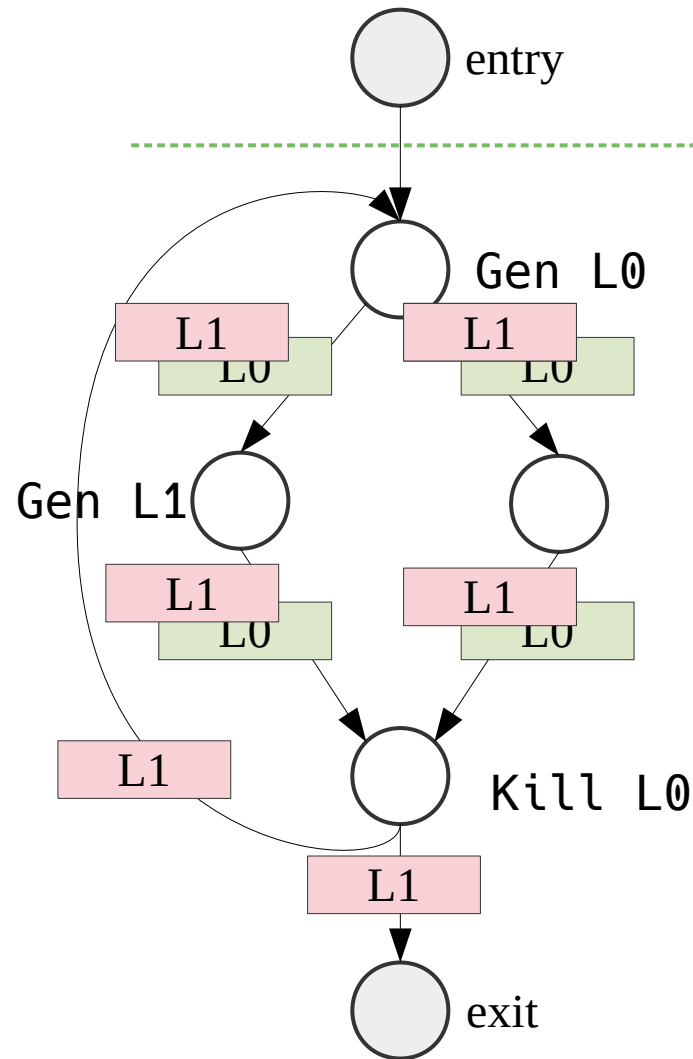
When all the sets are stable, that's mean **the state is not changed anymore**, then the data flow computation is complete.

# When to Gen, Kill

## Gen Loan :

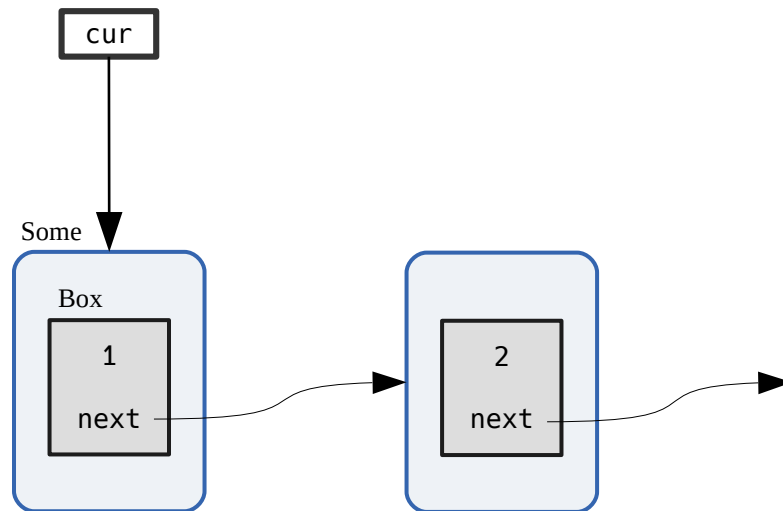If it's a borrow expression, then gen a Loan

## Kill Loan :

1) $LV = Loan_i \cdot path$
2) $point \notin Loan_i \cdot region$

# Example

```rust
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);
    }

}
```
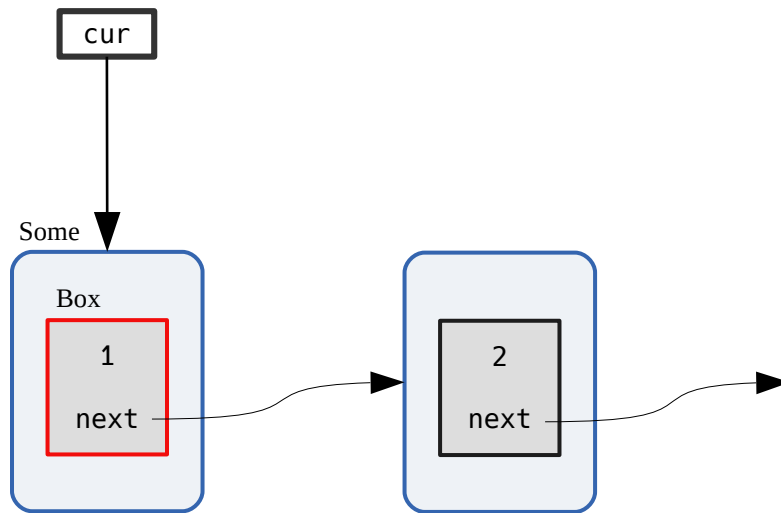
# Example

```
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

```rust
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);
    }

}
```
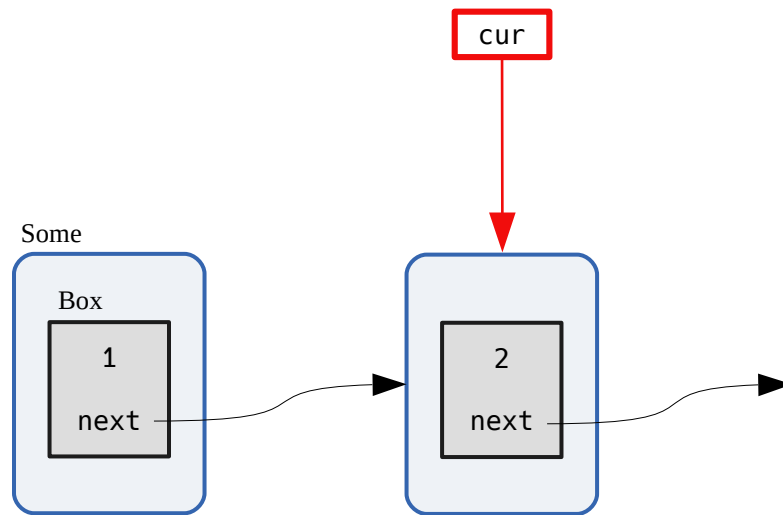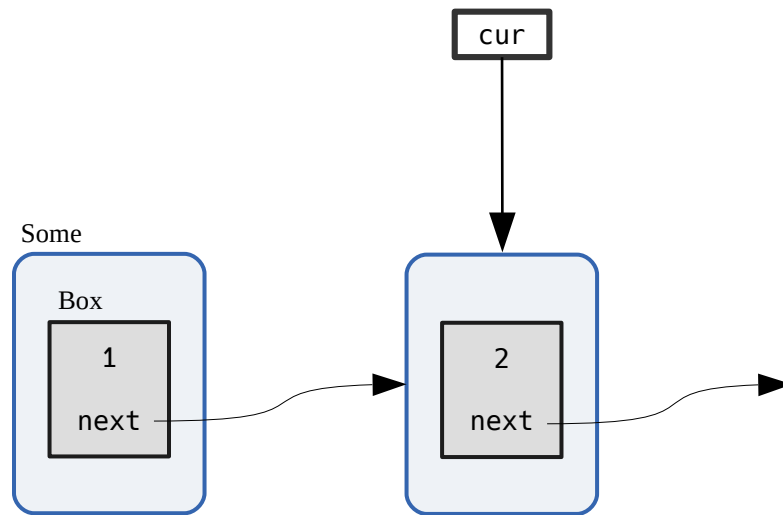
# Example

```
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

```rust
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);
    }

}
```
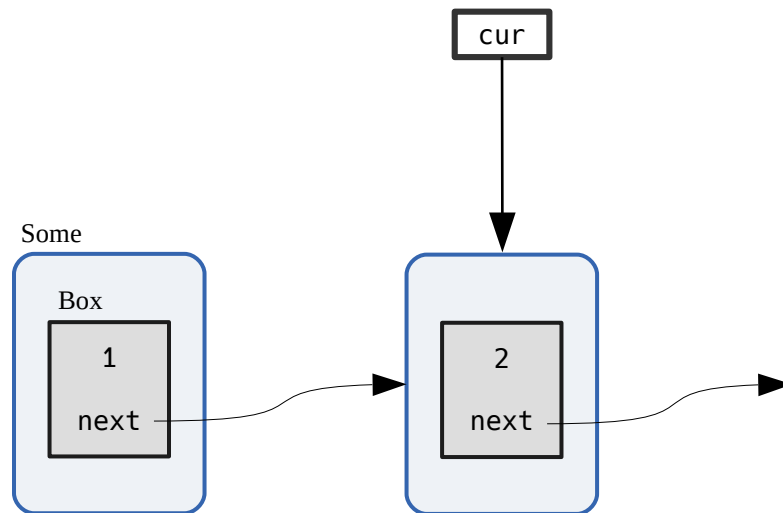
# Example

```rust
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);
    }

}
```
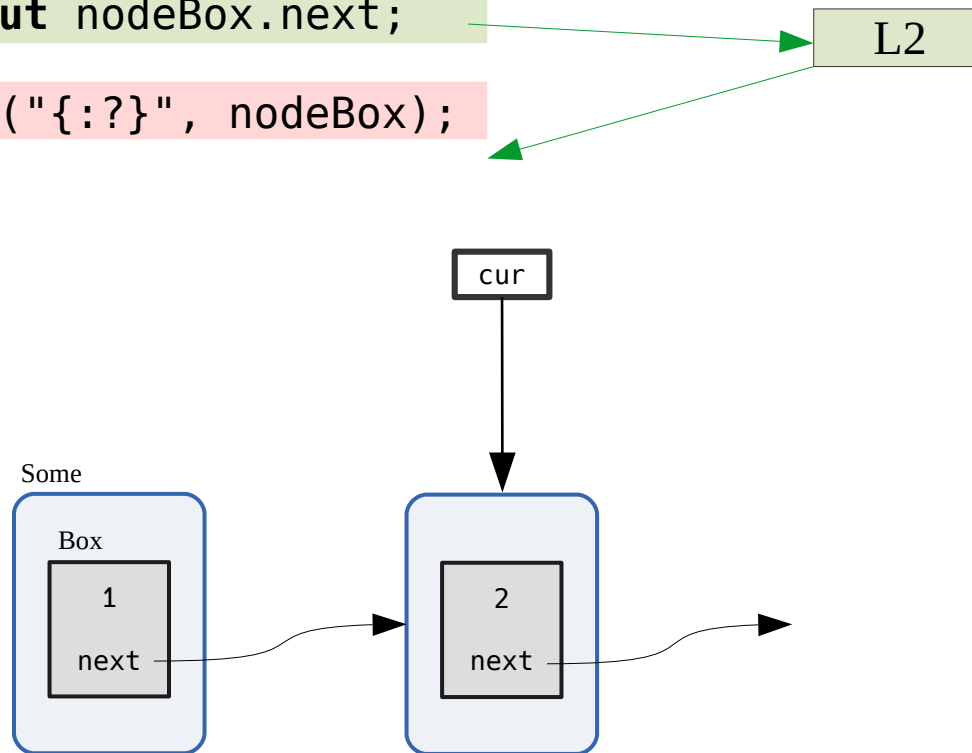
L2

cur

Some

Box

1

next

2

next
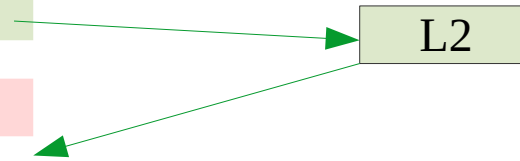
# Example

```
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);
    }

}
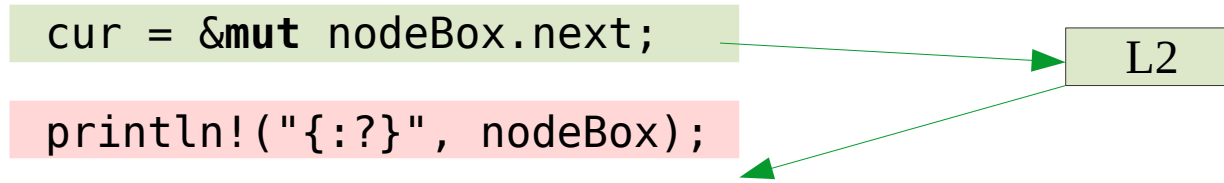```

# Example

```rust
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

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);
    }

}
```
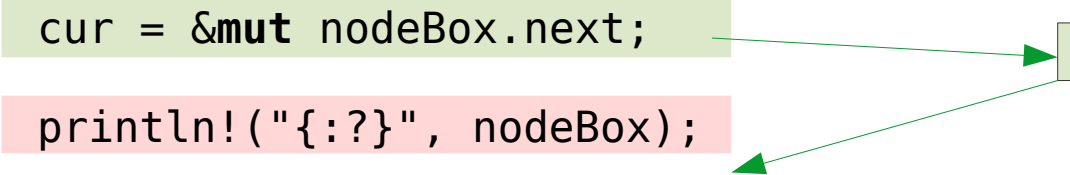
L2

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);
    }

}
```

L2

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.

# 參考題目

Leetcode : remove linked list elements

# Datafrog

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

# Idea

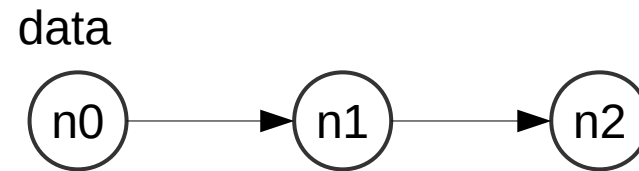▶ 每次都往前推論一步，直到每個節點都達到穩態即推論完畢

# 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)

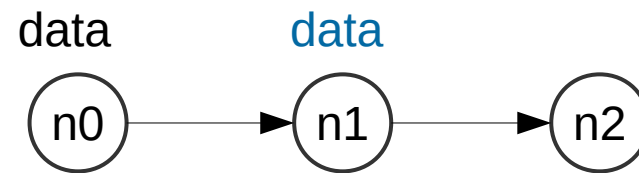# Idea

▶ 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)

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

data

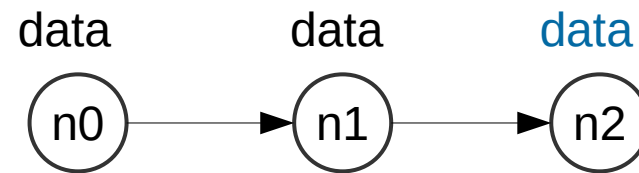# Idea

▶ 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)

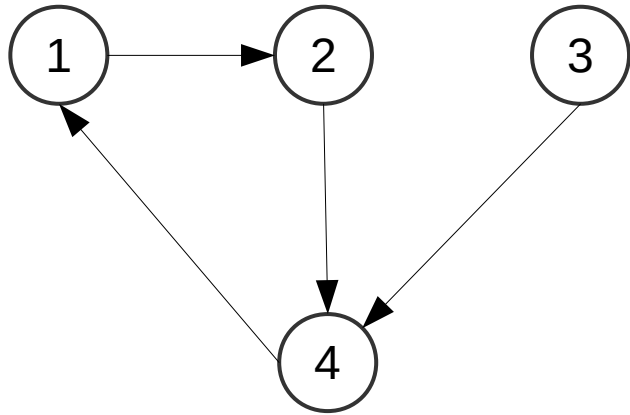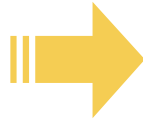$$\frac{N(a,x) \quad e(a,b)}{N(b,x)}$$

data      data

n0 → n1 → n2

# 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)

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

data     data     data

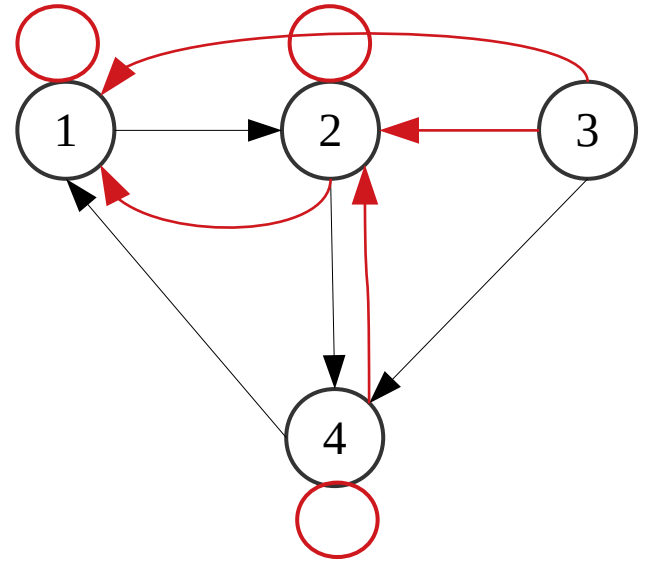(n0) → (n1) → (n2)

# Example . Transitive Closure

# Example . Transitive Closure
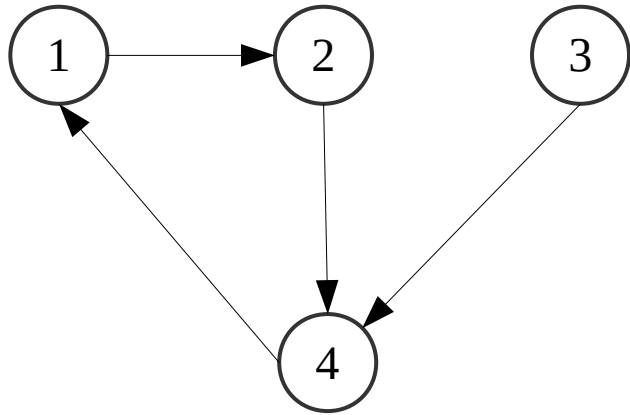


$$e(a,b)$$
$$\frac{e(b,c)}{e(a,c)}$$

# Implementation – Initial

```rust
// 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() {

    ...                        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));
}
```

$$\frac{\begin{array}{c} e(a,b) \\ e(b,c) \end{array}}{e(a,c)}$$
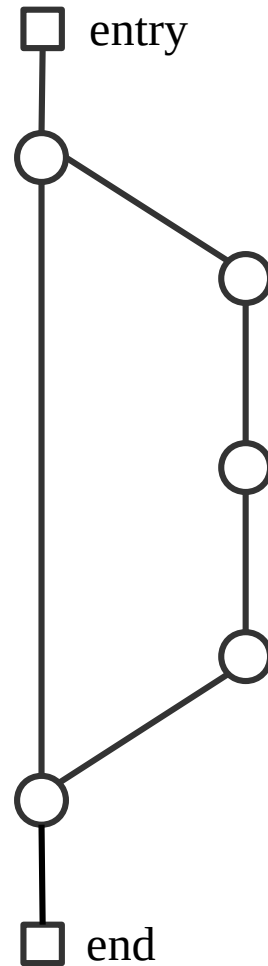
# Implementation – Writing Rules

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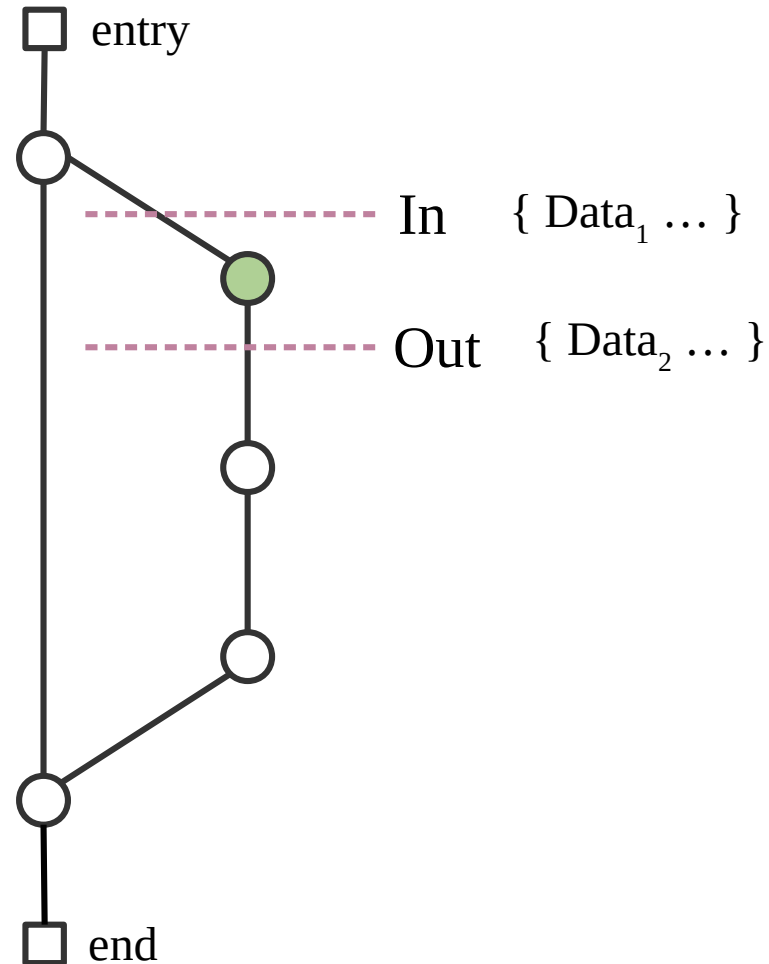
$$\frac{\begin{array}{c} e(a,b) \\ e(b,c) \end{array}}{e(a,c)} \quad \Longrightarrow \quad \frac{e(a,b)}{r(b,a)} \qquad \frac{\begin{array}{c} r(b,a) \\ e(b,c) \end{array}}{e(a,c)}$$

# 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));
}
```

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

# QA

# Data Flow Concepts $<D, V, \wedge, F>$

entry

end

# Data Flow Concepts <D, **V**, ∧, F>



entry

In     { Data$_1$ … }

Out     { Data$_2$ … }

end

# Data Flow Concepts $<\mathbf{D}, V, \wedge, F>$

# Data Flow Concepts <**D**, V, ∧, F>



entry

In    { Data$_1$ … }

Out    { Data$_2$ … }

end

# Data Flow Concepts $<D, V, \wedge, F>$



entry

$\{\ Data_2 \ \dots\ \}$

$\{\ Data_1 \ \dots\ \}$

?

end

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

# Data Flow Concepts $<D, V, \wedge, \mathbf{F}>$

IN     { $Data_1$ … }

Out     { $Data_2$ … }

# Data Flow Concepts $<D, V, \wedge, F>$

IN     { $Data_1$ … }

Gen & Kill

Out     { $Data_2$ … }

# Data Flow Concepts <D, V, ∧, F>

# Data Flow Concepts <D, **V**, ∧, F>

entry

In     { Data$_1$ … }

Out    { Data$_2$ … }

end

# Data Flow Concepts $<\mathbf{D}, V, \wedge, F>$



entry

In    { Data$_1$ … }

Out    { Data$_2$ … }

end

# Data Flow Concepts $<$**D**$, V, \wedge, F>$

# Data Flow Concepts $<D, V, \wedge, F>$



entry

$\{ Data_2 \dots \}$

$\{ Data_1 \dots \}$

?

end

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

# Data Flow Concepts $<D, V, \wedge, \textbf{F}>$

IN    { $Data_1$ … }

Out   { $Data_2$ … }

# Data Flow Concepts <D, V, ∧, **F**>

IN  { $Data_1$ … }

Out  { $Data_2$ … }

Gen & Kill