# Rustライブラリのスレッド安全性に関するバグのプログラム合成による発見

岡田 大空, 松下 祐介, 五十嵐 淳 (京都大学)

## 背景: RustのSync・Send境界バグ

**スレッド安全性**のバグは致命的になりうる & 発見が困難
→ Rustでは専用の**トレイト Sync・Send**を用いて対処

**スレッド間共有**可能　　**スレッド間移動**可能

Sync・Sendの付与方法は2通り
- safe — コンパイラがスレッド安全性の解析結果に応じ自動付与
- unsafe — プログラマーが手動付与 → **Sync・Send境界バグ**

**既存研究** Rudra [Bae+ '21]
静的解析を用いてSync・Send境界バグが疑われる箇所を推測

スレッド安全なスマートポインタ

**例**　BuggyArc: Arcと同様のAPIを持つ型

BuggyArc<T>がスレッド間共有可能

```
unsafe impl<T> Sync for BuggyArc<T> {}
```

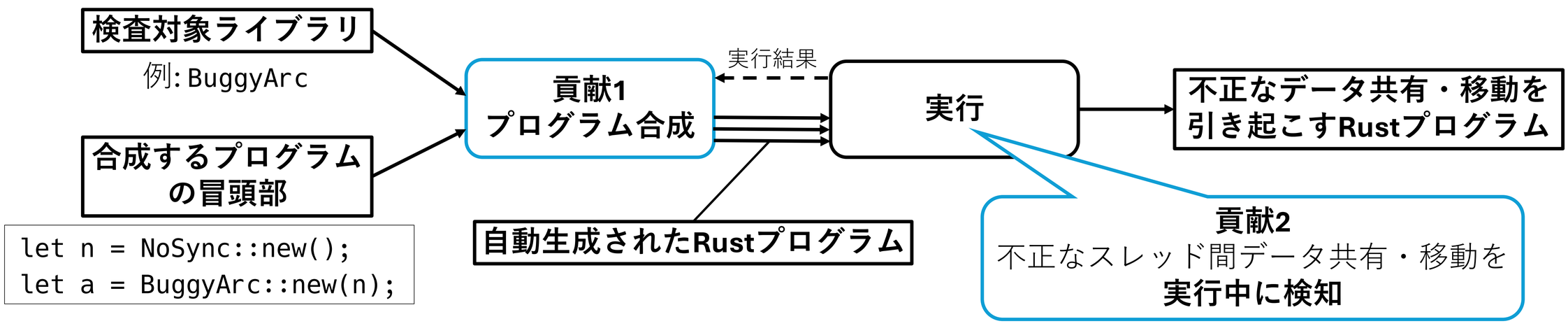
正しくは `T: Sync + Send`

&BuggyArc<T>から**&T**が取り出せる
→ &Tが複数スレッドで**共有**される
→ **アクセス競合**が起こりうる

e.g. `Cell<[i32; 10000]>: !Sync`

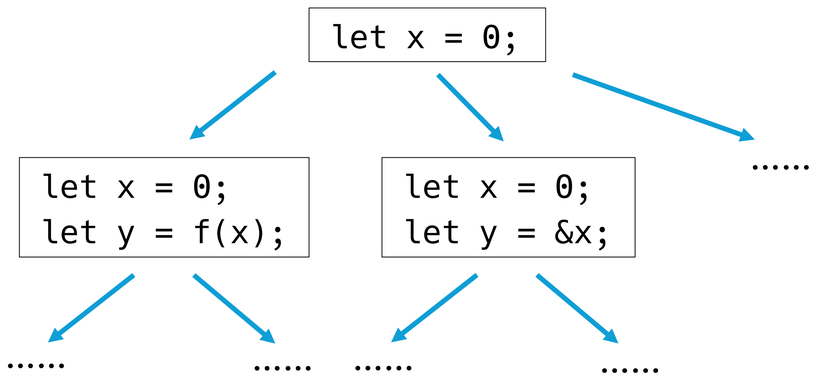
## 本研究

**RustライブラリのSync・Send境界バグ**を動的に発見:
テストプログラムを**合成**、!Sync・!Sendの違反を**動的検知**




## 貢献1: マルチスレッドRustプログラムの合成

- 各スレッドごとにAPI呼び出し列を合成
- **インクリメンタルな合成**:
  以前合成した列の末尾へのAPI呼び出しの追加を繰り返す
- シングルスレッドRustプログラムの合成アルゴリズム
  Crabtree [Takashima+ '24] を参考とした




## 貢献2: 不正なデータ共有・移動の動的検知

```
fn main() {
    let n = NoSync::new();
    let a = BuggyArc::new(n);

    let waiter = &DropWaiter::new(2);
    std::thread::scope(|scope| {
        let mut v_2 = &a;
        scope.spawn(move || {
            let mut v_5 = BuggyArc::new(v_2);
            let mut v_6 = v_5.access();
            waiter.wait(vec![v_6]);
        });

        let mut v_1 = &a;
        let mut v_3 = BuggyArc::new(v_1);
        let mut v_4 = v_3.access();
        waiter.wait(vec![v_4]);
    });
}
```

T1　T2

- 動的検知用の型 NoSync:
  自身が共有されている
  スレッドの集合$S$を記録
- NoSync: !Sync

① $S \leftarrow \{T1\}$

② $S \leftarrow \{T1, T2\}$

!Syncなnが複数スレッドに共有されている
→ **不正なデータ共有を検知**

## 現時点での成果

- 予備実験: 現実のバグに対して動的検知を行うプログラムを手で作成
  - `FuturesUnordered` (futures)
  - `MappedMutexGuard` (parking_lot)
- 本手法の簡易的な実装
- 人工的な例に対してテストプログラムの自動合成を行う実験
  - `BuggyArc`・`BuggyCell`

## 今後の課題

- 動的検知の正当性の証明
- プログラム合成の高速化
- スレッド実行順序による非決定性の対処
- より現実的な例への適用