Inside the Rust Borrow Checker

Jakub Dupák

#lang-talk meetup

19. 2. 2024

Borrow Checker Rules

- Move
- Lifetime subset relation
- Borrow must outlive borrowee
- One mutable borrow or multiple immutable borrows
- No modification of immutable borrow data

Borrow Checker Rules

- Move

```
let mut v1 = Vec::new();
v1.push(42)
let mut v2 = v1; // <- Move
println!(v1[0]); // <- Error
```

- Lifetime subset relation
- Borrow must outlive borrowee
- One mutable borrow or multiple immutable borrows
- No modification of immutable borrow data

Borrow Checker Rules

- Move
- Lifetime subset relation
- Borrow must outlive borrowee

```
fn f() -> &i32 {
  &(1+1)
} // <- Error
```

- One mutable borrow or multiple immutable borrows
- No modification of immutable borrow data

Borrow Checker Rules

- Move
- Lifetime subset relation
- Borrow must outlive borrowee
- One mutable borrow or multiple immutable borrows
- No modification of immutable borrow data

```
let mut counter = 0;
let ref1 = &mut counter;
// ...
let ref2 = &mut counter; //  <- Error
```

Checking Functions

```
struct Vec<'a> { ... }

impl<'a> Vec<'a> {
  fn push<'b> where 'b: 'a (&mut self, x: &'b i32) {
    // ...
  }
}
```

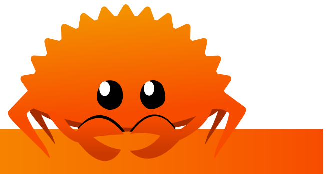
Checking Functions

```
struct Vec<'a> { ... }

impl<'a> Vec<'a> {
  fn push<'b> where 'b: 'a (&mut self, x: &'b i32) {
    // ...
  }
}
```

```
let a = 5;                       //  'a   'b   'b: 'a
{                                //
   let mut v = Vec::new();      //   *
   v.push(&a);                   //   *    *     OK
   let x = v[0];                 //   *    *     OK
 }                               //        *     OK
```

Borrow checker evolution

Lexical, NLL, Polonius



Lexical borrow checker

```
fn foo() {
  let mut data = vec!['a', 'b', 'c'];
  capitalize(&mut data[..]);
  data.push('d');
  data.push('e');
  data.push('f');
}
```

Lexical borrow checker

```
fn foo() {
  let mut data = vec!['a', 'b', 'c']; // --+ 'scope
  capitalize(&mut data[..]);          //   |
  // ^~~~~~~~~~~~~~~~~~~~~ 'lifetime //   |
  data.push('d');                     //   |
  data.push('e');                     //   |
  data.push('f');                     //   |
} // <------------------------------------+
```

Lexical borrow checker

```
fn bar() {
  let mut data = vec!['a', 'b', 'c'];
  let slice = &mut data[..]; // <-+ 'lifetime
  capitalize(slice);         //   |
  data.push('d'); // ERROR!  //   |
  data.push('e'); // ERROR!  //   |
  data.push('f'); // ERROR!  //   |
} // <----------------------------+
```

Lexical borrow checker

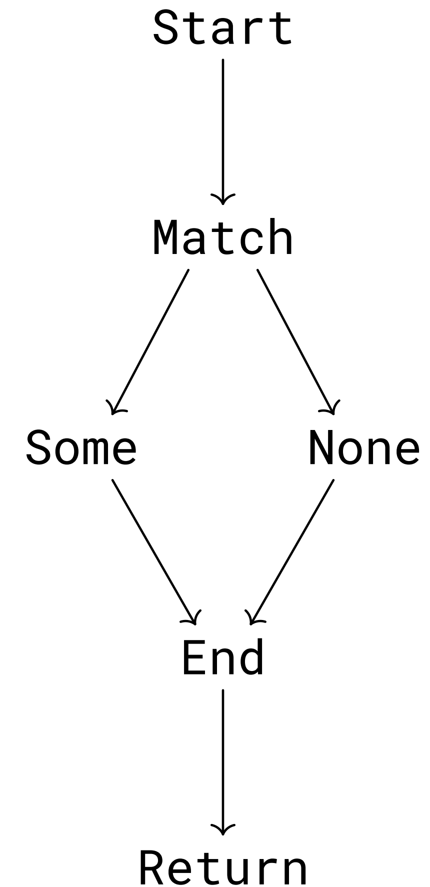
```
fn process_or_default() {
  let mut map = ...;
  let key = ...;
  match map.get_mut(&key) { // -------------+ 'lifetime
      Some(value) => process(value),     // |
      None => {                          // |
          map.insert(key, V::default()); // |
          //  ^~~~~~ ERROR.              // |
      }                                  // |
  }; // <------------------------------------+
}
```

Non-lexical lifetimes (NLL)

lifetime = set of CFG nodes

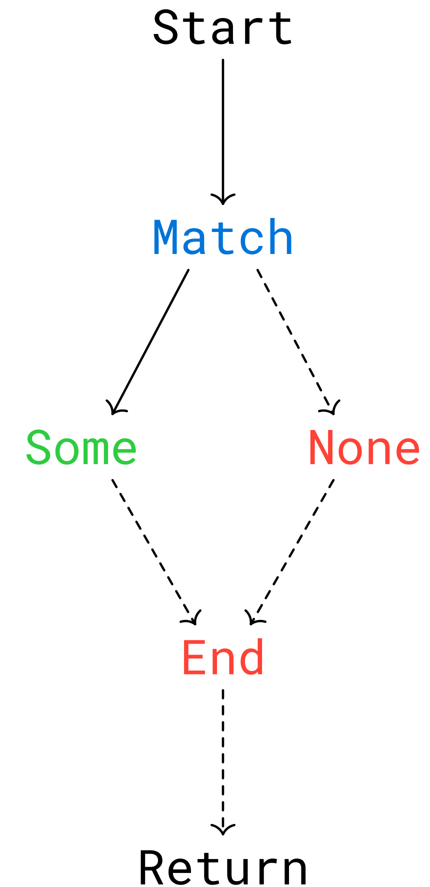
Non-lexical lifetimes (NLL)

```
fn f<'a>(map: &'r mut HashMap<K, V>) {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default());
    }
  }
}
```



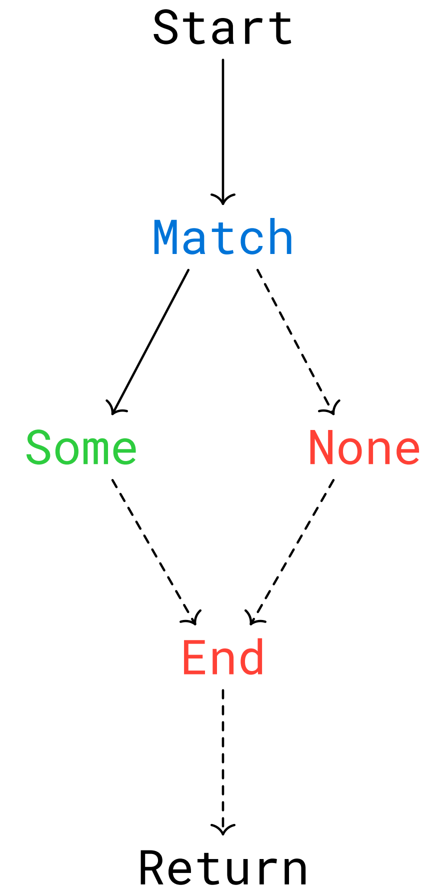
Non-lexical lifetimes (NLL)

```
fn f<'a>(map: &'r mut HashMap<K, V>) {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default());
    }
  }
}
```



Non-lexical lifetimes (NLL)

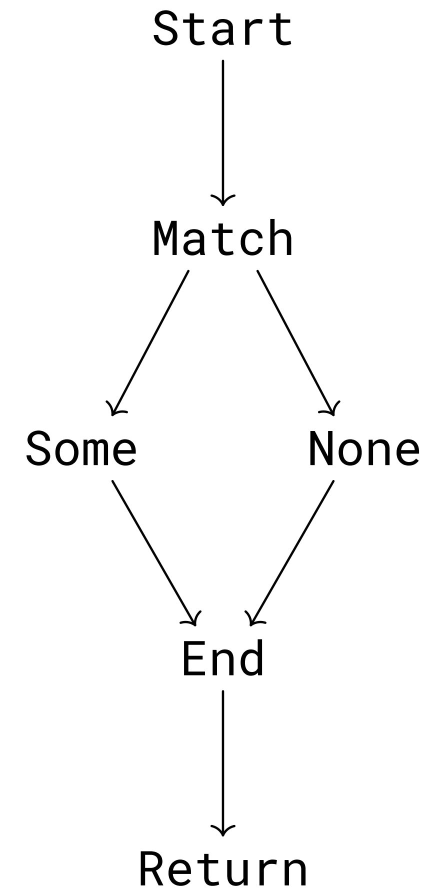
```
fn f<'a>(map: &'r mut HashMap<K, V>) {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default());
    }
  }
}
```



NLL → lifetimes are CFG nodes

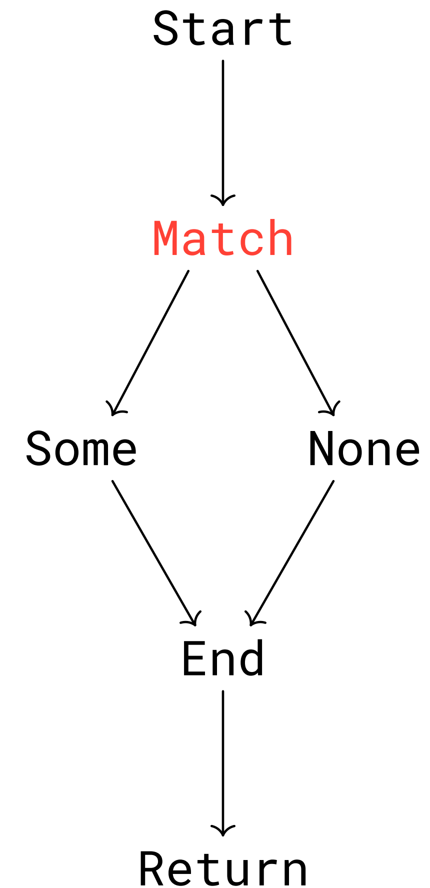
Breaking NLL

```
fn f<'a>(map: &'a mut Map<K, V>) -> &'a V {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default())
    }
  }
}
```



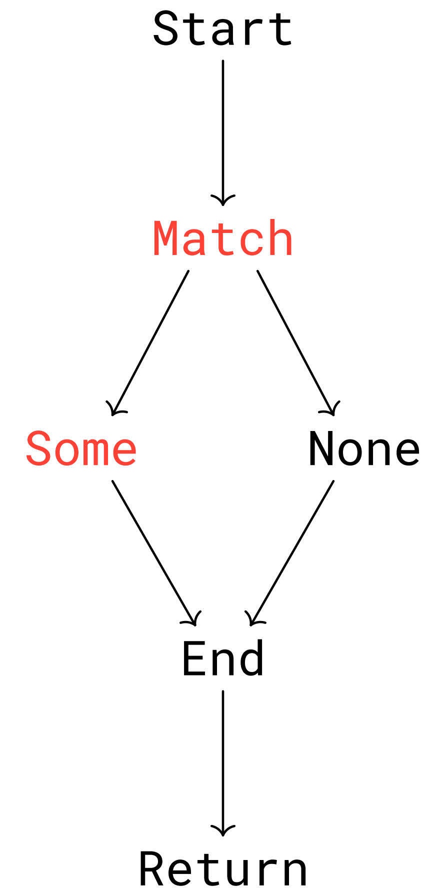
Breaking NLL

```
fn f<'a>(map: &'a mut Map<K, V>) -> &'a
V {
    ...
    match map.get_mut(&key) {
      Some(value) => process(value),
      None => {
        map.insert(key, V::default())
      }
    }
}
```



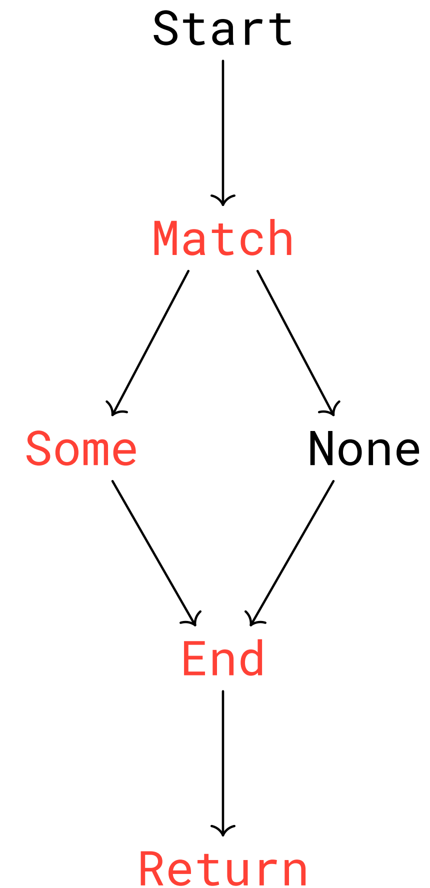
Breaking NLL

```
fn f<'a>(map: &'a mut Map<K, V>) -> &'a
V {
    ...
    match map.get_mut(&key) {
        Some(value) => process(value),
        None => {
            map.insert(key, V::default())
        }
    }
}
```



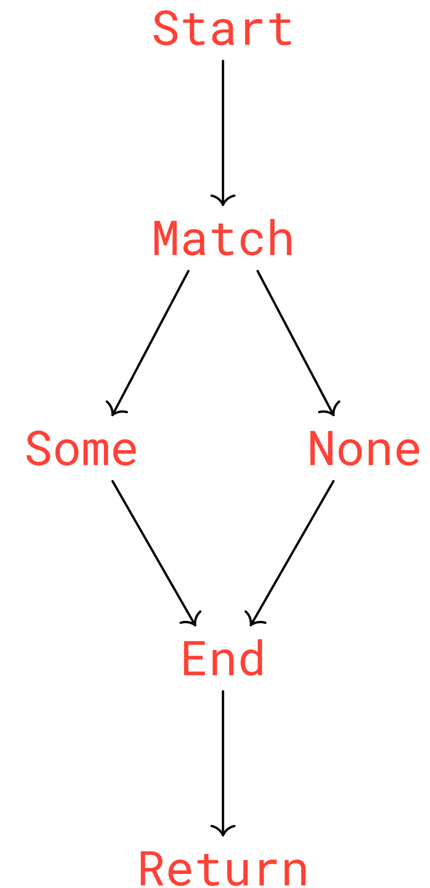
Breaking NLL

```
fn f<'a>(map: &'a mut Map<K, V>) -> &'a V {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default())
    }
  }
}
```



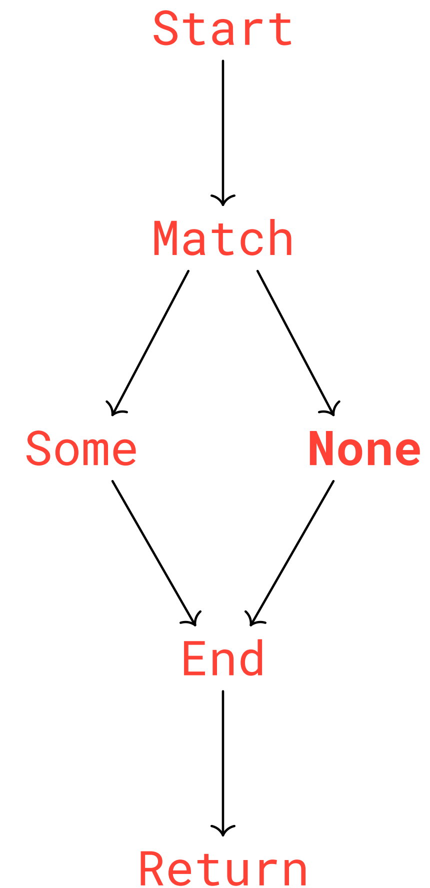
Breaking NLL

```
fn f<'a>(map: &'a mut Map<K, V>) -> &'a
V {
    ...
    match map.get_mut(&key) {
      Some(value) => process(value),
      None => {
        map.insert(key, V::default())
      }
    }
}
```



Breaking NLL

```
  fn f<'a>(map: &'a mut Map<K, V>) -> &'a
V {
    ...
    match map.get_mut(&key) {
      Some(value) => process(value),
      None => {
        map.insert(key, V::default())
      }
    }
  }
```



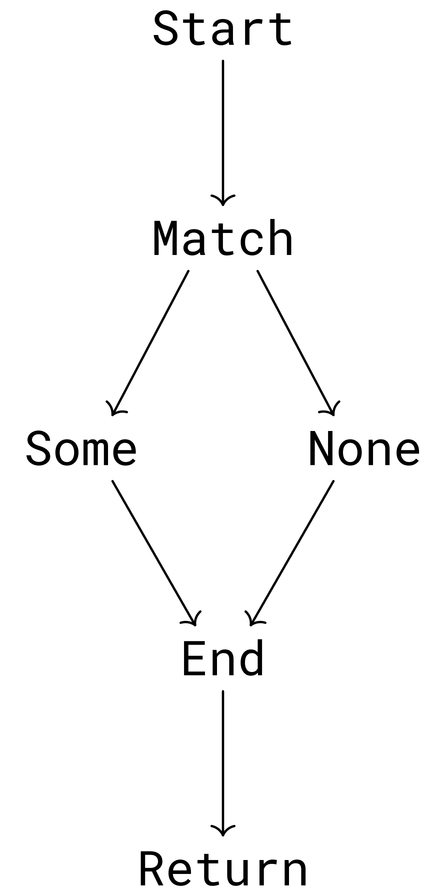
Error!

Polonius

Lifetime = set of loans

Polonius

```
fn f<'a>(map: Map<K, V>) -> &'a V {
  ...
  match map.get_mut(&key) {
    Some(value) => process(value),
    None => {
      map.insert(key, V::default());
    }
  }
}
```



Polonius

```
let r: &'0 i32 = if (cond) {
  &x /* Loan L0 */
} else {
  &y /* Loan L1 */
};
```

How does the program look?

Internal representations

Internal representations

- AST = abstract syntax tree
- HIR = high-level IR
- Ty = type IR
- THIR = typed HIR
- **MIR** = mid-level IR

```
struct Foo(i31);

fn foo(x: i31) -> Foo {
    Foo(x)
}
```

HIR

```
Fn {
generics: Generics { ... },
sig: FnSig {
  header: FnHeader { ... },
  decl: FnDecl {
    inputs: [
      Param {
        ty: Ty {
          Path { segments: [ PathSegment {
                  ident: i32#0 } ] }
        }
        pat: Pat { Ident(x#0) }
      },
    ],
    output: Ty { Path { segments: [ PathSegment {
        ident: Foo#0 } ] }
```

MIR

```
fn foo(_1: i32) -> Foo {
    debug x => _1;
    let mut _0: Foo;

    bb0: {
        _0 = Foo(_1);
        return;
    }
}
```

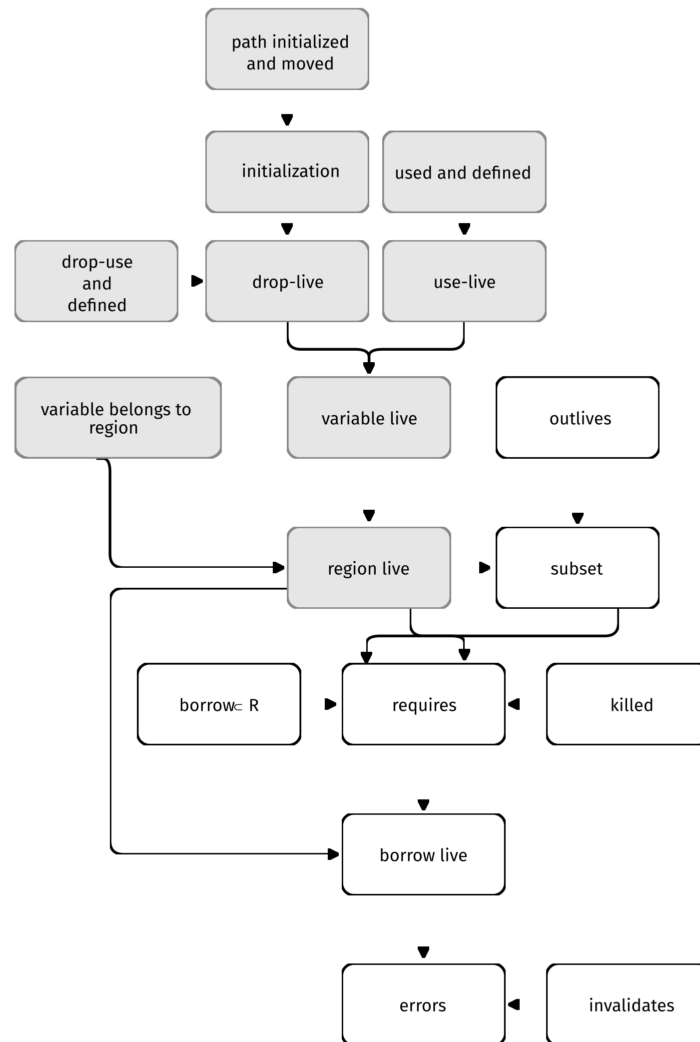
MIR: Fibonacci

```
fn fib(_2: u32) -> u32 {
  bb0: {
  0    StorageLive(_3);
  1    StorageLive(_5);
  2    _5 = _2;
  3    StorageLive(_6);
  4    _6 = Operator(move _5, const u32);
  5    switchInt(move _6) -> [bb1, bb2];
  }

  bb1: {
  0    _3 = const bool;
  1    goto -> bb3;
  }

  bb2: {
  0    StorageLive(_8);
  1    _8 = _2;
  2    StorageLive(_9);
  3    _9 = Operator(move _8, const u32);
  4    _3 = move _9;
  5    goto -> bb3;
  }

  bb3: {
  0    switchInt(move _3) -> [bb4, bb5];
  }

  bb4: {
  0    _1 = const u32;
  1    goto -> bb8;
  }

  bb5: {
  0    StorageLive(_14);
  1    _14 = _2;
  2    StorageLive(_15);
  3    _15 = Operator(move _14, const u32);
  4    StorageLive(_16);
  5    _16 = Call(fib)(move _15) -> [bb6];
  }

  bb6: {
  1    _19 = _2;
  3    _20 = Operator(move _19, const u32);
  5    _21 = Call(fib)(move _20) -> [bb7];
  }

  bb7: {
  0    _1 = Operator(move _16, move _21);
  7    goto -> bb8;
  }

  bb8: {
  5    return;
  }
}
```

Computing!

Steps of the borrow checker

What do we need?



What do we need?

path initialized and moved

initialization

used and defined

drop-use and defined

drop-live

use-live

variable belongs to region

variable live

outlives

What do we need?

and defined

drop-live

use-live

variable belongs to region

variable live

outlives

region live

subset

borrow⊂ R

requires

killed

What do we need?

region live

subset

borrow⊂ R

requires

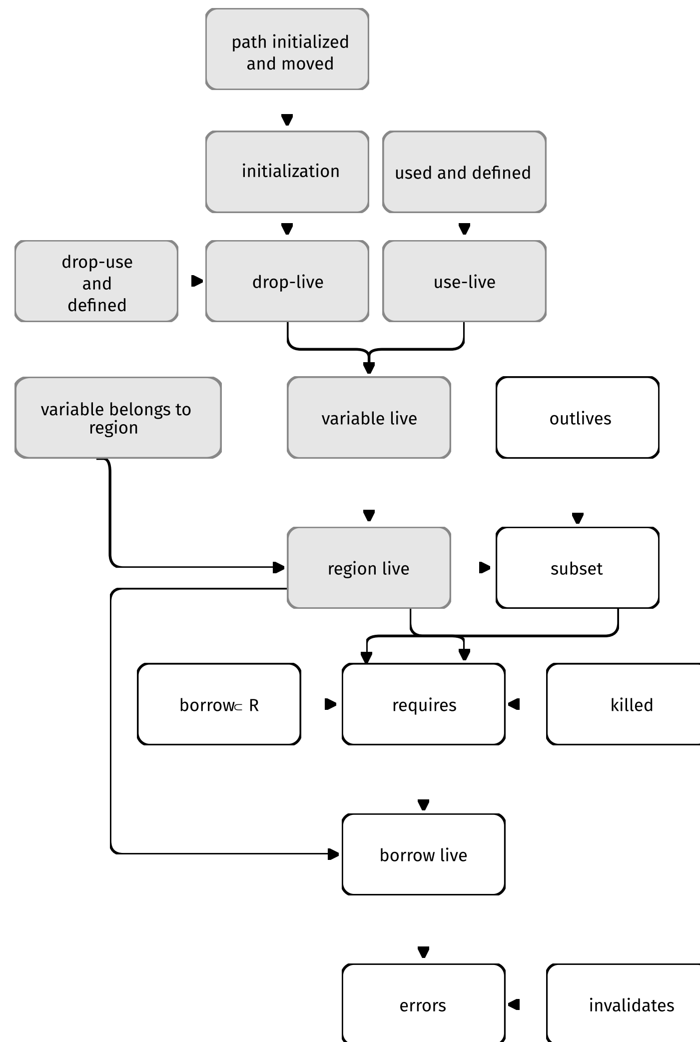
killed

borrow live

errors

invalidates

What do we need?



What about lifetime annotations?

```
let x: &'a i32;
```

Lifetime annotations everywhere

```rust
fn max_ref(a: &i32, b: &i32) -> &i32 {
  let mut max = a;
  if (*max < *b) {
    max = b;
  }
  max
}
```

Lifetime annotations everywhere

```
fn max_ref(a: &'a i32, b: &'a i32) -> &'a i32 {
    let mut max = a;
    if (*max < *b) {
        max = b;
    }
    max
}
```

Lifetime annotations everywhere

```
fn max_ref(a: &'a i32, b: &'b i32) -> &'c i32 {
  let mut max = a;
  if (*max < *b) {
    max = b;
  }
  max
}
```

Lifetime annotations everywhere

```
fn max_ref(a: &'a i32, b: &'b i32) -> &'c i32 {
  let mut max: &i32 = a;
  if (*max < *b) {
    max = b;
  }
  max
}
```

Lifetime annotations everywhere

```
fn max_ref(a: &'a i32, b: &'b i32) -> &'c i32 {
  let mut max: &'?1 i32 = a;
  if (*max < *b) {
    max = b;
  }
  max
}
```

```
max = a        'a: '?1
max = b        'b: '?1
return max     '?1: 'c
```

and
defined

drop-live

use-live

variable belongs to region

variable live

outlives

region live

subset

borrow⊏ R

requires

killed

Is it that simple?

```
Customer<'&a, Vec<(Box<dyn Dealer>, &'b mut i32)>>
```

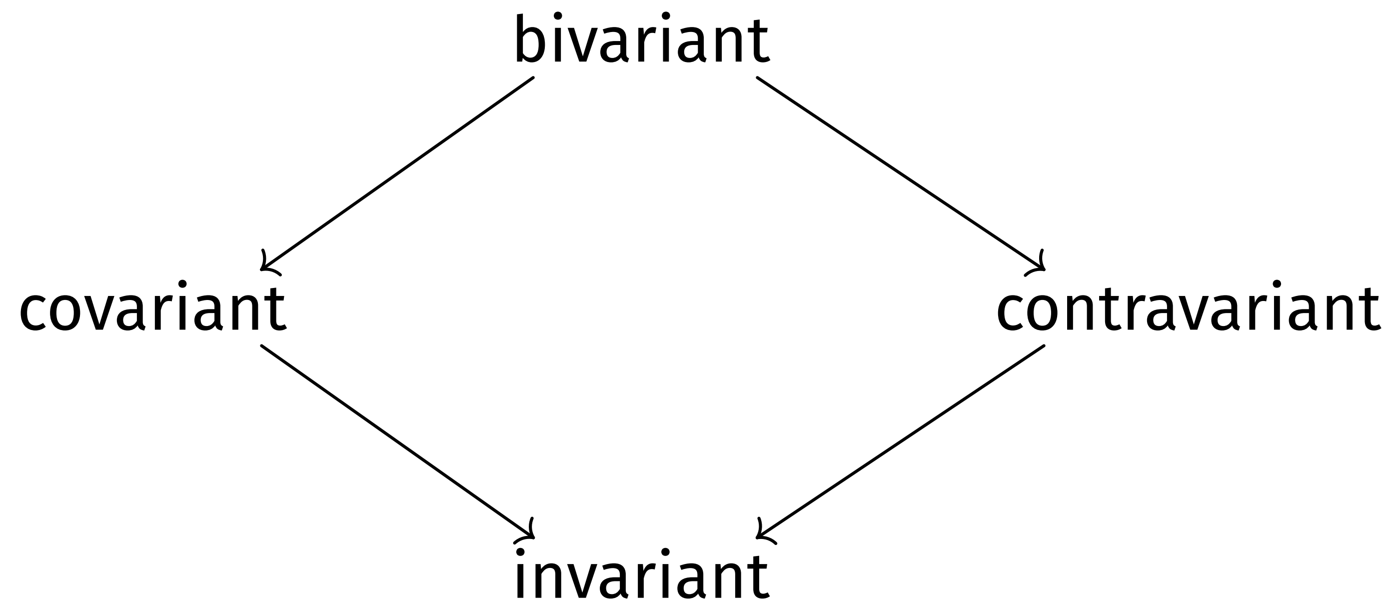
Variance

```
struct T<'a> {
  a: &'a i32,
  f: fn(&'a i32),
}
```

$$T\langle 'a \rangle \subseteq T\langle 'b \rangle$$

$$'a <?> 'b$$

Variance



Example: Variance Computation

```
struct Foo<'a, 'b, T> {
    x: &'a T,
    y: Bar<T>,
}
```

- **Collect variance info**
  - $f_0 = o$, $f_1 = o$, $f_2 = o$
  - `x` in the covariant position:
    - `&'a T` in the covariant position: $f_0 = +$ and $f_2 = +$
  - `y` in the covariant position:
    - $f_2 = \text{join}(f_2, \text{transform}(+, b_0))$

Example: Variance Computation

```
struct Foo<'a, 'b, T> {
    x: &'a T,
    y: Bar<T>,
}
```

- **Iteration 1**:
  - $f_0 = +$, $f_1 = o$, $f_2 = +$.
  - $\text{transform}(+, b_0) = -$
  - $\text{join}(*, -) = *$

Example: Variance Computation

```
struct Foo<'a, 'b, T> {
    x: &'a T,
    y: Bar<T>,
}
```

- **Iteration 2**:
  - $f_0 = +$, $f_1 = o$, $f_2 = *$.
  - $\text{transform}(+, b_0) = -$
  - $\text{join}(*, -) = *$

Example: Variance Computation

```
struct Foo<'a, 'b, T> {
    x: &'a T,
    y: Bar<T>,
}
```

- Final variances: $f_0 = +$, $f_1 = o$, $f_2 = *$:
  - f0 is evident.
  - f1 remains bivariant, as it is not mentioned in the type.
  - f2 is invariant due to its usage in both covariant and contravariant positions.

Why is it useful?

```
fn main() {
    let s = String::new();
    let x: &'static str = "hello world";
    let mut y = &*s;
    y = x;
}
```

Example: Variance in rustc

```
fn write_scope_tree(
  tcx: TyCtxt<'_>,
  body: &Body<'_>,
  scope_tree: &FxHashMap<...>,
  w: &mut dyn io::Write,
  parent: SourceScope,
  depth: usize,
) -> io::Result<()> { ... }
```

Example: Variance in rustc

```
fn write_scope_tree(
  tcx: TyCtxt<'_>,
  body: &Body<'_>,
  scope_tree: &FxHashMap<...>,
  w: &mut dyn io::Write,
  parent: SourceScope,
  depth: usize,
) -> io::Result<()> { ... }
```

```
if let ty::Adt(_, _) = local_decl.ty.kind() {
    display_adt(tcx, &mut indented_decl, local_decl.ty);
}
```

```
pub fn display_adt<'tcx>(tcx: TyCtxt<'tcx>, w: &mut
String, ty: Ty<'tcx>) {...}
```

Example: Variance in rustc

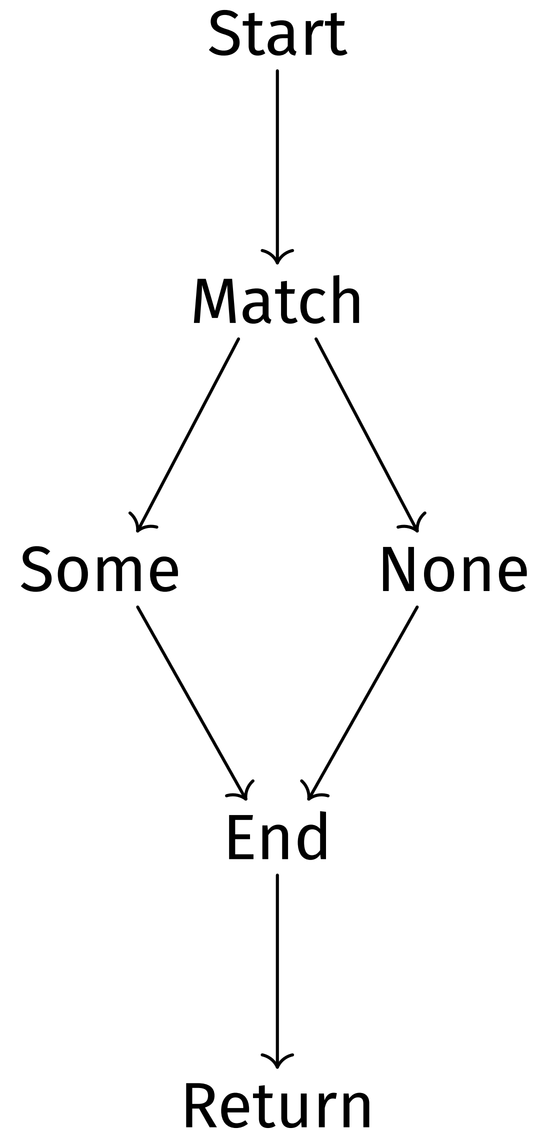
```
fn write_scope_tree<'a>(
  tcx: TyCtxt<'a>,
  body: &Body<'a>,
  scope_tree: &FxHashMap<...>,
  w: &mut dyn io::Write,
  parent: SourceScope,
  depth: usize,
) -> io::Result<()> { ... }
```

But how?

Dataflow, datalog, Polonius

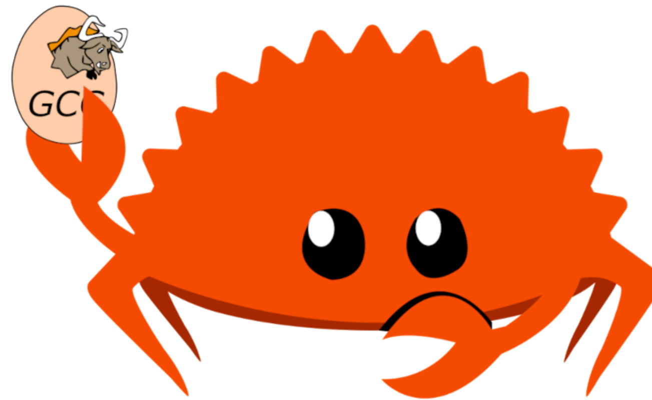
Dataflow

- Semilattice
- State
  - IN
  - OUT
- Transform function
- Iteration



Datalog Polonius

```
origin_contains_loan_on_entry(Origin, Loan, Point) :-
  loan_issued_at(Origin, Loan, Point).

origin_contains_loan_on_entry(Origin2, Loan, Point) :-
  origin_contains_loan_on_entry(Origin1, Loan, Point),
  subset(Origin1, Origin2, Point).

origin_contains_loan_on_entry(Origin, Loan, TargetPoint) :-
  origin_contains_loan_on_entry(Origin, Loan, SourcePoint),
  !loan_killed_at(Loan, SourcePoint),
  cfg_edge(SourcePoint, TargetPoint),
  (origin_live_on_entry(Origin, TargetPoint);
placeholder(Origin, _)).
```



Bonus: Rust GCC

Rust GCC

source

gccrs

AST

HIR

TyTy

GCC

GENERIC

GIMPLE

rustc

AST

HIR

TyTy

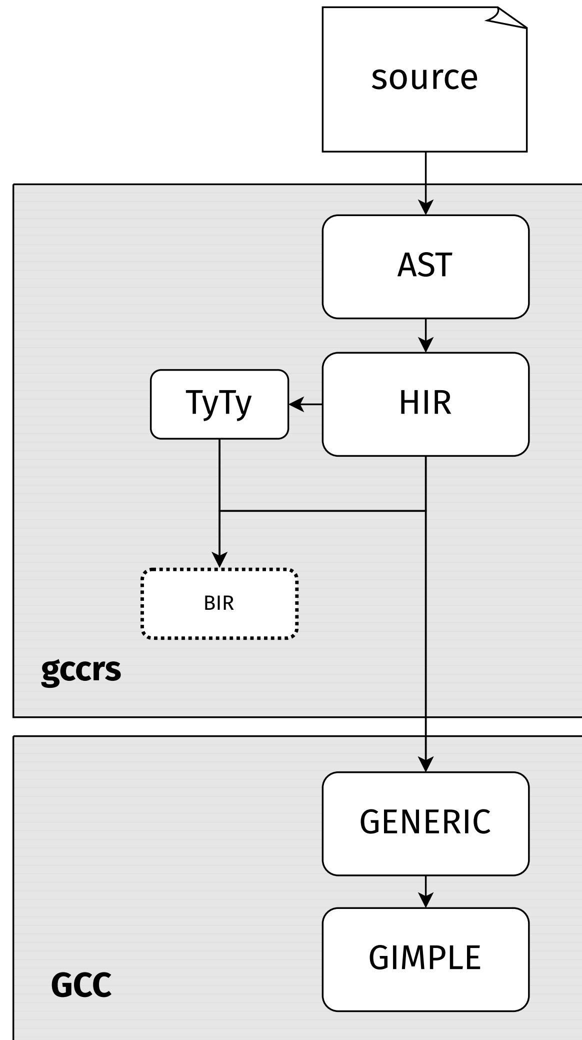
THIR

MIR

LLVM

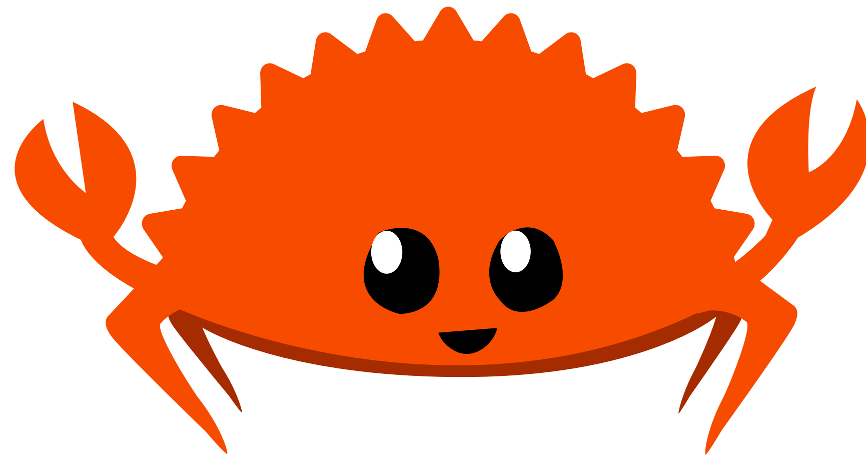
LLVM IR

Rust GCC



References

- MATSAKIS, Niko. 2094-nll. In : The Rust RFC Book. Online. Rust Foundation, 2017. [Accessed 18 December 2023]. Available from https: rust-lang.github.io/rfcs/2094-nll.html
- STJERNA, Amanda. Modelling Rust's Reference Ownership Analysis Declaratively in Datalog. Online. Master's thesis. Uppsala University, 2020. [Accessed 28 December 2023]. Available from: https://www.diva-portal.org/smash/get/diva2:1684081/fulltext01.pdf
- MATSAKIS, Niko, RAKIC, Rémy and OTHERS. The Polonius Book. 2021. Rust Foundation.
- GJENGSET, Jon. Crust of Rust: Subtyping and Variance. 2022. [Accessed 19 February 2024]. Available from https://www.youtube.com/watch?v=iVYWDIW71jk
- Rust Compiler Development Guide. Online. Rust Foundation, 2023. [Accessed 18 December 2023]. Available from https://rustc-dev-guide.rust-lang.org/index.html
- TOLVA, Karen Rustad. Original Ferris.svg. Available from https://en.wikipedia.org/wiki/File:Original_Ferris.svg



That’s all Folks!