How NLL make life easier

Rnic / H.-S Zheng

April 17, 2019

About Me

Rnic / 鄭弘昇

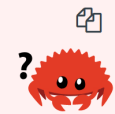
-  Telegram: { t.me/rnicinr }
-  **Emacs** 教派的信徒

Outline

- **Lifetimes Concept**
  - What is Lifetimes ?
  - Subtyping
- **NLL**
  - Phases of Borrow checker
  - Future: Polonius

Lifetimes Concept

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```



Borrow

&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

(by. https://rufflewind.com/img/rust-move-copy-borrow.png)

Borrow

&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

How long does it borrow ?

(by. https://rufflewind.com/img/rust-move-copy-borrow.png)

Borrow

&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

How long does it borrow ?

NLL: What is can borrowed here ?

(by. https://rufflewind.com/img/rust-move-copy-borrow.png)

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow → Reference

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow → Reference --- Lifetime

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow → Reference --- Lifetime

```
let r;
{
    let x = 5;
    r = &x;

}
println!("{}", r);
```

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow → Reference --- Lifetime

```
let r;
{
    let x = 5;
    r = &'b x;      // -+-- 'b
                    //  |
}                   // -+
println!("{}", r); //
```

Borrow

A borrow will generate a reference, and a reference will tagged with a lifetime

Borrow → Reference --- Lifetime

```
let r;
{
    let x = 5;
    r = &'b x;      // -+-- 'b
                    //  |
}                   // -+
println!("{}", r); //
```

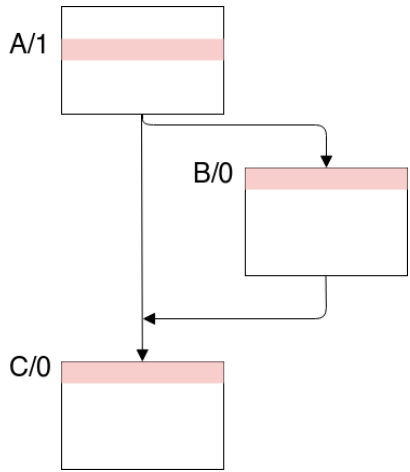
Lifetime must smaller than Scope

Lifetimes

Set of points on CFG

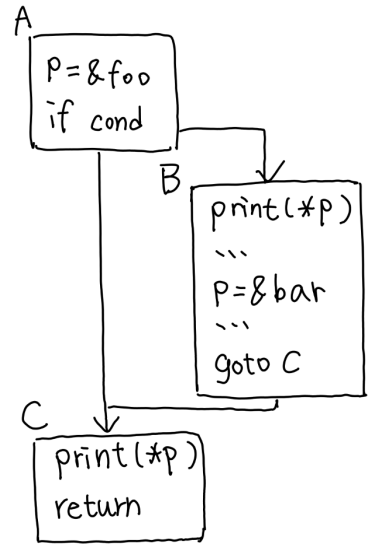
Lifetimes

Set of points on CFG



Lifetimes

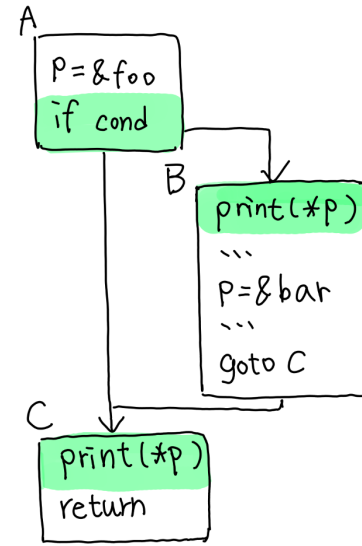
```
let mut foo: T = ...;
let mut bar: T = ...;
let p: &'p T;

p = &'foo foo;

if condition {

    print(*p);

    p = &'bar bar;
}

print(*p);
```



Lifetimes

```
let mut foo: T = ...;
let mut bar: T = ...;
let p: &'p T;

p = &'foo foo;

if condition {

    print(*p);

    p = &'bar bar;
}

print(*p);
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: { A/1, B/0, C/0 }

'bar: { B/3, B/4, C/0 }
```



Lifetimes

Original Limits

```
let mut s = "hello".to_string();
let mut c = || s += " world"; // captured by &mut and c become FnMut
c();
println!("{}", s);
```

Lifetimes

Original Limits

```
let mut s = "hello".to_string();

let mut c = || s += " world"; // captured by &mut and c become FnMut

c();

println!("{}", s);
```

```
|     let mut c = || s += " world";
|                 -- - previous borrow occurs due to use of `s` in closure
|                 |
|                 mutable borrow occurs here
|     c();
|     println!("{}", s);
|                    ^ immutable borrow occurs here
| }
| - mutable borrow ends here
```

Lifetimes

How to solve it ?

```
let mut s = "hello".to_string();

let mut c = || s += " world"; ---+--- 'a
                                 |
c();                             |
                                 |
println!("{}", s);          -----+--- 'b
                                 |
                             ----+
```

Lifetimes

How to solve it ?

```
let mut s = "hello".to_string();

let mut c = || s += " world"; ---+--- 'a
                                 |
c();                             |
                                 |
println!("{}", s);          -----+--- 'b
                                 |
                           ------+
```

```
let mut s = "hello".to_string();
 {
    let mut c = || s += " world"; ---+--- 'a
                                     |
    c();                             |
 }                                ----+

println!("{}", s);                ----+--- 'b
```

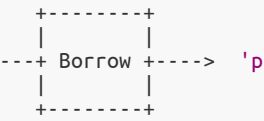
Subtyping

`'a : 'b` : lifetimes of `'a` is outlives `'b`

Subtyping

`'a : 'b` : lifetimes of `'a` is outlives `'b`

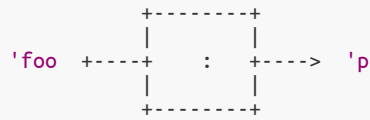
```
let p: &'p T;

p = &'foo foo;
```



Subtyping

`'a : 'b` : lifetimes of `'a` is outlives `'b`

```
let p: &'p T;

p = &'foo foo;

_____________

 'foo : 'p
```



Variance

	'a	T	U
&'a T	covariant	covariant	
&'a mut T	covariant	invariant	
Box		covariant	
Vec		covariant	
UnsafeCell		invariant	
Cell		invariant	
fn(T) -> U		contravariant	covariant
*const T		covariant	
*mut T		invariant	

```
+---+---+---+---+
| X | 0 | + | - |
+---------------+
| 0 | 0 | 0 | 0 |
+---------------+
| + | 0 | + | - |
+---------------+
| - | 0 | - | + |
+---+---+---+---+
```

0: invariant
+: convariant
-: contravariant

```
+---+---+---+---+
| ^ | 0 | + | - |
+---------------+
| 0 | 0 | 0 | 0 |
+---------------+
| + | 0 | + | 0 |
+---------------+
| - | 0 | 0 | - |
+---+---+---+---+
```

(by. https://medium.com/@kennytm/variance-in-rust-964134dd5b3e)

Example

```
let a = 5;

let b = &a;
```

Example

```
let a = 5;

let b: &'b i32 = &'a a;

_____________________

'a : 'b
```

Example

```
let a = 5;

let b: &'b i32 = &'a a;

_____________________

'a : 'b
```

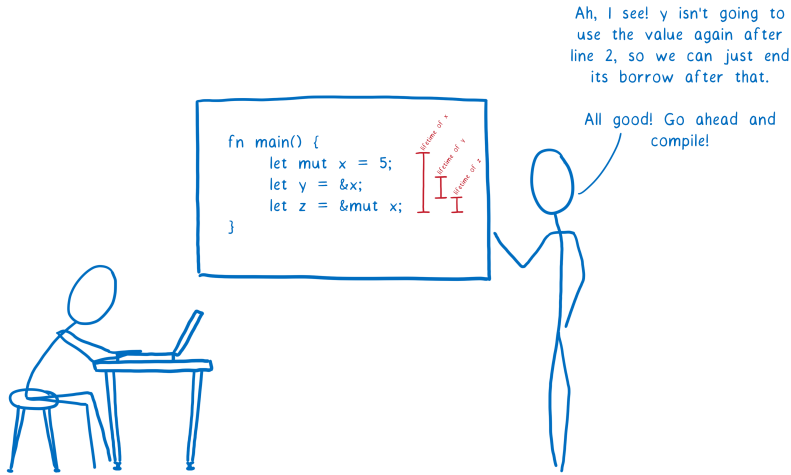
```
let x = 5;

let p = Cell::new(&x);
```

Example

```
let a = 5;

let b: &'b i32 = &'a a;

_____________________

'a : 'b
```

```
let x = 5;

let p: Cell<&'p i32> = Cell::new(&'x x);

_____________________________________

'x: 'p

'p: 'x
```

NLL


Ah, I see! y isn't going to use the value again after line 2, so we can just end its borrow after that.
All good! Go ahead and compile!
fn main() {
let mut x = 5;
let y = &x;
let z = &mut x;
}
lifetime of x
lifetime of y
lifetime of z

Problem

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
   match map.get_mut(&key) {
       Some(value) => {
           process(value);
           return;
       }
       None => {
           map.insert(key, V::default());
       }
   }
}
```

(play: link to play.rust-lang.org/ of this code with eidtion=2015)

Problem

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
   match map.get_mut(&key) {  --------------+-- 'a
       Some(value) => {                     |
           process(value);                  |
           return;                          |
       }                                    |
       None => {                            |
           map.insert(key, V::default()); ----- 'b
       }                                    |
   } <--------------------------------------+
}
```

```
|     match map.get_mut(&key) {
|           --- first mutable borrow occurs here
...
|             map.insert(key, V::default());
|             ^^^ second mutable borrow occurs here
|         }
|     }
|     - first borrow ends here
```

(play: link to play.rust-lang.org/ of this code with eidtion=2015)

Solution

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
    match map.get_mut(&key) {
        Some(value) => {
            process(value);
            return;
        }
        None => {

        }
    }
    map.insert(key, V::default());
}
```

(play: link to play.rust-lang.org/ of this code with eidtion=2015)

Solution

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
    match map.get_mut(&key) {
        Some(value) => {
            process(value);
            return;
        }
        None => {

        }
    }
    map.insert(key, V::default());
}
```

But it's annoying

(play: link to play.rust-lang.org/ of this code with eidtion=2015)

Phases of Borrowck

1. **Build liveness and constraints**
2. **Infer the lifetimes**
3. **Compute loans in scope**
4. **Check action and report errors**

Demo

(ref. Prototype of NLL https://github.com/nikomatsakis/borrowck)

Location-aware subtyping

`'a : 'b`:

Lifetimes of `'a` must outlives `'b`

Location-aware subtyping

`('a : 'b) @ P`:

`'a` must include all points in `'b`

that are reachable from location `P`

Location-aware subtyping

`('a : 'b) @ P`:

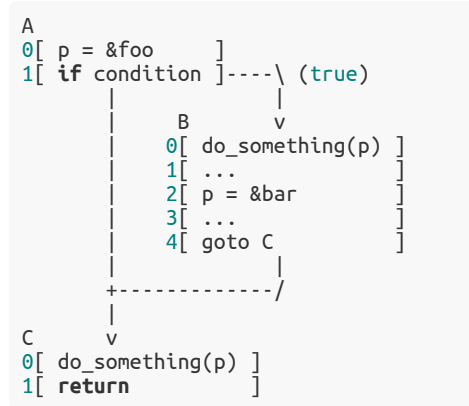
`'a` must include all points in `'b`

that are reachable from location `P`

Doing DFS on CFG

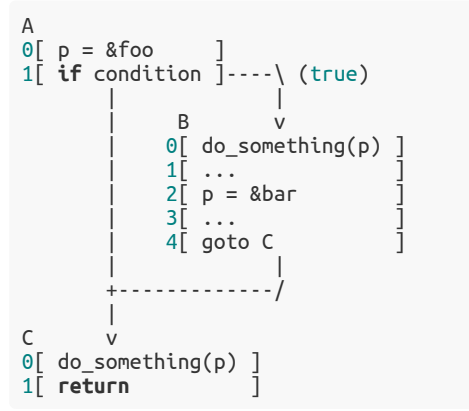
1. Build liveness and contraints

```
let foo = 2;
let bar = 5;
let mut p: &i32;

p = &foo;

if condition {

   do_something(p);

   p = &bar;
}

do_something(p);
```



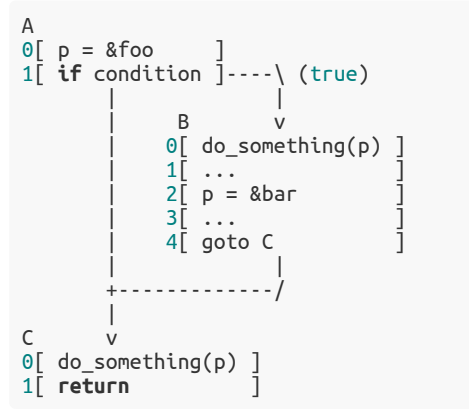
1. Build liveness and contraints

liveness: a variable v is live at point p if and only if there exists a path in CFG from p to a use of v along which v is not redefined.



1. Build liveness and contraints

liveness: a variable *v* is **live at point *p*** if and only if there exists a path in CFG **from *p* to a use of *v*** along which *v* is not redefined.



1. Build liveness and contraints

liveness: a variable *v* is **live at point *p*** if and only if there exists a path in CFG **from *p* to a use of *v*** along which *v* is not redefined.

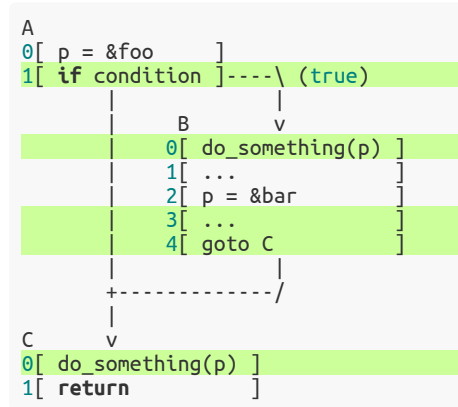
```
'p':   {

'foo': {

'bar': {
```

```
A
0[ p = &foo     ]
1[ if condition ]----\ (true)
       |             |
       |     B       v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &bar        ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

1. Build liveness and contraints

liveness: a variable *v* is **live at point *p*** if and only if there exists a path in CFG **from *p* to a use of *v*** along which *v* is not redefined.

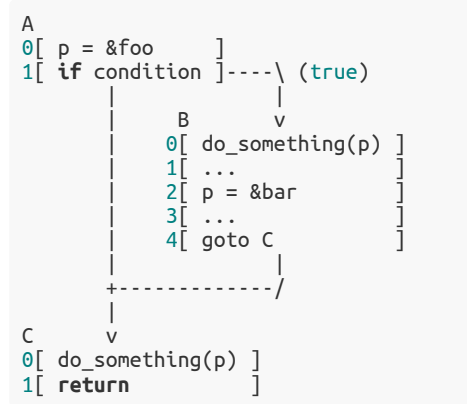
```
'p:   { A/1, B/0, B/3, B/4, C/0

'foo: {

'bar: {
```



1. Build liveness and contraints

contraints: It's lifetime contraints which is converted from subtyping rule.

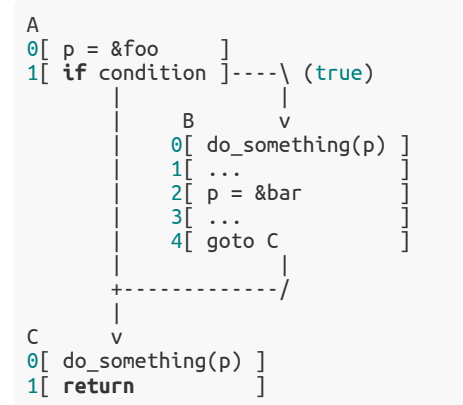
```
'p:   { A/1, B/0, B/3, B/4, C/0

'foo: {

'bar: {
```



1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

```
'p:   { A/1, B/0, B/3, B/4, C/0

'foo: {

'bar: {
```



1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

```
'p:   { A/1, B/0, B/3, B/4, C/0

'foo: {

'bar: {
```

```
p: &'p T;

A
0[ p = &'foo foo ]
1[ if condition  ]---\ (true)
       |            |
       |     B      v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &'bar bar   ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

- **('foo : 'p) @ A/0**

```
p: &'p T;

A
0[ p = &'foo foo ]
1[ if condition  ]---\ (true)
       |             |
       |     B       v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &'bar bar   ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

- **('foo : 'p) @ A/0**
- **('bar : 'p) @ B/2**

```
p: &'p T;

A
0[ p = &'foo foo ]
1[ if condition  ]---\ (true)
       |            |
       |     B      v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &'bar bar   ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

```
'p:   { A/1, B/0, B/3, B/4, C/0 }
'foo: {
'bar: {
`('foo : 'p) @ A/0`
`('bar : 'p) @ B/2`
```

1. Build liveness and contraints

contraints: It's lifetime contraints which is **converted from subtyping rule**, e.g. `(&'a T : &'b U)` will convert to `('a : 'b)`.

```
'p:   { A/1, B/0, B/3, B/4, C/0 }
'foo: {
'bar: {
`('foo : 'p) @ A/0`
`('bar : 'p) @ B/2`
```

Infer the lifetimes

Solving contraints (use DFS)

2. Infer lifetimes (solving contraints)

```
A
0[ p = &foo     ]
1[ if condition ]----\ (true)
       |             |
       |     B       v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &bar        ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: {

'bar: {

('foo : 'p) @ A/0

('bar : 'p) @ B/2
```

2. Infer lifetimes (solving contraints)

```
A
>> 0[ p = &foo  ]
1[ if condition ]----\ (true)
       |             |
       |     B       v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &bar        ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: {

'bar: {

>>('foo : 'p) @ A/0

('bar : 'p) @ B/2
```

2. Infer lifetimes (solving contraints)

```
A
0[ p = &foo     ]
1[ if condition ]----\ (true)
       |             |
       |     B       v
       |    0[ do_something(p) ]
       |    1[ ...             ]
       |    2[ p = &bar        ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: { A/1, B/0, C/0

'bar: {

('foo : 'p) @ A/0

('bar : 'p) @ B/2
```

2. Infer lifetimes (solving contraints)

```
A
0[ p = &foo     ]
1[ if condition ]----\ (true)
       |             |
       |      B      v
       |     0[ do_something(p) ]
       |     1[ ...             ]
>>     |     2[ p = &bar        ]
       |     3[ ...             ]
       |     4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: { A/1, B/0, C/0

'bar: {

('foo : 'p) @ A/0

>>('bar : 'p) @ B/2
```

2. Infer lifetimes (solving contraints)

```
A
0[ p = &foo     ]
1[ if condition ]----\ (true)
       |             |
       |      B      v
       |    0[ do_something(p) ]
       |    1[ ...             ]
>>     |    2[ p = &bar        ]
       |    3[ ...             ]
       |    4[ goto C          ]
       |             |
       +-------------/
       |
C      v
0[ do_something(p) ]
1[ return          ]
```

```
'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: { A/1, B/0, C/0

'bar: { B/3, B/4, C/0

('foo : 'p) @ A/0

('bar : 'p) @ B/2
```

2. Infer lifetimes (solving contraints)

```
'p:   { A/1, B/0, B/3, B/4, C/0 }
'foo: { A/1, B/0, C/0 }
'bar: { B/3, B/4, C/0 }
```

3. Compute loans in scope

loans: a set of borrow expressions

```
A/0 [ p = &'foo foo; ] // loan L0
```

```
loan L0 {
    point: A/0,
    path: foo,
    kind: shared,
    region 'foo {
        A/1, B/0, C/0
    }
}
```

3. Compute loans in scope

loans: a set of borrow expressions

Borrow checker will compute loans at each point via **fixed-point dataflow computation** with **transfer function**:

- `Kill Li @ P If P ∉ Li.region`
- `Gen  Li @ P If Borrow expression occur @ P`
- `Kill Li @ P If LV is redefined && LV ∈ Li.path`

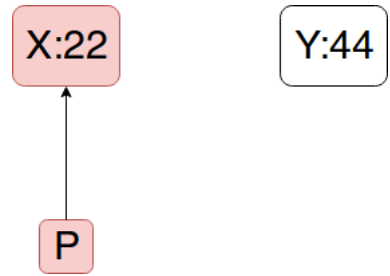
3. Compute loans in scope

```
          CFG                                        Loans @ each point
A
0[ p = &foo     ]                  "gen  L0" path: foo, region: { A/1, B/0, C/0 }
1[ if condition ]----\ (true)         L0
       |             |
       |     B       v
       |    0[ do_something(p) ]     L0
       |    1[ ...             ]  "kill L0" (rule 1)
       |    2[ p = &bar        ]  "gen  L1" path: bar, region: { B/3, B/4, C/0 }
       |    3[ ...             ]     L1
       |    4[ goto C          ]     L1
       |             |
       +-------------/
       |
       v
0[ do_something(p) ]                L0, L1
1[ return          ]               "kill L0, L1"


        Regions
_____________________________

'p:   { A/1, B/0, B/3, B/4, C/0 }

'foo: { A/1, B/0, C/0 }

'bar: { B/3, B/4, C/0 }
```

3. Compute loans in scope

Another example:

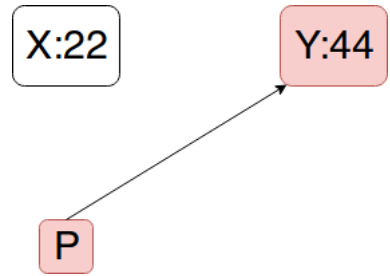
```
let mut x = 22;

let y = 44;

let mut p = &x;

p = &y;

x += 1;

use(*p);
```



3. Compute loans in scope

Another example:

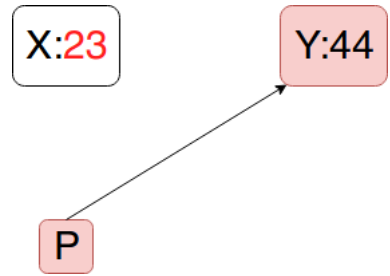
```
let mut x = 22;

let y = 44;

let mut p = &x;

p = &y;

x += 1;

use(*p);
```



3. Compute loans in scope

Another example:

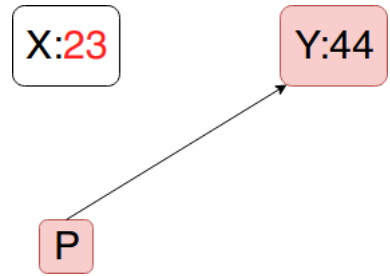
```
let mut x = 22;

let y = 44;

let mut p = &x;

p = &y;

x += 1;

use(*p);
```



3. Compute loans in scope

Another example:

```
let mut x = 22;
let y = 44;
let mut p = &x;   "L0"
p = &y;           "gen L1"
x += 1;           "kill L0"
use(*p);
```



4. Check action and report Error

Check action at each point which dependent on loans we computed before.

```
fn access_legal(lvalue, is_shallow, is_read) {
    let relevant_borrows = select_relevant_borrows(lvalue, is_shallow);

    for borrow in relevant_borrows {
        // shared borrows like `&x` still permit reads from `x` (but not writes)
        if is_read && borrow.is_read { continue; }

        // otherwise, report an error, because we have an access
        // that conflicts with an in-scope borrow
        report_error();
    }
}
```

	read	write
shallow		storageDead() lvalue
deep	rvalue(copy) &lvalue	rvalue(move) &mut lvalue Drop(lvalue)

lvalue = rvalue;

```
x = x + 1

// shallow access to lvalue x
```

We Make it !

We Make it !

```
A/0 | let mut x = 42;
A/1 | let y = &x;
A/2 | x += 1;
```

We Make it !

```
A/0 | let mut x = 42;
A/1 | let y = &x;
A/2 | x += 1;
```

```
x: { A/0, A/1, A/2 }

y: { A/1 }
```

We Make it !

```
A/0 | let mut x = 42;
A/1 | let y = &x;
A/2 | x += 1;
```

```
x: { A/0, A/1, A/2 }

y: { A/1 }
```

⇒ y is not live at A/2

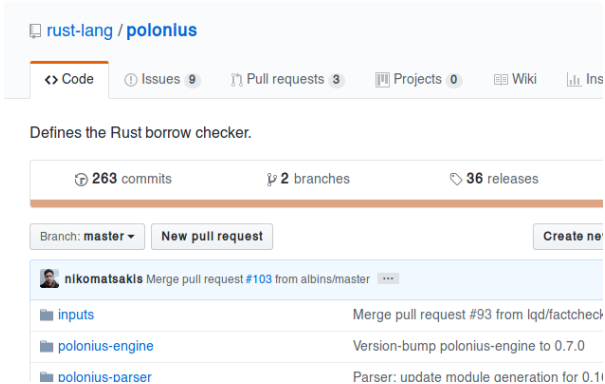
We Make it !

```
A/0 | let mut x = 42;
A/1 | let y = &x;
A/2 | x += 1;
```

```
x: { A/0, A/1, A/2 }

y: { A/1 }
```

⇒ y is not live at A/2 ⇒ Legal for you to write x

Future: Polonius



Polonius

- He is chief counsellor of the king
- A busy-body, [who] is accordingly officious, garrulous, and impertinent
- Polonius hides himself behind an arras in Gertrude's room. Hamlet deals roughly with his mother, causing her to cry for help. Polonius echoes the request for help and is heard by Hamlet, who then mistakes the voice for Claudius' and stabs through the arras and kills him.

(ref. https://en.wikipedia.org/wiki/Polonius)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

(ref. an aliased-based formulation of the borrow checker)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
              ^^^^ "mutable borrow starts here in previous iteration of loop"

            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

(ref. an aliased-based formulation of the borrow checker)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
              ^^^^ "mutable borrow starts here in previous iteration of loop"

            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

Reason: In `None` arm, `temp` was not reassigned.

(ref. an aliased-based formulation of the borrow checker)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
              ^^^^ "mutable borrow starts here in previous iteration of loop"

            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

Reason: In `None` arm, `temp` was not reassigned.

Why accepted in the future?

(ref. an aliased-based formulation of the borrow checker)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
              ^^^^ "mutable borrow starts here in previous iteration of loop"

            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

Reason: In `None` arm, `temp` was not reassigned.

Why accepted in the future?

1. `Some` path: Killed the `loan`
2. `None` path: `requires relation` is dropped.

(ref. an aliased-based formulation of the borrow checker)

Problem

We want to traversal the Thing and loop forever

```
struct Thing;

impl Thing { fn maybe_next(&mut self) -> Option<&mut Self> { None } }

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
              ^^^^ "mutable borrow starts here in previous iteration of loop"

            Some(v) => { temp = v; }
            None => { }
        }
    }
}
```

```
R1 live @ P
R1 require L @ P
________________

L live @ p
```

(ref. an aliased-based formulation of the borrow checker)

What is change ?

What is change ?

Region 'a: Set of points in CFG

```
{ A/0, A/1, ... }
```

What is change ?

Region 'a: ~~Set of points in CFG~~

Set of Loans

```
{ L0, L1 ... }
```

What is change ?

Region 'a: ~~Set of points in CFG~~

Set of Loans

```
{ L0, L1 ... }
```

And with some rules, but I'm not mentioned in this talk, e.g. `require_relation`, `borrow_region`...etc

I have not mentioned in this talk

- **Reborrow contraints :** supporting prefixes rule
- **Infinite Loops :** add unwind edge
- **Drop variable :** [may_dangle] → not consider live
- **Named Lifetimes** : add end_regions

I have not mentioned in this talk

- **Reborrow contraints :** supporting prefixes rule
- **Infinite Loops :** add unwind edge
- **Drop variable :** [may_dangle] → not consider live

```
&'a T   // 'a is [may_dangle]

Foo<'a> and you impl Drop // 'a cannot dangle
```

- **Named Lifetimes** : add end_regions

Command

```
"Gen MIR information:"

    rustc --dump-mir=main src/main.rs

"or"
    rustc -Z unpretty=mir src/main.rs


"Test NLL Borrowck prototype:"

    env NLL_DEBUG=true cargo run ../test/foobar.nll
```

reference

(subtype: https://doc.rust-lang.org/nomicon/subtyping.html)

(variance: https://medium.com/@kennytm/variance-in-rust-964134dd5b3e)

(nll: https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md)

(alias-based-formulation: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/)

That's All.

QA

 **Rust Taiwan @ https://t.me/rust_tw**