QCon+ 案例研习社

# 线性类型的实现与应用

从 Rust 到 Move / jolestar

# 目录

1 **理解线性类型**

2 线性类型在 Rust：所有权（Ownership）& 借用（Borrowing）

3 线性类型在 Move：Ability & Resource

4 编译器的借用检查

5 类型系统与外部存储

# 理解线性类型

## 为什么要了解编译器

编程语言爱好者：新语言的改进以及创造

编程语言学习者：更快学习编程语言的使用

# 理解线性类型

## 线性逻辑

**线性逻辑**由法国数学家让·伊夫·吉拉德（Jean–Yves Girard）在 1987 年提出

用于在逻辑中表达“资源”的概念

# 理解线性类型

## 类型系统（Type System）

Ordered type ：必须按引入的顺序使用

线性类型（Linear type）：必须且只能使用一次

仿射类型（Affine type）：最多使用一次

Relevant type： 至少使用一次

普通类型（Normal type） ： 可以随意使用或丢弃

# 目录

1 理解线性类型

2 线性类型在 Rust：所有权（Ownership）& 借用（Borrowing）

3 线性类型在 Move：Ability & Resource

4 编译器的借用检查

5 类型系统与外部存储

# 线性类型在 Rust

所有权（Ownership）

手动内存管理

垃圾回收

所有权

每一个值（内存空间）都有一个所有者（变量），并且只能有一个

当所有者离开作用域时，值将被丢弃（内存回收）

# 线性类型在 Rust

## 所有权（Ownership）

```
struct MyStruct{
    x: u64
}

#[test]
fn test_ownership(){
    let x: u64 = 1;
    take_ownership(x); //copy
    let x1 = x;     //copy
    println!("{} {}", x, x1);

    let s = MyStruct{
        x
    };
    take_ownership(s); //move
    let x2 = s.x;  //error: value used here after move
    println!("{}", x2);
}

fn take_ownership<T>(_v: T){
}
```

# 线性类型在 Rust

## 借用（Borrowing）

```
#[test]
fn test_borrow1(){
    let mut x: u64 = 1;
    let x1 = &x;
    let x2 = &x;  // multi immutable borrow is ok
    let x3 = &mut x; //error[E0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
    println!("{} {} {} {}", x, x1, x2, x3);
}

fn dangle() -> &MyStruct{
    let s = MyStruct{ x: 1};
    &s  //error: this function's return type contains a borrowed value, but there is no value for it to be borrowed from.
}
```

多个不可变引用不冲突

可变引用是独占的

引用未释放之前不能 Move

引用指向的值必须是有效的（悬垂引用）

# 线性类型在 Rust

## 生命周期（Lifetime）

```
#[test]
fn test_lifetime(){
    let x: u64 = 1;
    let mut x1 = &x;
    {
        let y: u64 = 2;
        x1 = &y;  //error: borrowed value does not live long enough
    }
    println!("{}", x1);
}
```

# 线性类型在 Rust

## 生命周期（Lifetime）

```
fn borrow(_s: &MyStruct, x: &u64)-> &u64{
    x
}
//error[E0106]: missing lifetime specifier

#[test]
fn test_life_time(){
    let s = MyStruct{
        x: 1,
    };
    let x = 2;
    let x1 = borrow(&s, &x);
    let s1 = s;
    println!("{} {} {} {}", s.x, x, x1, s1.x);
}
```

# 线性类型在 Rust

## Drop

```
#[test]
fn test_drop(){
    let s = MyStruct{ x: 1};  // warning: unused variable: `s`
}
```

Rust 允许隐式 drop， 严格的说应该是仿射类型（Affine type）

# 目录

1 理解线性类型

2 线性类型在 Rust：所有权（Ownership）& 借用（Borrowing）

3 线性类型在 Move：Ability & Resource

4 编译器的借用检查

5 类型系统与外部存储

# 线性类型在 Move

## Drop

```
struct MyStruct{
    x: u64,
}

#[test]
fun test_drop(){
    let s = MyStruct{ x: 1};  // error[E06001]: unused value without ‘drop’
}
```

```
struct MyStruct has drop {
    x: u64,
}

#[test]
fun test_drop(){
    let s = MyStruct{ x: 1};  // warning[W09002]: unused variable
}
```

# 线性类型在 Move

## Copy

```
struct MyStruct has drop, copy, store{
        x: u64,
}

#[test]
public fun copy_test(){
    let t = MyStruct{x: 1};
    let t1 = copy t;
    let t2 = copy t;
    Debug::print(&t);
}
```

# 目录

1 理解线性类型

2 线性类型在 Rust：所有权（Ownership）& 借用（Borrowing）

3 线性类型在 Move：Ability & Resource

4 **编译器的借用检查**

5 类型系统与外部存储

# 借用检查

以 Move 为例

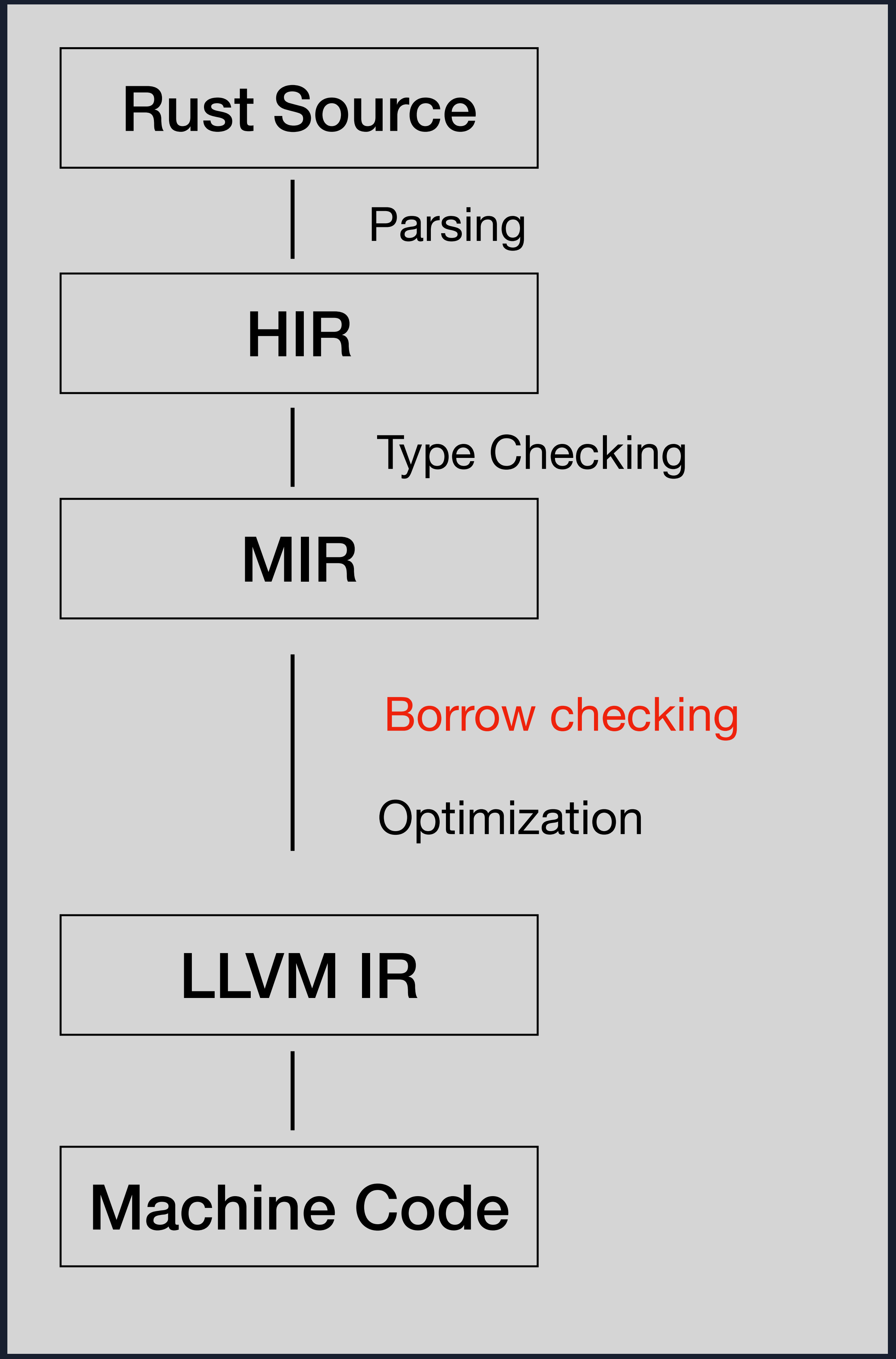
借用检查的时机

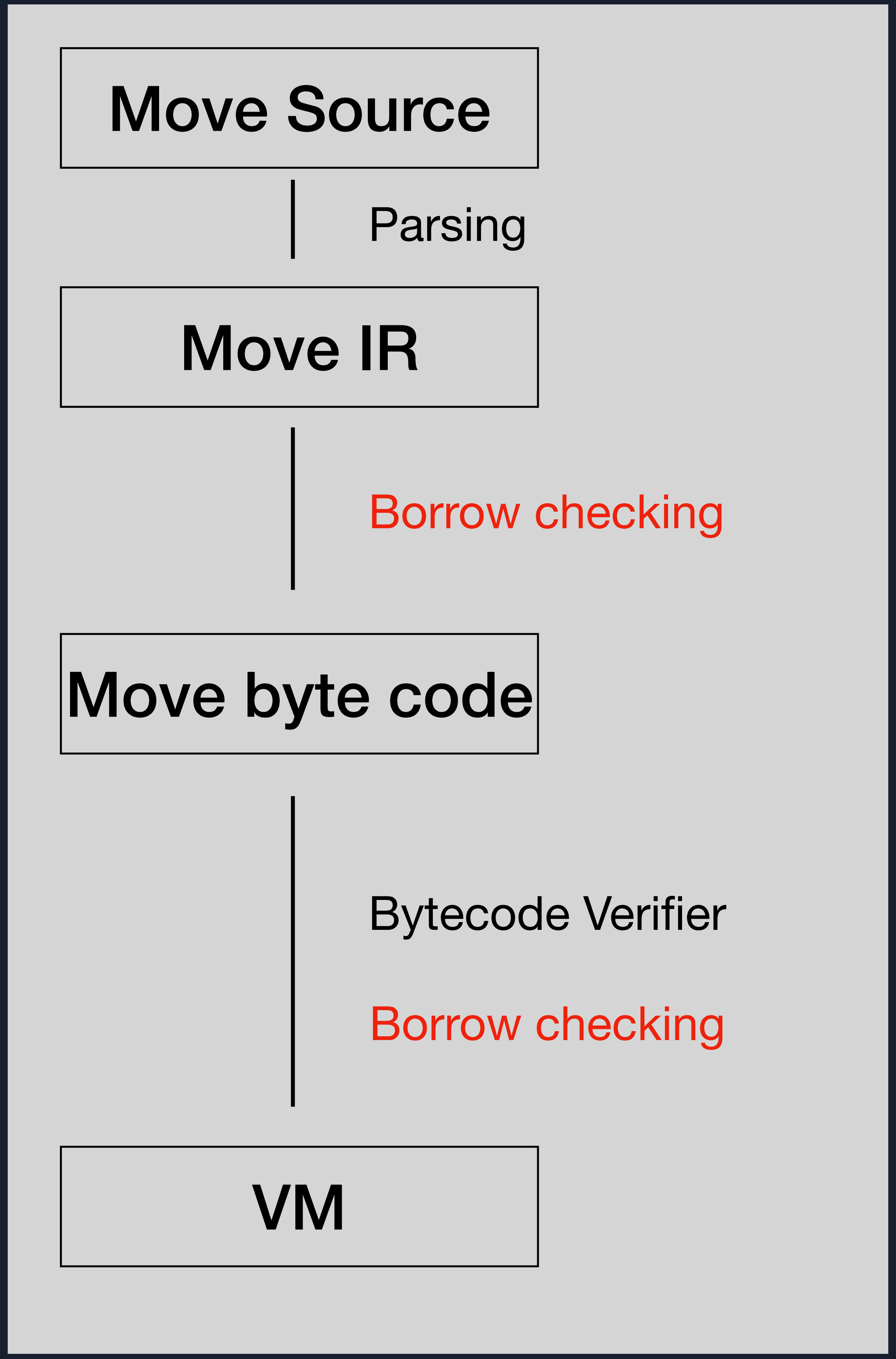
借用检查的范围

借用检查的实现

# 借用检查

借用检查的时机：编译期，字节码运行期





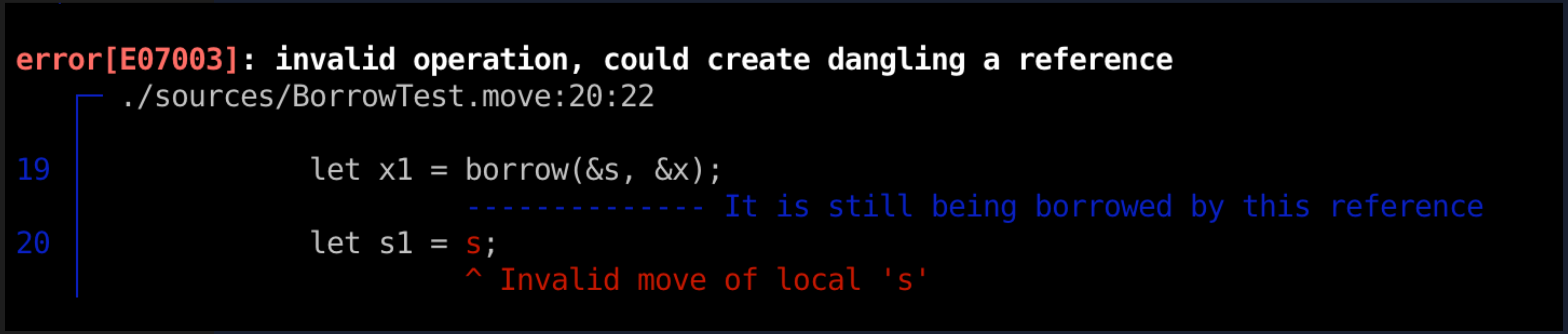
Mid–level Intermediate Representation

# 借用检查

## 借用检查的范围：同一个方法内

```
fun borrow(_s: &MyStruct, x: &u64): &u64{
    x
}

#[test]
fun test_borrow() {
    let s = MyStruct{
        x: 1,
    };
    let x = 2;
    let x1 = borrow(&s, &x);
    let s1 = s;
    Debug::print(&s1);
    Debug::print(x1);
}
```



```
error[E07003]: invalid operation, could create dangling a reference
    ┌─ ./sources/BorrowTest.move:20:22
    │
19  │             let x1 = borrow(&s, &x);
    │                      -------------- It is still being borrowed by this reference
20  │             let s1 = s;
    │                      ^ Invalid move of local 's'
```

# 借用检查

## 借用检查的实现：RefID & BorrowGraph

给每个 Ref 分配一个 ID：RefID

构造 BorrowGraph，追踪 Ref 的生命周期，创建以及销毁

在 Ref 创建，Move 时进行检查

方法返回时保证没有指向 local 变量的引用

# 借用检查

## 借用检查的实现：引用的生命周期

```
public fun borrow_test1(){
    let s = MyStruct{x: 1};

    let s_ref = &s;
    let x_ref = &s_ref.x;
    Debug::print(x_ref);
    Debug::print(x_ref);

    let s_ref = &mut s;
    s_ref.x = 2;
}

public fun borrow_test2(s: &MyStruct): &u64{
    &s.x
}
```

```
public borrow_test1() {
L0:     s: MyStruct
L1:     s_ref: &MyStruct
L2:     s_ref#1: &mut MyStruct
L3:     x_ref: &u64
B0:
        0: LdU64(1)
        1: Pack[0](MyStruct)
        2: StLoc[0](s: MyStruct)
        3: ImmBorrowLoc[0](s: MyStruct)
        4: StLoc[1](s_ref: &MyStruct)
        5: MoveLoc[1](s_ref: &MyStruct)
        6: ImmBorrowField[0](MyStruct.x: u64)
        7: StLoc[3](x_ref: &u64)
        8: CopyLoc[3](x_ref: &u64)
        9: Call[0](print<u64>(&u64))
        10: MoveLoc[3](x_ref: &u64)
        11: Call[0](print<u64>(&u64))
        12: MutBorrowLoc[0](s: MyStruct)
        13: StLoc[2](s_ref#1: &mut MyStruct)
        14: LdU64(2)
        15: MoveLoc[2](s_ref#1: &mut MyStruct)
        16: MutBorrowField[0](MyStruct.x: u64)
        17: WriteRef
        18: Ret
}

public borrow_test2(): &u64 {
B0:
        0: MoveLoc[0](s: &MyStruct)
        1: ImmBorrowField[0](MyStruct.x: u64)
        2: Ret
}
```

# 借用检查

## 借用检查的实现：引用的生命周期

```
public fun ref_test(){
    let s = MyStruct{x: 1};
    let x_ref = &s.x;
    let y = *x_ref;
    Debug::print(&y);
    let s_mut_ref = &mut s;
    s_mut_ref.x = 2;
}
```

```
public ref_test() {
L0:     s: MyStruct
L1:     s_mut_ref: &mut MyStruct
L2:     x_ref: &u64
L3:     y: u64
B0:
        0: LdU64(1)
        1: Pack[0](MyStruct)
        2: StLoc[0](s: MyStruct)
        3: ImmBorrowLoc[0](s: MyStruct)
        4: ImmBorrowField[0](MyStruct.x: u64)
        5: StLoc[2](x_ref: &u64)
        6: MoveLoc[2](x_ref: &u64)
        7: ReadRef
        8: StLoc[3](y: u64)
        9: ImmBorrowLoc[3](y: u64)
        10: Call[0](print<u64>(&u64))
        11: MutBorrowLoc[0](s: MyStruct)
        12: StLoc[1](s_mut_ref: &mut MyStruct)
        13: LdU64(2)
        14: MoveLoc[1](s_mut_ref: &mut MyStruct)
        15: MutBorrowField[0](MyStruct.x: u64)
        16: WriteRef
        17: Ret
}
```

# 目录

1 理解线性类型

2 线性类型在 Rust：所有权（Ownership）& 借用（Borrowing）

3 线性类型在 Move：Ability & Resource

4 编译器的借用检查

5 类型系统与外部存储

# 类型系统与外部存储

## Store

```
struct LockCap {} //No copy, store, drop

public fun lock(s: &signer): LockCap{
    //访问权限检查
}

// 类型系统保证调用方必须还回来
public fun unlock(lock: LockCap){}
```

# 类型系统与外部存储

## 模拟物理世界

```
struct Token has store{ value: u64}

public fun transfer(sender: &signer, to: address, amount: u64){
    //从 sender 余额 - amount
    //给 to 余额 + amount
}

public fun withdraw(sender: &signer): Token{
    //从 sender 余额 - amount
}

public fun deposit(to: address, token:Token){
    //给 to 余额 + amount
}

struct MyBox has store,key{
    token: Token,
}

// Withdraw and save to MyBox
```

# 总结页

线性类型在编程语言中的应用

线性类型在编译器中的实现方式

参考资料：

1. https://arxiv.org/abs/2205.05181  The Move Borrow Checker 的一篇论文
2. https://github.com/move-language/move/issues/210
2. https://github.com/move-language/move/tree/main/language/move-borrow-graph
3. https://github.com/starcoinorg/starcoin-framework/blob/main/sources/Account.move

# 另外一个新的编程语言的黄金时代

# THANKS

QCon⁺ 案例研习社