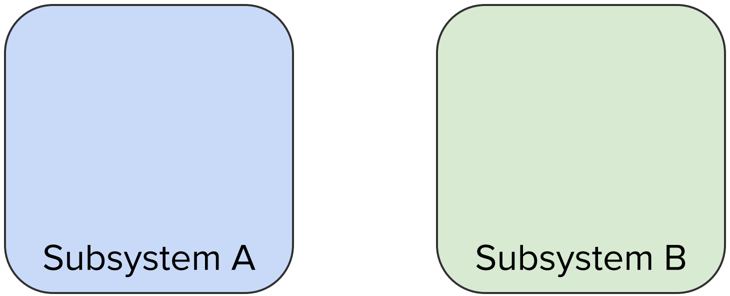
Extending Rust with Support for Zero Copy Communication

Arthur Lafrance[1], David Detweiler[1], Zhaofeng Li[2], Xiangdong Chen[2], Vikram Narayanan[2], Anton Burtsev[2]

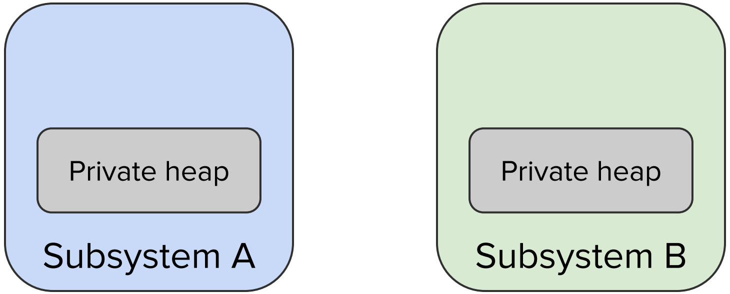
[1] University of California, Irvine [2] University of Utah

PLOS 2023

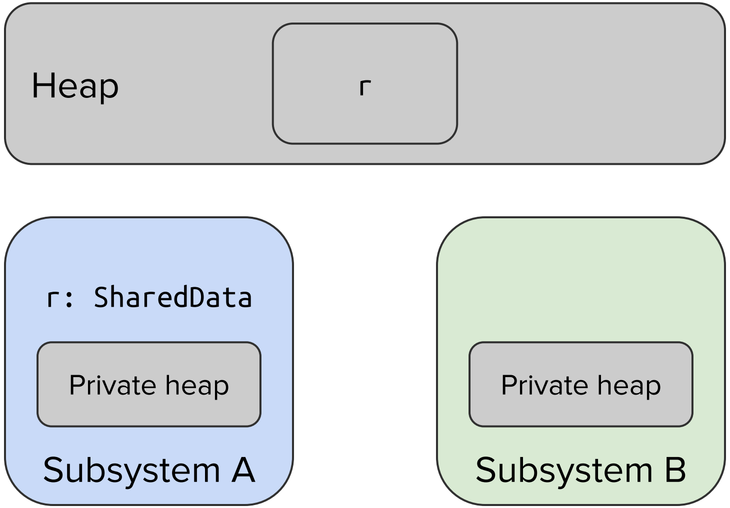
Language-Based Isolation



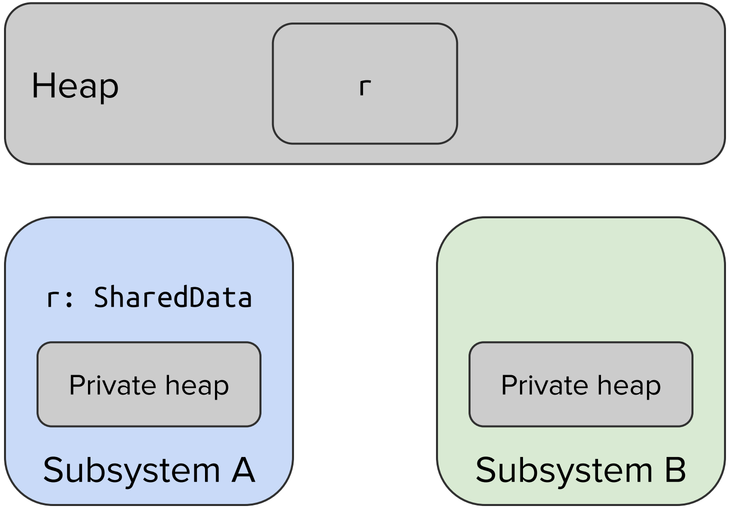
Language-Based Isolation



Language-Based Isolation

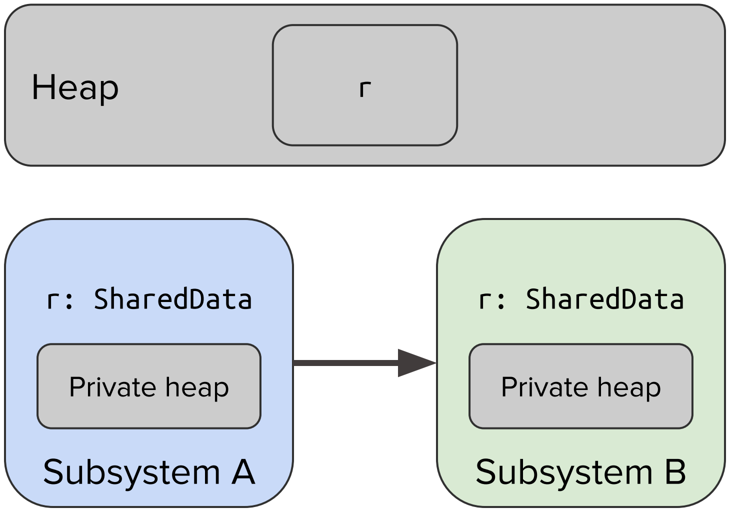


Language-Based Isolation



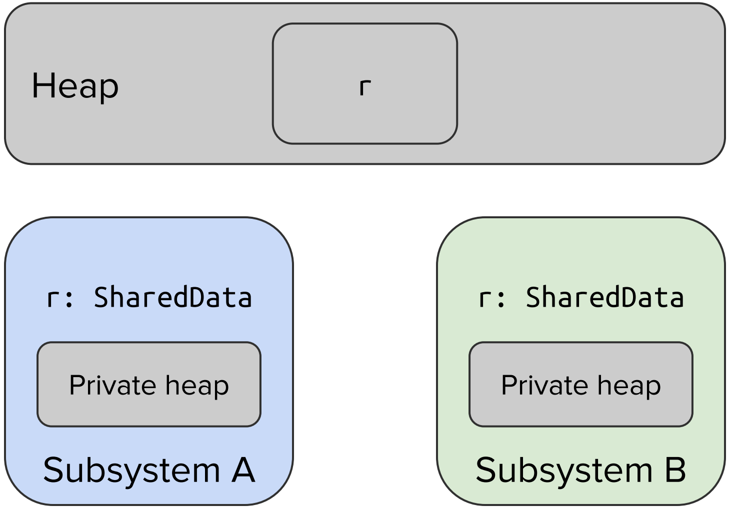
Language-enforced access control

Language-Based Isolation



Language-enforced access control

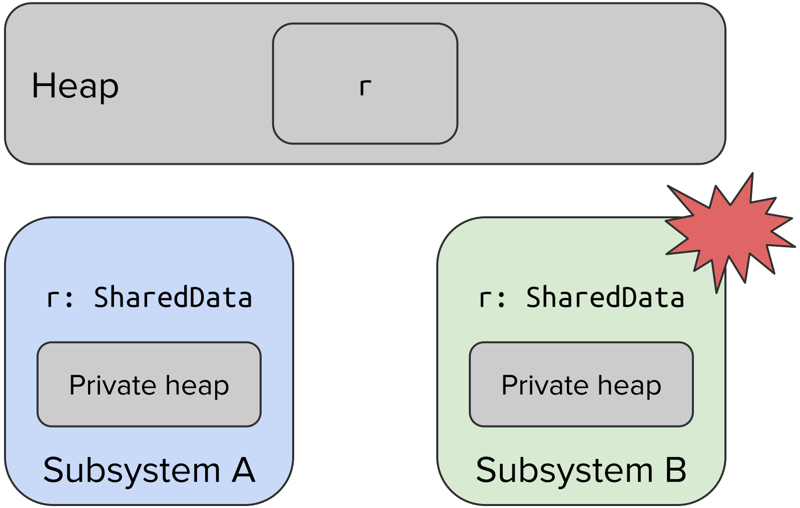
Language-Based Isolation



Language-enforced access control

Zero copy communication

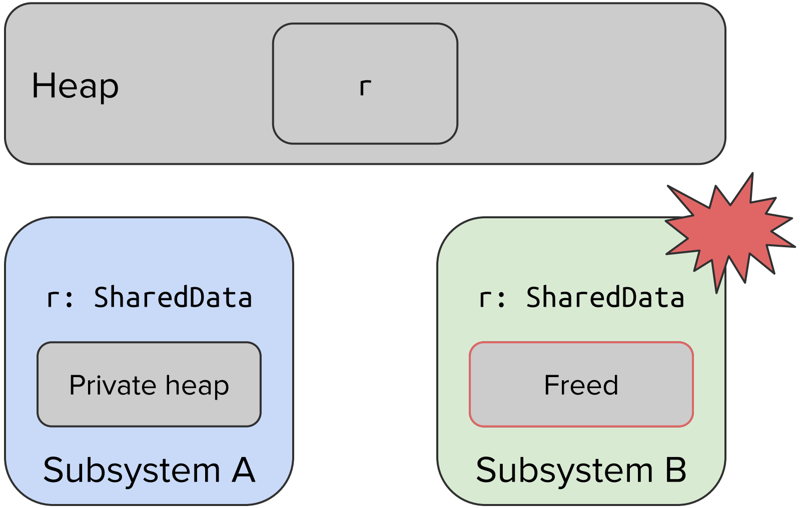
Language-Based Isolation



Language-enforced access control

Zero copy communication

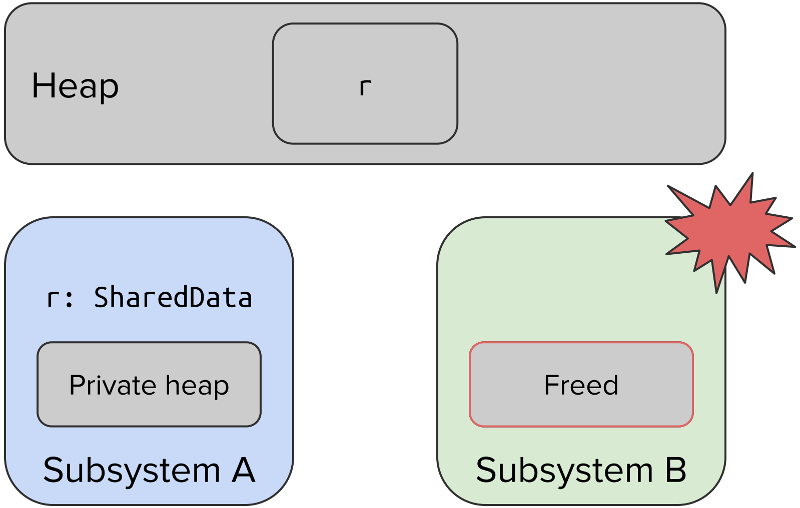
Language-Based Isolation



Language-enforced access control

Zero copy communication

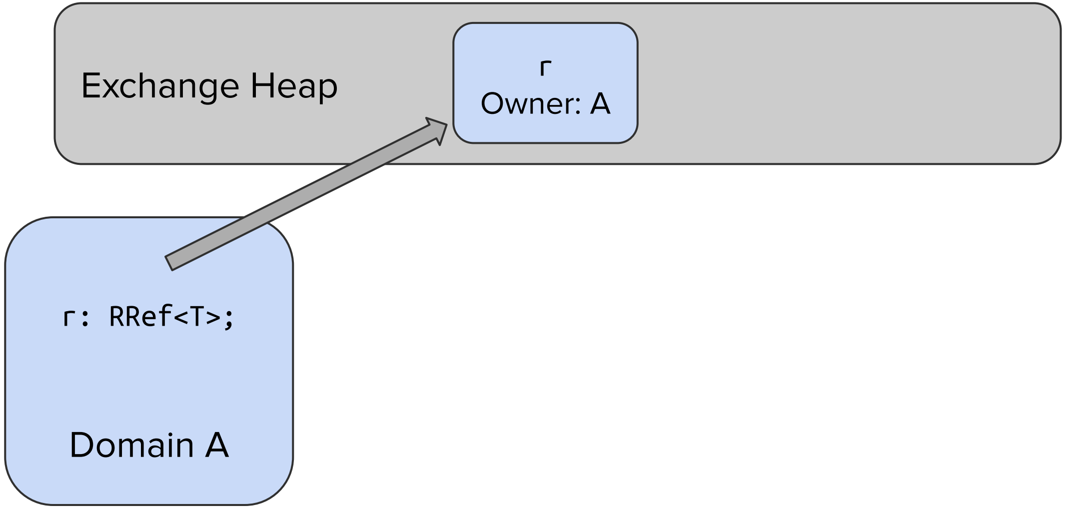
Language-Based Isolation



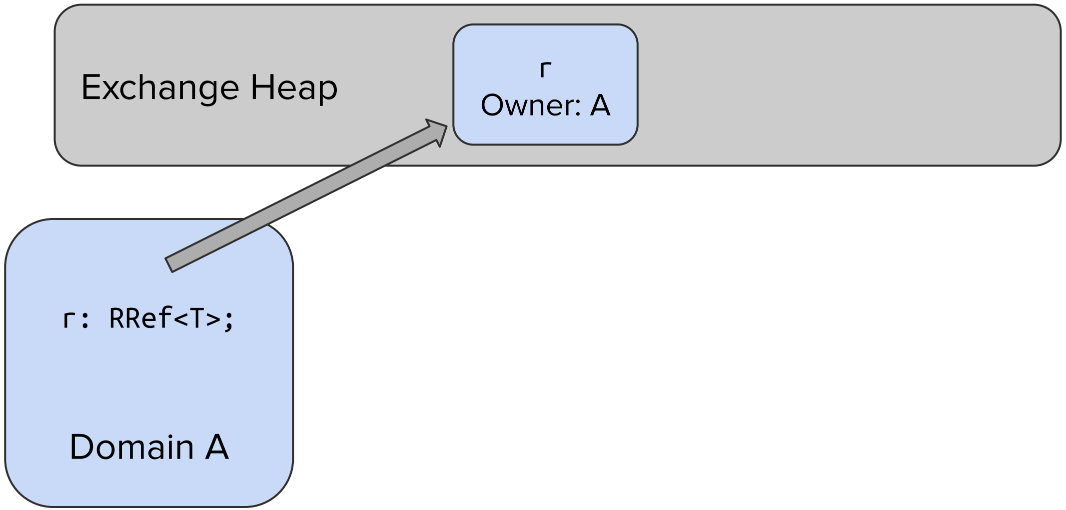
Language-enforced access control

Zero copy communication

RedLeaf

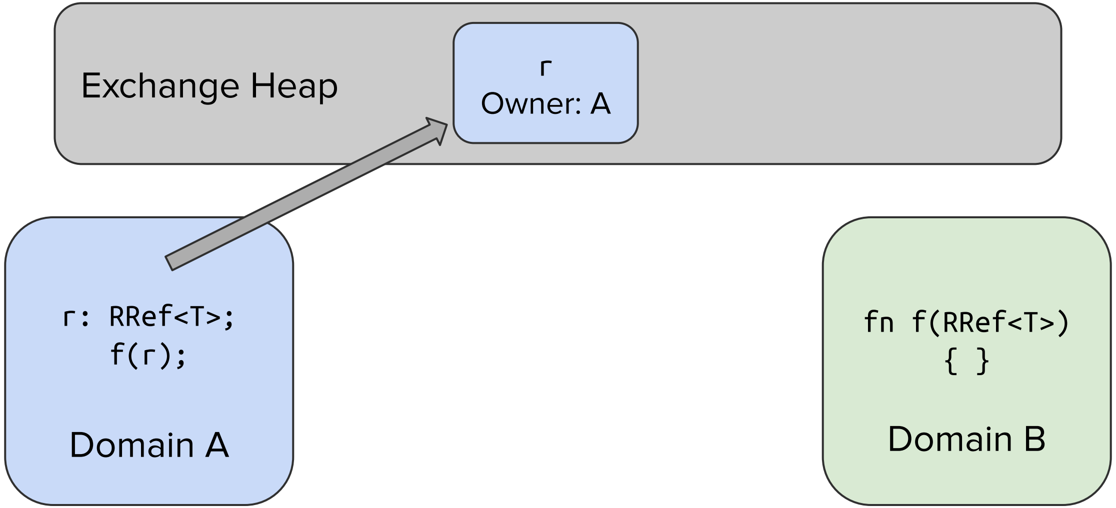


RedLeaf



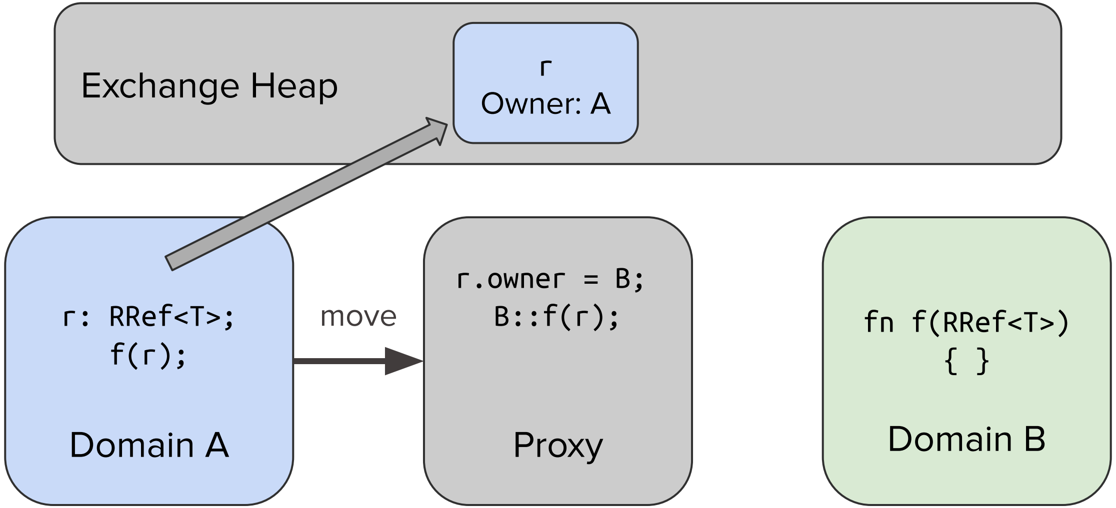
RRef: Single ownership shared data

RedLeaf



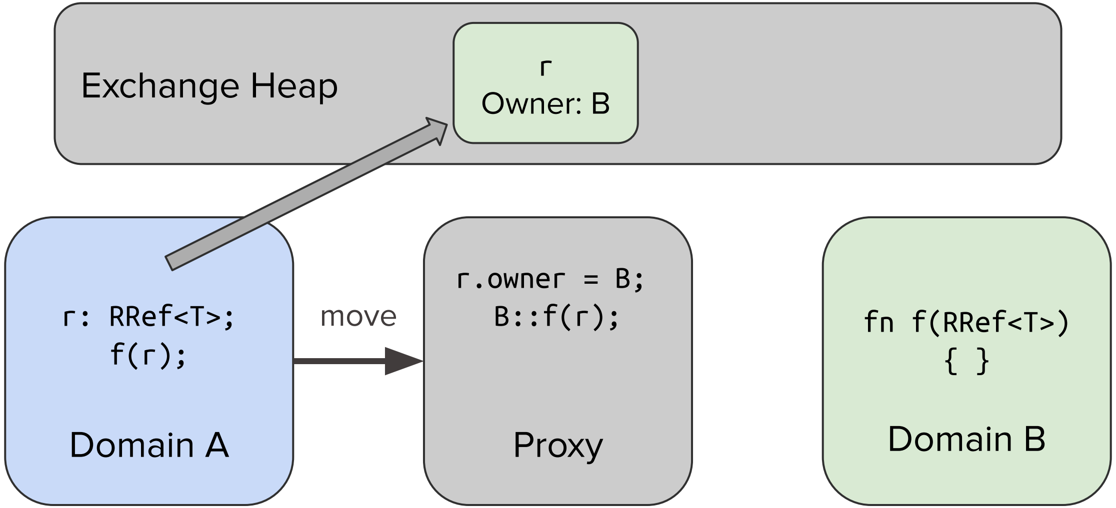
RRef: Single ownership shared data

RedLeaf



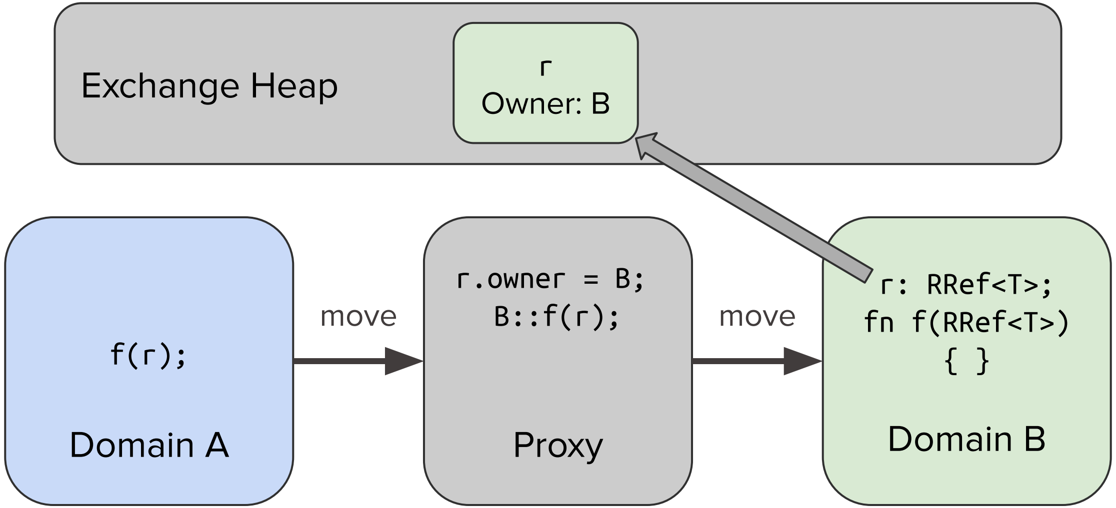
RRef: Single ownership shared data

RedLeaf



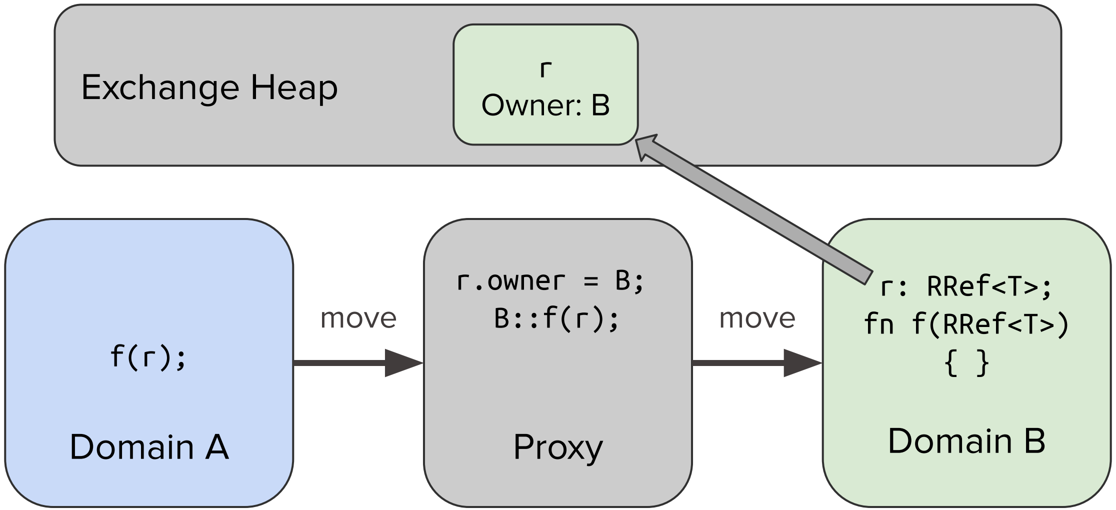
RRef: Single ownership shared data

RedLeaf



RRef: Single ownership shared data

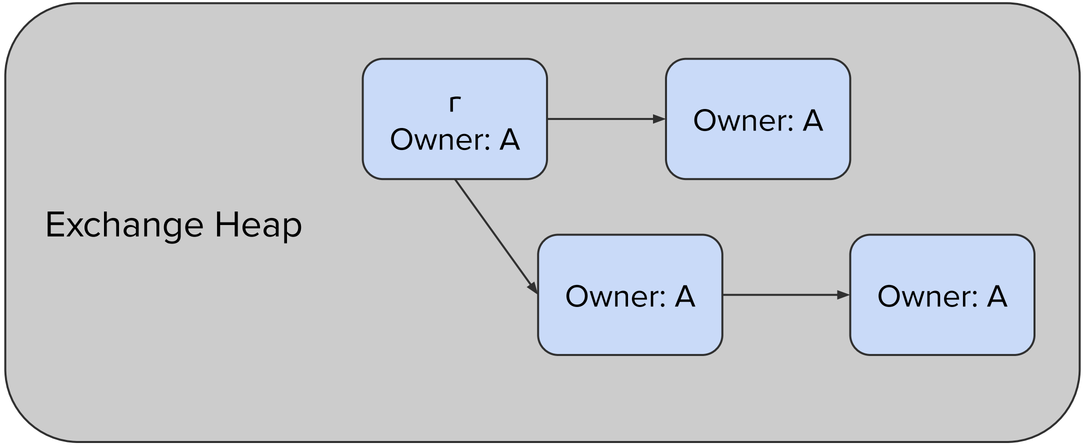
RedLeaf



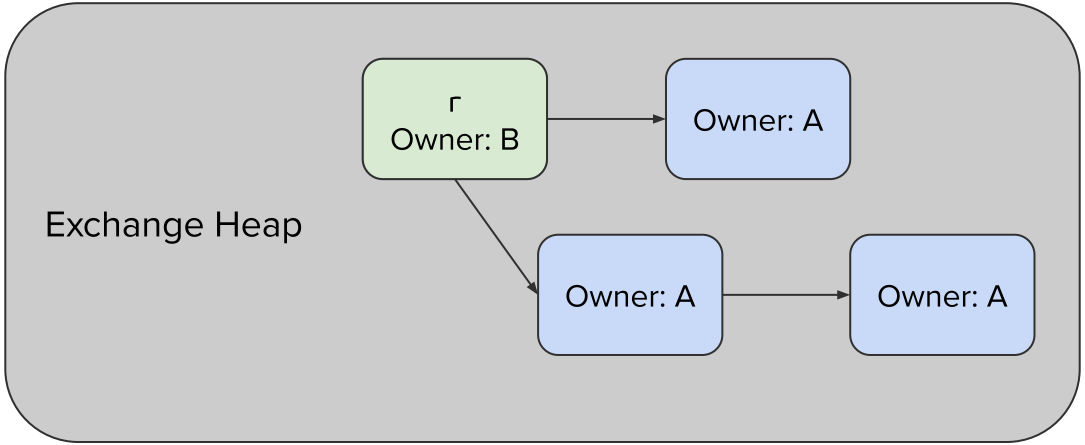
RRef: Single ownership shared data

Ownership update upon IPC

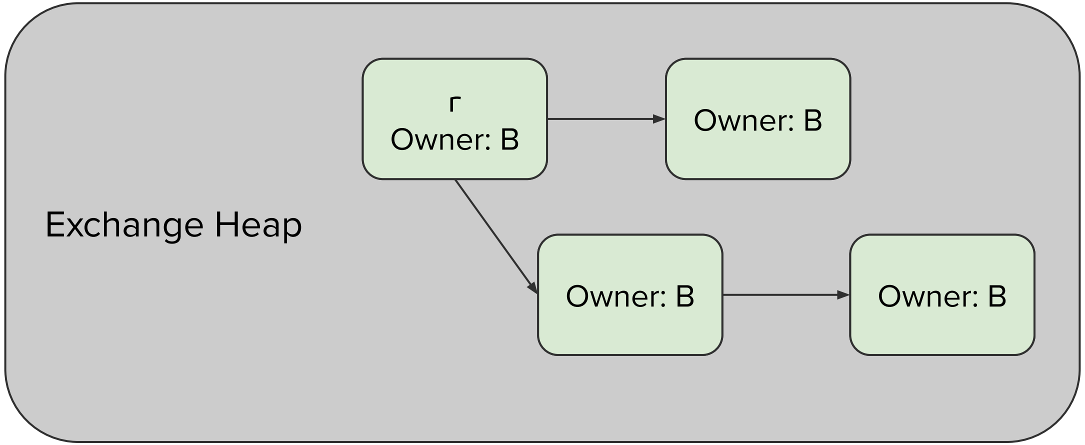
Nested RRefs



Nested RRefs

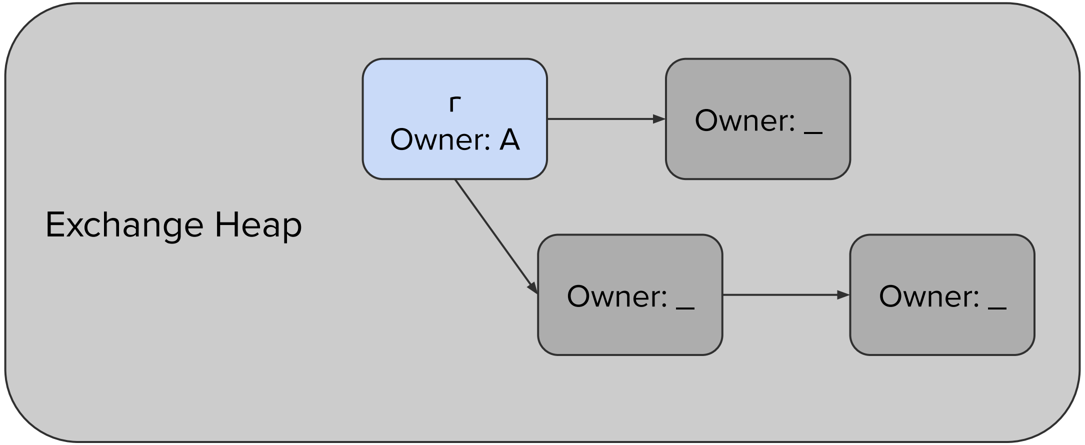


Nested RRefs



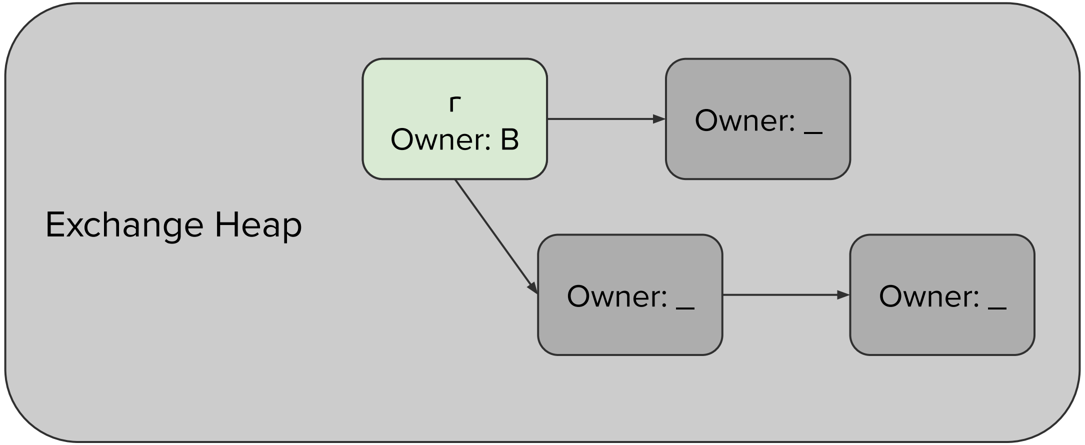
Costly ownership updates

Nested RRefs



Better idea: Track ownership at **roots** only

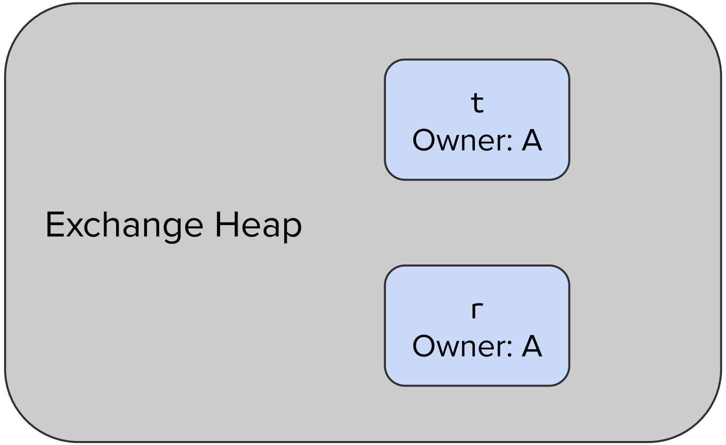
Nested RRefs



Better idea: Track ownership at **roots** only

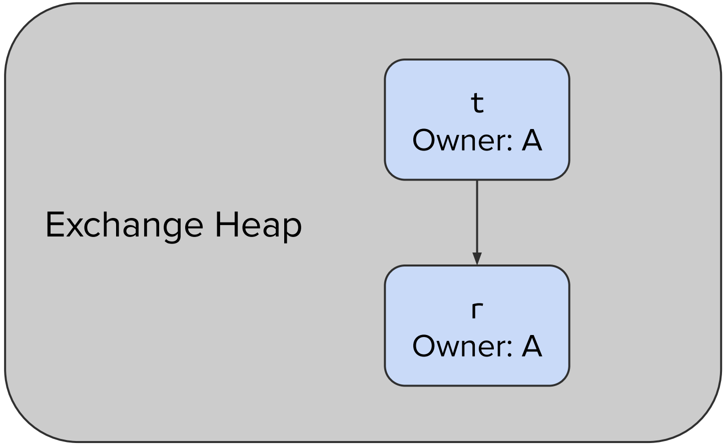
Problem: Maintaining “Bottomness”

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut t: RRef<T>) {
    let r = RRef::new(1);


}
```



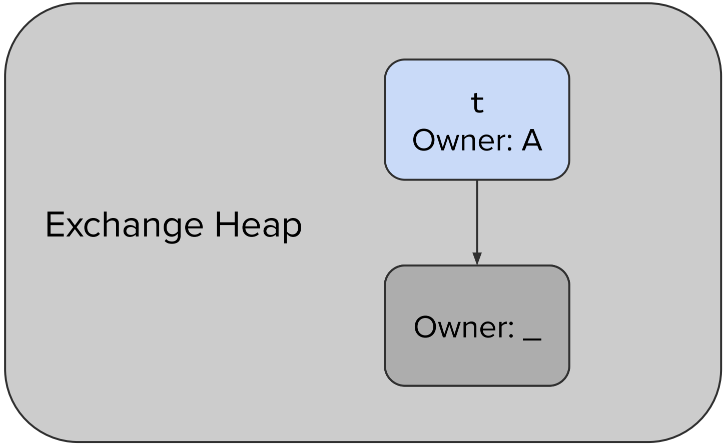
Problem: Maintaining "Bottomness"

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut t: RRef<T>) {
    let r = RRef::new(1);
    t.r = r;

}
```



Problem: Maintaining “Bottomness”

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut t: RRef<T>) {
    let r = RRef::new(1);
    t.r = r;
    mark_bottom(&mut t.r);
}
```



Assignment Analysis

Static analysis: Did nesting change?

```
t.r = r;
mark_bottom(&mut t.r);
```

Codegen: Update ownership

Composite Types

```
struct T {
    r: RRef<E>,
}

enum E {
    Zero,
    One(RRef<i32>),
    Two(RRef<i32>, RRef<i32>),
}
```

```
fn move_in(mut t: RRef<T>, r: RRef<E>) {
    t.r = r;







}
```

Composite Types

```
struct T {
    r: RRef<E>,
}

enum E {
    Zero,
    One(RRef<i32>),
    Two(RRef<i32>, RRef<i32>),
}
```

```
fn move_in(mut t: RRef<T>, r: RRef<E>) {
    t.r = r;
    match t.r {
        E::One(ref mut r) => mark_bottom(r),




    }
}
```

Composite Types

```
struct T {
    r: RRef<E>,
}

enum E {
    Zero,
    One(RRef<i32>),
    Two(RRef<i32>, RRef<i32>),
}
```

```
fn move_in(mut t: RRef<T>, r: RRef<E>) {
    t.r = r;
    match t.r {
        E::One(ref mut r) => mark_bottom(r),
        E::Two(ref mut r1, ref mut r2) => {
            mark_bottom(r1);
            mark_bottom(r2);
        },

    }
}
```

Composite Types

```
struct T {
    r: RRef<E>,
}

enum E {
    Zero,
    One(RRef<i32>),
    Two(RRef<i32>, RRef<i32>),
}
```

```
fn move_in(mut t: RRef<T>, r: RRef<E>) {
    t.r = r;
    match t.r {
        E::One(ref mut r) => mark_bottom(r),
        E::Two(ref mut r1, ref mut r2) => {
            mark_bottom(r1);
            mark_bottom(r2);
        },
        _ => {},
    }
}
```

Dynamic decision making

Mutable Borrows

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut rt: RRef<T>) {
    let t = &mut *rt;



}
```

Borrow **from** RRef

Mutable Borrows

```
struct T {
   r: RRef<i32>,
}

fn move_in(mut rt: RRef<T>) {
   let t = &mut *rt;
   let r = RRef::new(1);
   t.r = r;
   mark_bottom(&mut t.r);
}
```

Borrow **from** RRef

Mutable Borrows

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut rt: RRef<T>) {
    let t = &mut *rt;
    let r = RRef::new(1);
    t.r = r;
    mark_bottom(&mut t.r);
}
```

Borrow **from** RRef

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut t: T) {
    let t = &mut t;
}
```

Borrow **without** RRef

Mutable Borrows

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut rt: RRef<T>) {
    let t = &mut *rt;
    let r = RRef::new(1);
    t.r = r;
    mark_bottom(&mut t.r);
}
```

Borrow **from** RRef

```
struct T {
    r: RRef<i32>,
}

fn move_in(mut t: T) {
    let t = &mut t;
    let r = RRef::new(1);
    t.r = r;
    // no need to mark bottom
}
```

Borrow **without** RRef

Analysis loses **locality**

Interprocedural Analysis

```
struct T {
    r: RRef<i32>,
}

fn move_in(t: &mut T) {
    let r = RRef::new(1);
    t.r = r;

}
```

Interprocedural Analysis

```
struct T {
    r: RRef<i32>,
}

fn move_in(t: &mut T) {
    let r = RRef::new(1);
    t.r = r;
    // mark bottom here?
}
```

Interprocedural Analysis

```
struct T {
    r: RRef<i32>,
}

fn move_in(t: &mut T) {
    let r = RRef::new(1);
    t.r = r;
    // mark bottom here?
}
```

```
fn within_rref(mut t: RRef<T>) {
    move_in(&mut *t);
    // must mark bottom
    mark_bottom(&mut t.r);
}
```

Interprocedural Analysis

```
struct T {
    r: RRef<i32>,
}

fn move_in(t: &mut T) {
    let r = RRef::new(1);
    t.r = r;
    // mark bottom here?
}
```

```
fn within_rref(mut t: RRef<T>) {
    move_in(&mut *t);
    // must mark bottom
    mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
    move_in(&mut t);
    // no need to mark bottom
}
```

Interprocedural Analysis

```
struct T {
    r: RRef<i32>,
}

fn move_in(t: &mut T) {
    let r = RRef::new(1);
    t.r = r;
    // mark bottom here?
}
```

```
fn within_rref(mut t: RRef<T>) {
    move_in(&mut *t);
    // must mark bottom
    mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
    move_in(&mut t);
    // no need to mark bottom
}

fn transitive(t: &mut T) {
    move_in(t);
    // ambiguity propagates to caller
}
```

Interprocedural Analysis

```rust
struct T {
   r: RRef<i32>,
}

fn move_in(t: &mut T) {
   let r = RRef::new(1);
   t.r = r;
   // mark bottom here?
}
```

RRef moved into
t @ t.r

```rust
fn within_rref(mut t: RRef<T>) {
   move_in(&mut *t);
   // must mark bottom
   mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
   move_in(&mut t);
   // no need to mark bottom
}

fn transitive(t: &mut T) {
   move_in(t);
   // ambiguity propagates to caller
}
```

Interprocedural Analysis

```
struct T {
   r: RRef<i32>,
}

fn move_in(t: &mut T) {
   let r = RRef::new(1);
   t.r = r;
   // mark bottom here?
}
```

RRef moved into
t @ t.r

```
fn within_rref(mut t: RRef<T>) {
   move_in(&mut *t);
   // must mark bottom
   mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
   move_in(&mut t);
   // no need to mark bottom
}

fn transitive(t: &mut T) {
   move_in(t);
   // ambiguity propagates to caller
}
```

Interprocedural Analysis

```
struct T {
   r: RRef<i32>,
}

fn move_in(t: &mut T) {
   let r = RRef::new(1);
   t.r = r;
   // mark bottom here?
}
```

RRef moved into
t @ t.r

```
fn within_rref(mut t: RRef<T>) {
   move_in(&mut *t);
   // must mark bottom
   mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
   move_in(&mut t);
   // no need to mark bottom
}

fn transitive(t: &mut T) {
   move_in(t);
   // ambiguity propagates to caller
}
```

Interprocedural Analysis

```
struct T {
   r: RRef<i32>,
}

fn move_in(t: &mut T) {
   let r = RRef::new(1);
   t.r = r;
   // mark bottom here?
}
```

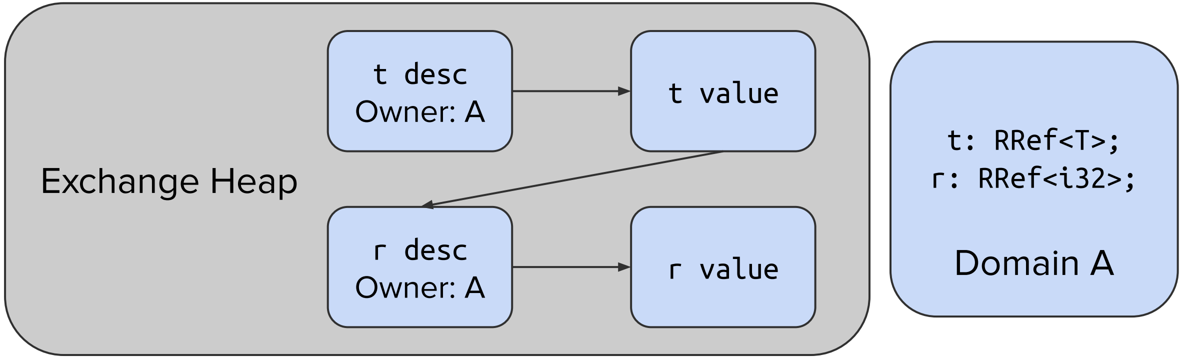
RRef moved into
t @ t.r

```
fn within_rref(mut t: RRef<T>) {
   move_in(&mut *t);
   // must mark bottom
   mark_bottom(&mut t.r);
}

fn outside_rref(mut t: T) {
   move_in(&mut t);
   // no need to mark bottom
}

fn transitive(t: &mut T) {
   move_in(t);
   // ambiguity propagates to caller
}
```

Crash Cleanup Safety

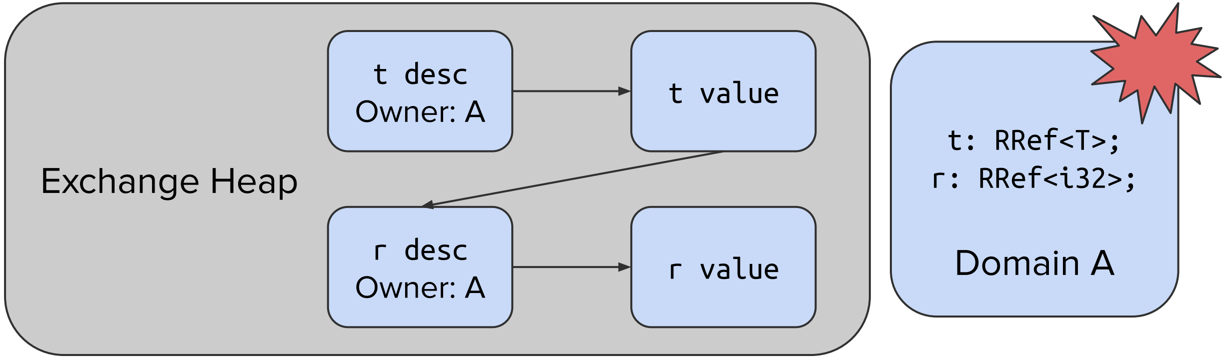
```
t.r = r;
mark_bottom(&mut t.r);
```

Crash Cleanup Safety



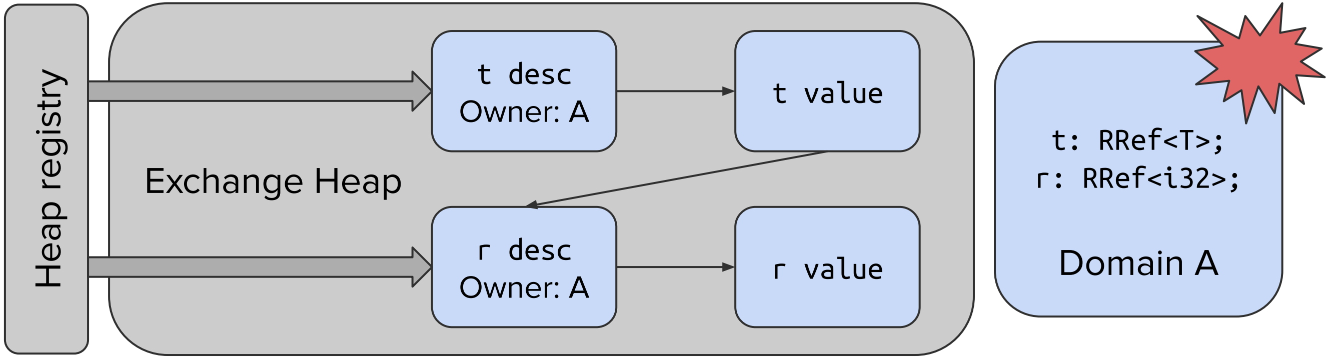

```
t.r = r;
mark_bottom(&mut t.r);
```

Crash Cleanup Safety



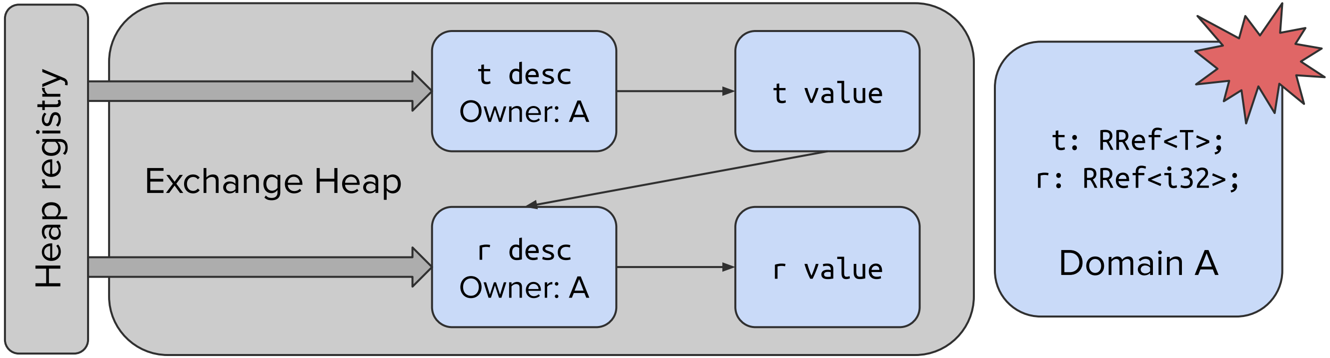

```
t.r = r;
mark_bottom(&mut t.r);
```

Crash Cleanup Safety




```
t.r = r;
mark_bottom(&mut t.r);
```

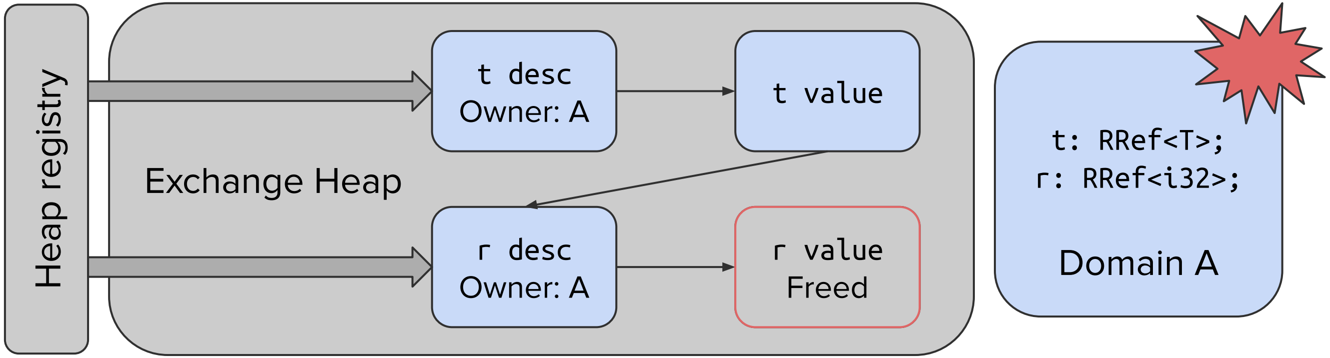
Crash Cleanup Safety




```
t.r = r;
mark_bottom(&mut t.r);
```

Two pass cleanup

Crash Cleanup Safety




```
t.r = r;
mark_bottom(&mut t.r);
```

Two pass cleanup

Crash Cleanup Safety: Selective Drop

```rust
impl<T> Drop for RRefDesc<T> {
    fn drop(&mut self) {
        if !crashed() || self.is_bottom() {
            self.reclaim_and_drop_value();
        }
    }
}
```

Crash Cleanup Safety: Selective Drop

```
impl<T> Drop for RRefDesc<T> {
    fn drop(&mut self) {
        if !crashed() || self.is_bottom() {
            self.reclaim_and_drop_value();
        }
    }
}
```

`!crashed()`

False

Crash Cleanup Safety: Selective Drop

```
impl<T> Drop for RRefDesc<T> {
    fn drop(&mut self) {
        if !crashed() || self.is_bottom() {
            self.reclaim_and_drop_value();
        }
    }
}
```

`!crashed()`

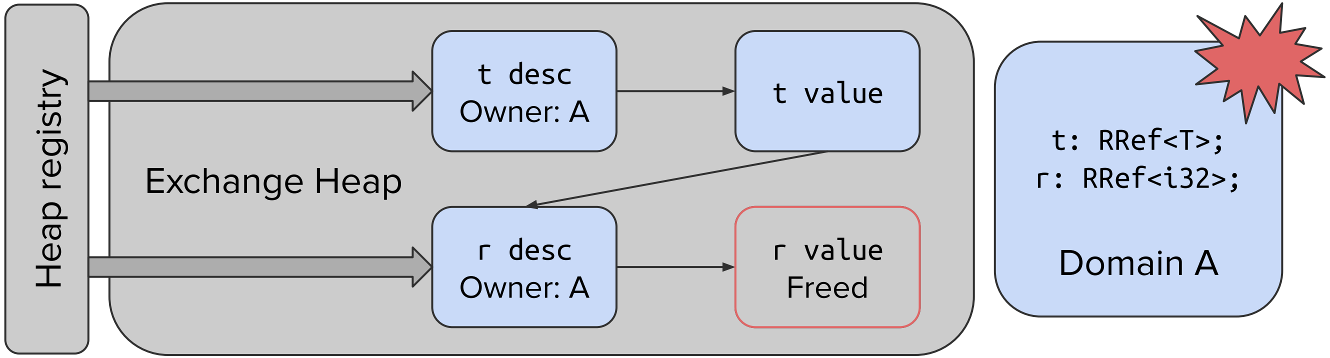
False

`self.is_bottom()`

False

No double free

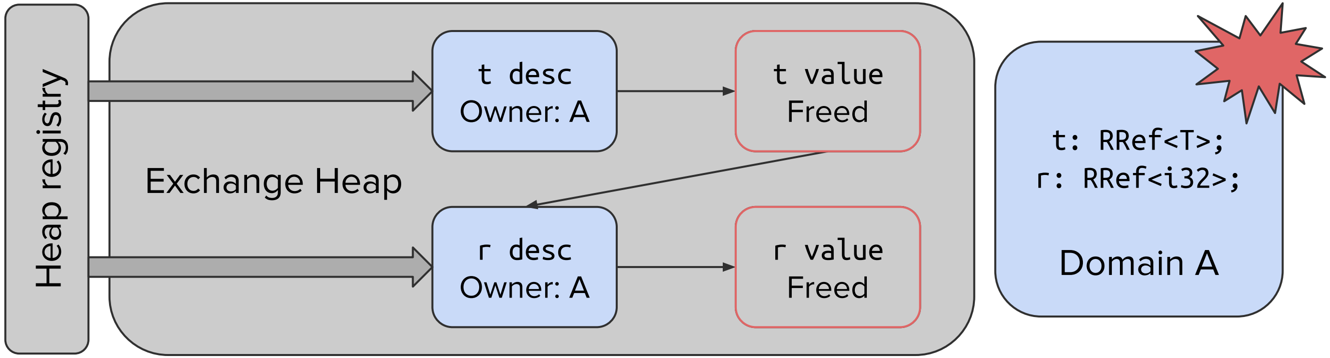
Crash Cleanup Safety



Two pass cleanup

Selective drop

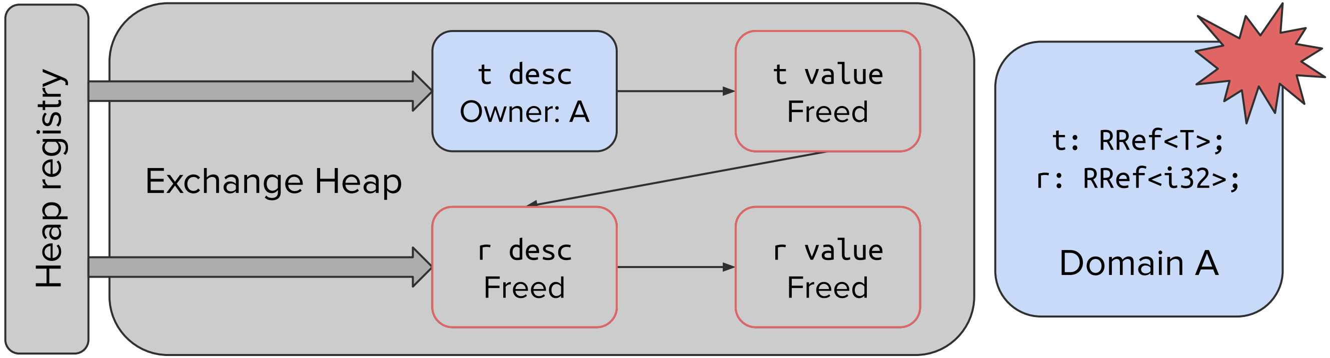
Crash Cleanup Safety



Two pass cleanup

Selective drop

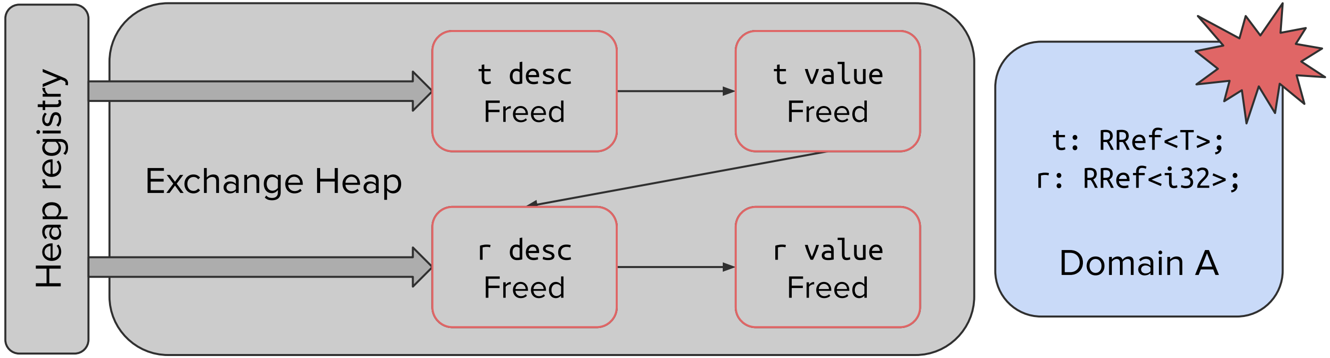
Crash Cleanup Safety



Two pass cleanup

Selective drop

Crash Cleanup Safety

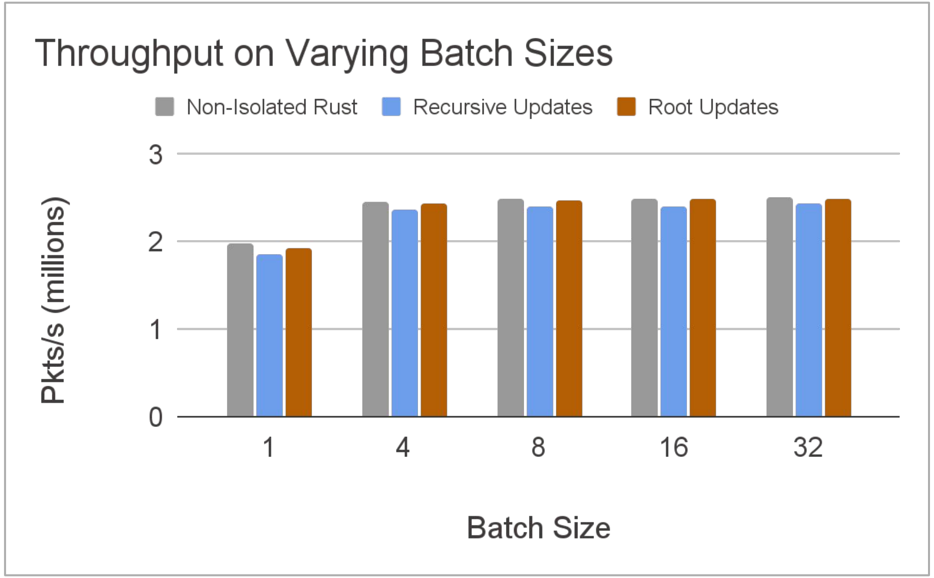


Two pass cleanup

Selective drop

Evaluation: Network Functions

- Relative to normal Rust, compare:
  - Linear updates (naive)
  - Root updates (our analysis)
- Naive proxy: **4%** overhead
- Our analysis: **1%** overhead



Conclusion

Zero copy communication in language-based isolation

enabled by

Static analysis to ensure correct nested data ownership

Thank You!