RustへのFractional Ownershipの動的検査の導入

東北大学 大学院 情報科学研究科
住井・松田研究室
C2IM1034
馬場風汰
指導教員：住井英二郎 教授

Rust [2015]

- メモリ安全なプログラミング言語
- Ownershipという概念に基づき安全な静的メモリ管理を行う
- 基本的に実行時GCを行わない
- 安全で高性能・低水準なメモリ管理が可能
  - 例: FirefoxのCSSエンジン、Dropboxの圧縮ルーチン等

Ownershipによるメモリ管理

- 各オブジェクトに唯一のowner（変数など）が静的に決められている
- オブジェクトのメモリ領域は、ownerのスコープが終了等すると解放
- オブジェクトの読み書きや解放にはownershipが必要

```
fn main() {
  let s = String::from("Hello"); // sがStringオブジェクトのownerとなる
  println!("{}", s);

  // sの指すStringオブジェクトが解放
}
```

OwnershipのMove

- Ownershipは、変数への代入や関数に引数を渡す際に移動 (move)

```
fn main() {
  let s = String::from("Hello");
  let new_s = s; // Stringオブジェクトのownershipがsからnew_sに移動
  println!("{}", s); // sはownerでないためコンパイルエラー

  // new_sの指すStringオブジェクトが解放
}
```

OwnershipのMove

- Ownershipは、変数への代入や関数に引数を渡す際に移動 (move)

```
fn f(x: String) { // Stringオブジェクトのownershipを受け取る
  println!("{}", x);
  // xの指すStringオブジェクトが解放
}
fn main() {
  let s = String::from("Hello");
  f(s); // ownershipがfの引数に移動
  println("{}",s); // sはもはやownerでないためコンパイルエラー
                   // 実際、sの指すStringオブジェクトは解放済
}
```

OwnershipのBorrowing

- Ownershipは「参照」を通して「借りる」ことができる (borrowing)

```
fn f(x: &String) {
  println!("{}", x);
}
fn main() {
  let s = String::from("Hello");

  f(&s); // ownershipが一時的に貸し出される
  // ownershipが自動的に返却される

  println!("{}", s); // sはownerなのでエラーは発生しない
}
```

Borrowing (参照)のLifetime

- ライフタイム ＝ 参照を用いることができる範囲
  - 基本的に自動推論されるが、明示的に注釈をつけることもできる

```
fn f<'a>(x: &'a String) { // xのライフタイムは関数fの本体と一致
  println!("{}", x);
}
fn main() {
  let s = String::from("Hello");

  f(&s); // ownershipが一時的に貸し出される
  // ownershipが自動的に返却される

  println!("{}", s); // sはownerなのでエラーは発生しない
}
```

参照カウントオブジェクト（Rc）

- オブジェクトへの参照と参照カウントを持つコンテナ
  - 内部でunsafeを利用(静的なownershipを無視)
- 静的なownershipでは柔軟性に欠ける場合に利用

コード例

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
```

コード例

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let h = Rc::new(String::from("Hello"));
  let w = Rc::new(String::from("World")); // 2つのStringオブジェクトを準備
```

コード例

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let h = Rc::new(String::from("Hello"));
  let w = Rc::new(String::from("World")); // 2つのStringオブジェクトを準備
  let mut rng = rand::thread_rng(); // 乱数生成器を準備
```

コード例

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let h = Rc::new(String::from("Hello"));
  let w = Rc::new(String::from("World")); // 2つのStringオブジェクトを準備
  let mut rng = rand::thread_rng(); // 乱数生成器を準備
  for _ in 0..10 { // Stringオブジェクトをランダムに挿入
    if rng.gen() { // ランダムにtrueかfalseを返す
      vec.push(h.clone()); // 参照hを複製して挿入
    }
    else {
      vec.push(w.clone()); // 参照wを複製して挿入
    }
  }
}
```

Rcの問題点 (1/4)

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let h = Rc::new(String::from("Hello"));
  let w = Rc::new(String::from("World")); // 2つのStringオブジェクトを準備
  let mut rng = rand::thread_rng(); // 乱数生成器を準備
  for _ in 0..10 { // Stringオブジェクトをランダムに挿入
    if rng.gen() { // ランダムにtrueかfalseを返す
      vec.push(h.clone()); // 参照hを複製して挿入
    }
    else {
      vec.push(w.clone()); // 参照wを複製して挿入
    }
  }

  remove_string(&mut vec, String::from("Hello")); // 配列から値"Hello"を削除
  … // vecを用いる処理（hやw自体は用いない）→ hは不要なのに残っている！
}
```

Rcの問題点 (2/4)

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let w = Rc::new(String::from("World"));
  {
    let h = Rc::new(String::from("Hello")); // 2つのStringオブジェクトを準備
    let mut rng = rand::thread_rng(); // 乱数生成器を準備
    for _ in 0..10 { // Stringオブジェクトをランダムに挿入
      if rng.gen() { // ランダムにtrueかfalseを返す
        vec.push(h.clone()); // 参照hを複製して挿入
      }
      else {
        vec.push(w_clone()); // 参照wを複製して挿入
      }
    }
    remove_string(&mut vec, String::from("Hello")); // 配列から値"Hello"を削除
  } // hが解放 → hの指していたStringオブジェクトも参照カウントが0になり解放
  … // vecを用いる処理（hやw自体は用いない）
}
```

Rcの問題点 (3/4)

```
fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new(); // Vectorを準備
  let w = Rc::new(String::from("World"));
  {
    let h = Rc::new(String::from("Hello")); // 2つのStringオブジェクトを準備
    let mut rng = rand::thread_rng(); // 乱数生成器を準備
    for _ in 0..10 { // Stringオブジェクトをランダムに挿入
      if rng.gen() { // ランダムにtrueかfalseを返す
        vec.push(h.clone()); // 参照hを複製して挿入
      }
      else {
        vec.push(w_clone()); // 参照wを複製して挿入
      }
    }
    remove_string(&mut vec, String::from("Hell")); // スペルミス
  } // h自体は解放されるがStringオブジェクトは解放されない
  … // vecを用いる処理（hやw自体は用いない）
}
```

Rcの問題点 (4/4)

意図せず参照の複製が残り、
オブジェクトが解放されない可能性がある

しかも静的にも動的にも検出されない

我々の提案

- Fractional Ownership [Boyland 03] の考え方を応用し、意図しない複製が残っていたら(動的に)検出

  - ユーザが解放のタイミングを指定(することを強制)

目次

- 序論
- Fractional Ownership [Boyland 03]
- 本研究で提案する参照オブジェクトの概要
- 使用例
- 実装の詳細

Fractional Ownership [Boyland 03]

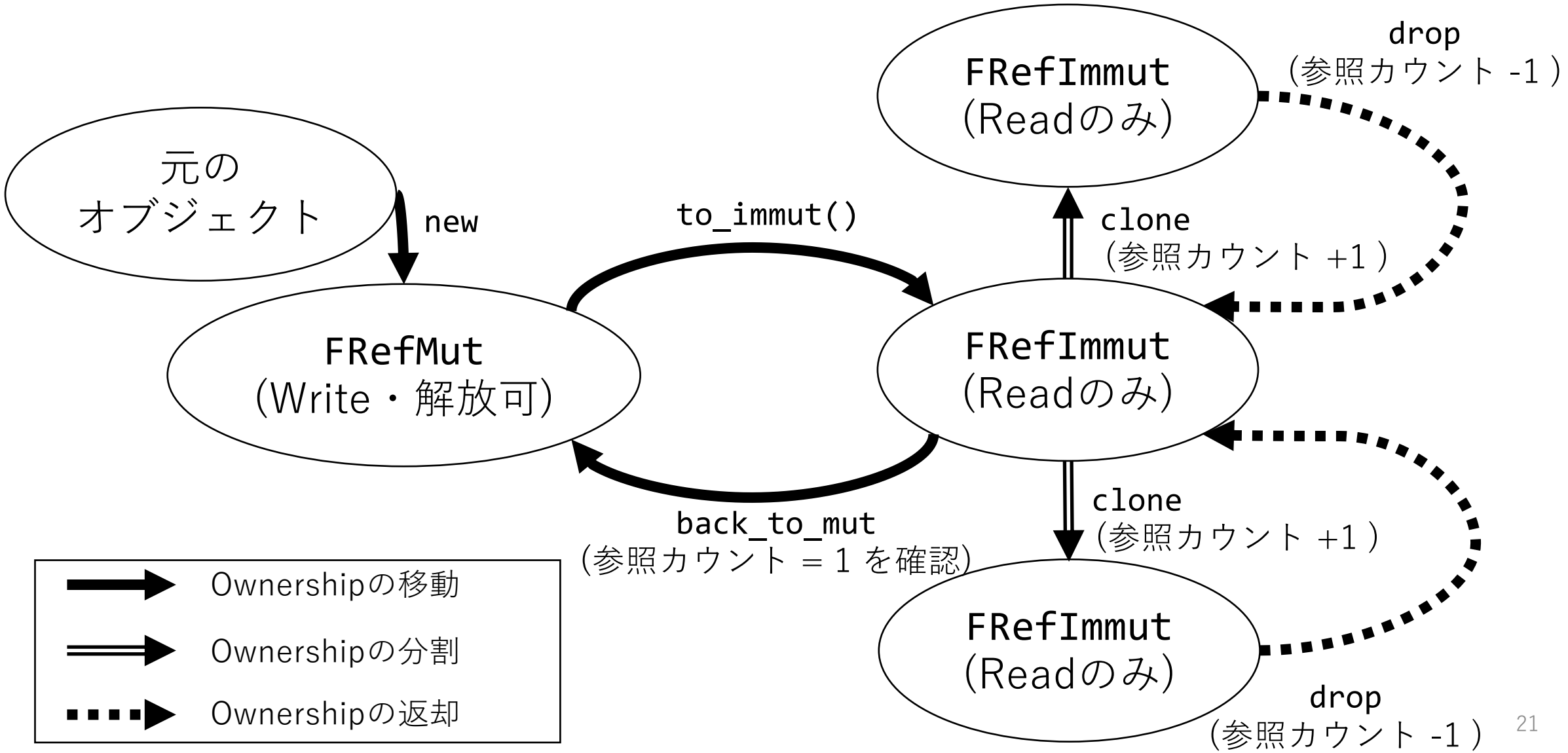
- 0以上1以下の有理数
  - オブジェクト生成時に1を与え、参照複製時に分割し、複製が消滅する際に集約する
  - 0より大きく1以下ならばread可
  - 1ならばwrite可、かつ解放可 (0になる)
  - 逆に、0より大きければ、いずれ解放しなければならない

- 本来は有理数計画法を用いて自動で静的に検査
  - 本研究では動的検査に応用

目次

- 序論
- Fractional Ownership [Boyland 03]
- 本研究で提案する参照オブジェクトの概要
- 使用例
- 実装の詳細

新たな参照オブジェクトの概要



目次

- 序論
- Fractional Ownership [Boyland 03]
- 本研究で提案する参照オブジェクトの概要
- 使用例
- 実装の詳細

本研究の参照オブジェクトの使用例

```
fn main() {
  let w = FRefMut::new(String::from("World")).to_immut(); // Ownership 1の参照の生成
```

本研究の参照オブジェクトの使用例

```
fn main() {
  let w = FRefMut::new(String::from("World")).to_immut(); // Ownership 1の参照の生成
  {
    let mut vec: Vec<FRefImmut<String>> = Vec::new(); // Vectorを準備
```

本研究の参照オブジェクトの使用例

```
fn main() {
  let w = FRefMut::new(String::from("World")).to_immut(); // Ownership 1の参照の生成
  {
    let mut vec: Vec<FRefImmut<String>> = Vec::new(); // Vectorを準備
    let mut rng = rand::thread_rng();
    {
      let h = FRefMut::new(String::from("Hello")).to_immut(); // Ownership 1の参照の生成
```

本研究の参照オブジェクトの使用例

```
fn main() {
  let w = FRefMut::new(String::from("World")).to_immut(); // Ownership 1の参照の生成
  {
    let mut vec: Vec<FRefImmut<String>> = Vec::new(); // Vectorを準備
    let mut rng = rand::thread_rng();
    {
      let h = FRefMut::new(String::from("Hello")).to_immut(); // Ownership 1の参照の生成
      for _ in 0..10 {
        if rng.gen() {
          vec.push(h.clone_immut()); // 参照の複製
        } else {
          vec.push(w.clone_immut()); // 参照の複製
        }
      }
      remove_string(&mut vec, String::from("Hello"));
```

本研究の参照オブジェクトの使用例

```
fn main() {
  let w = FRefMut::new(String::from("World")).to_immut(); // Ownership 1の参照の生成
  {
    let mut vec: Vec<FRefImmut<String>> = Vec::new(); // Vectorを準備
    let mut rng = rand::thread_rng();
    {
      let h = FRefMut::new(String::from("Hello")).to_immut(); // Ownership 1の参照の生成
      for _ in 0..10 {
        if rng.gen() {
          vec.push(h.clone_immut()); // 参照の複製
        } else {
          vec.push(w.clone_immut()); // 参照の複製
        }
      }
      remove_string(&mut vec, String::from("Hello"));
      h.back_to_mut(); // Ownershipが1であることを確認してからdrop
    }
    … // vecを用いる処理（hやw自体は用いない）
  }
  w.back_to_mut(); // Ownershipが1であることを確認してからdrop
}
```

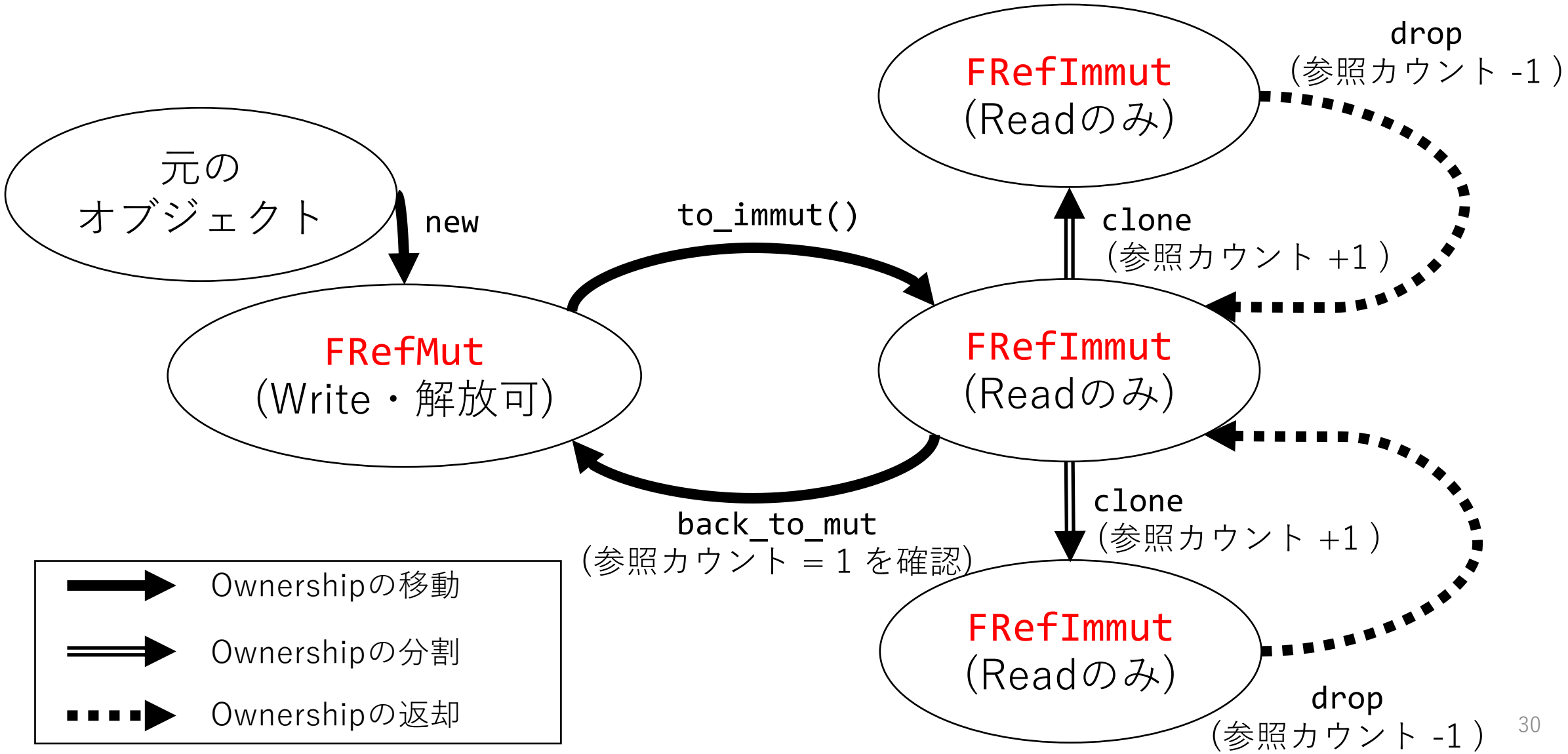
解放忘れを検出する例

```
fn main() {
  let mut w = FRefMut::new(String::from("World"));
  let w_immut = w.to_immut();
  {
    let mut vec: Vec<FRefImmut<String>> = Vec::new();
    let mut h = FRefMut::new(String::from("Hello"));
    let mut rng = rand::thread_rng();
    {
      let h_immut = h.to_immut();
      for _ in 0..10 {
        if rng.gen() {
          let h_immut1 = h_immut.clone_immut();
          vec.push(h_immut1);
        }
        else {
          let w_immut1 = w_immut.clone_immut();
          vec.push(w_immut1);
        }
      }
      remove_string(&mut vec, String::from("Hell")); // ユーザによるスペルミス
      let mut h_mut = h_immut.back_to_mut();  // 参照の複製が残っており、ownershipが1になっていないので実行時エラー
    }
    // hとwは用いず、vecを用いる処理
  }
  let mut w_mut = w_immut.back_to_mut();
}
```

目次

- 序論
- Fractional Ownership [Boyland 03]
- 本研究で提案する参照オブジェクトの概要
- 使用例
- 実装の詳細

新たな参照オブジェクトの概要

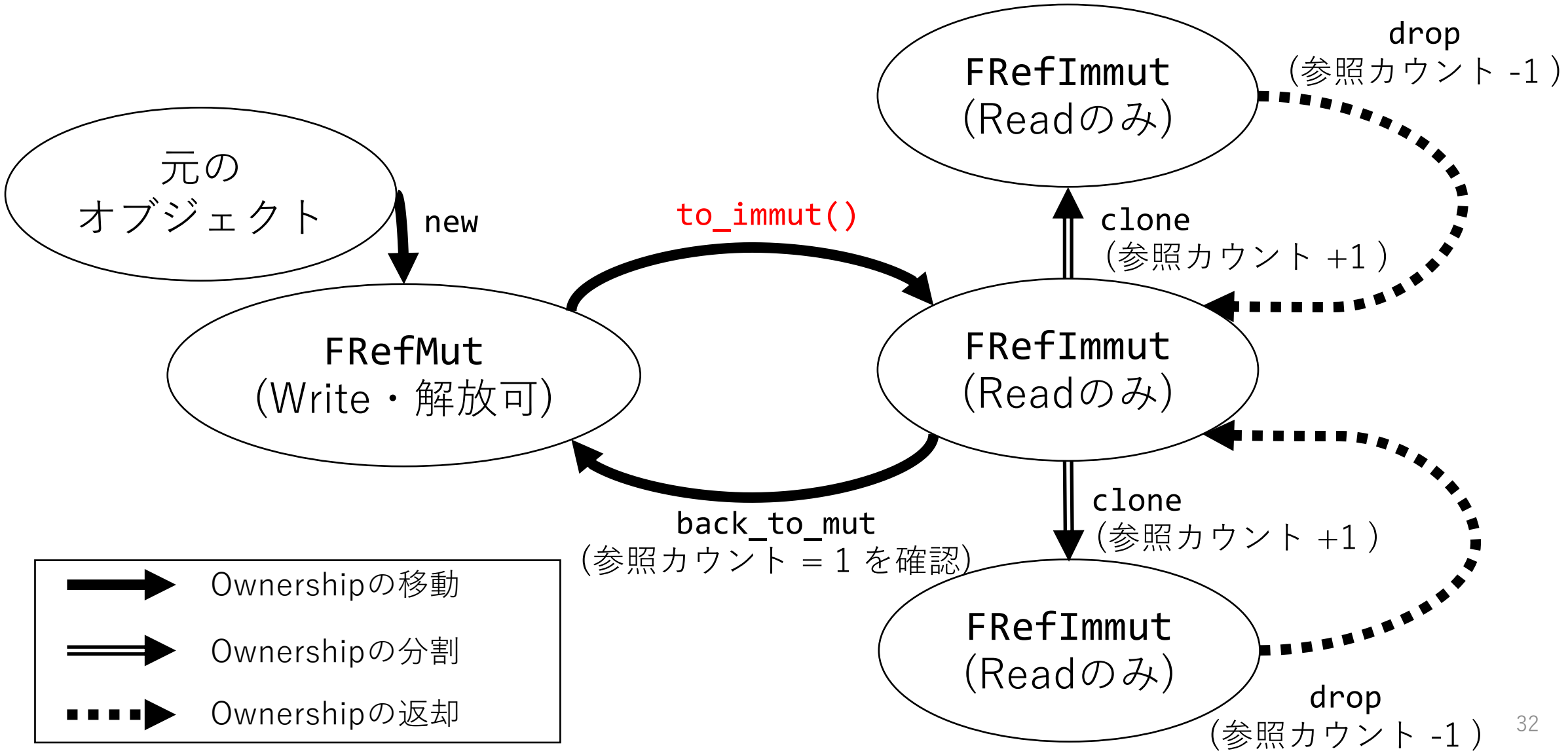


実装

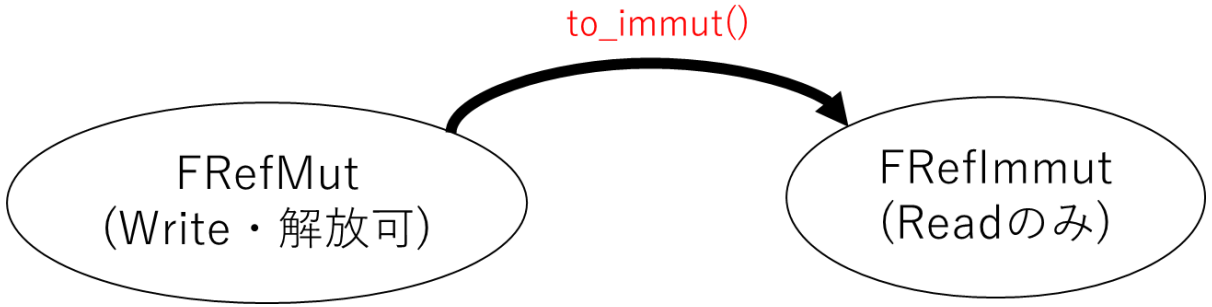
- FRefMutとFRefImmutな参照

```
pub struct FRefMut<T> {
  data: T // 中身のオブジェクト（RustのOwnershipに従う）
}
pub struct FRefImmut<T> {
  ptr: NonNull<FRefInner<T>> // Rustのownershipを無視する特別(unsafe)な参照
  phantom: PhantomData<FRefInner<T>>
}
struct FRefInner<T> {
  ref_count: atomic::AtomicUsize, // Immutableな参照の数
                                  // Ownershipはref_count分の1
  data: T // 中身のオブジェクト
}
```

新たな参照オブジェクトの概要



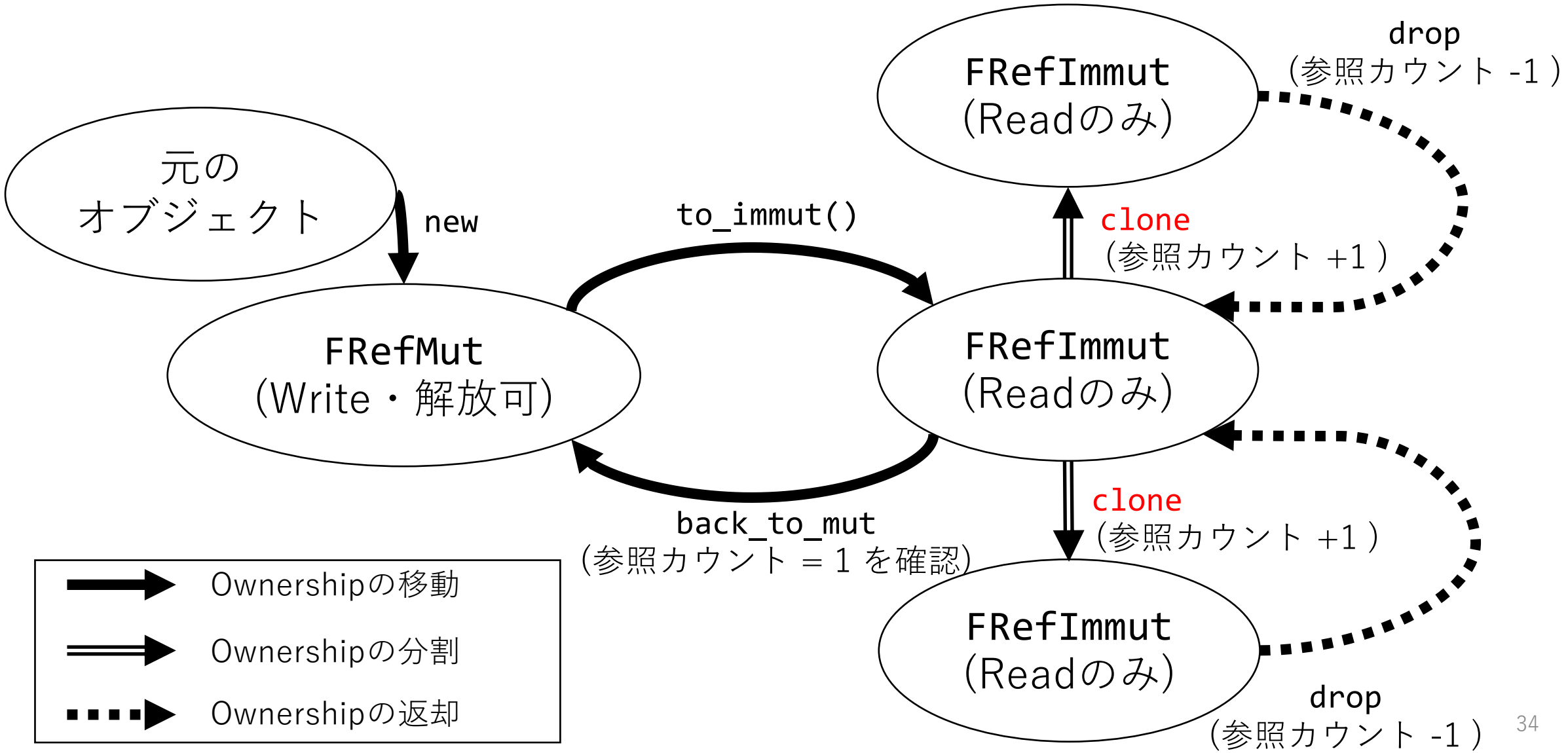
実装



- FRefMutからFRefImmutな参照に変換する関数to_immut
  - Rcのnewに類似

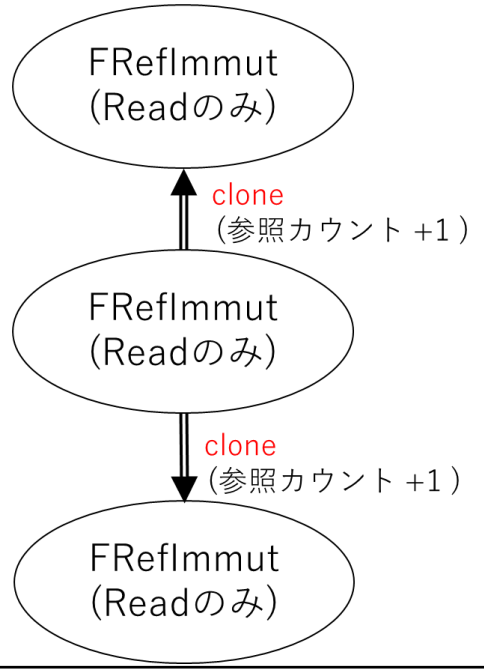
```
pub fn to_immut(self: FRefMut<T>) -> FRefImmut<T> { // selfがmove
  let mut this = ManuallyDrop::new(self);
  let inner = unsafe{ptr::read(&mut this.data)}; // 中身のオブジェクトを取り出す

  let x = Box::new(FRefInner { // FRefInnerを作成
    ref_count: AtomicUsize::new(1),
    data: inner,
  });
  FRefImmut {ptr: Box::leak(x).into(), // xはdropしないで生ポインタに変換
             phantom:PhantomData}
}
```

新たな参照オブジェクトの概要



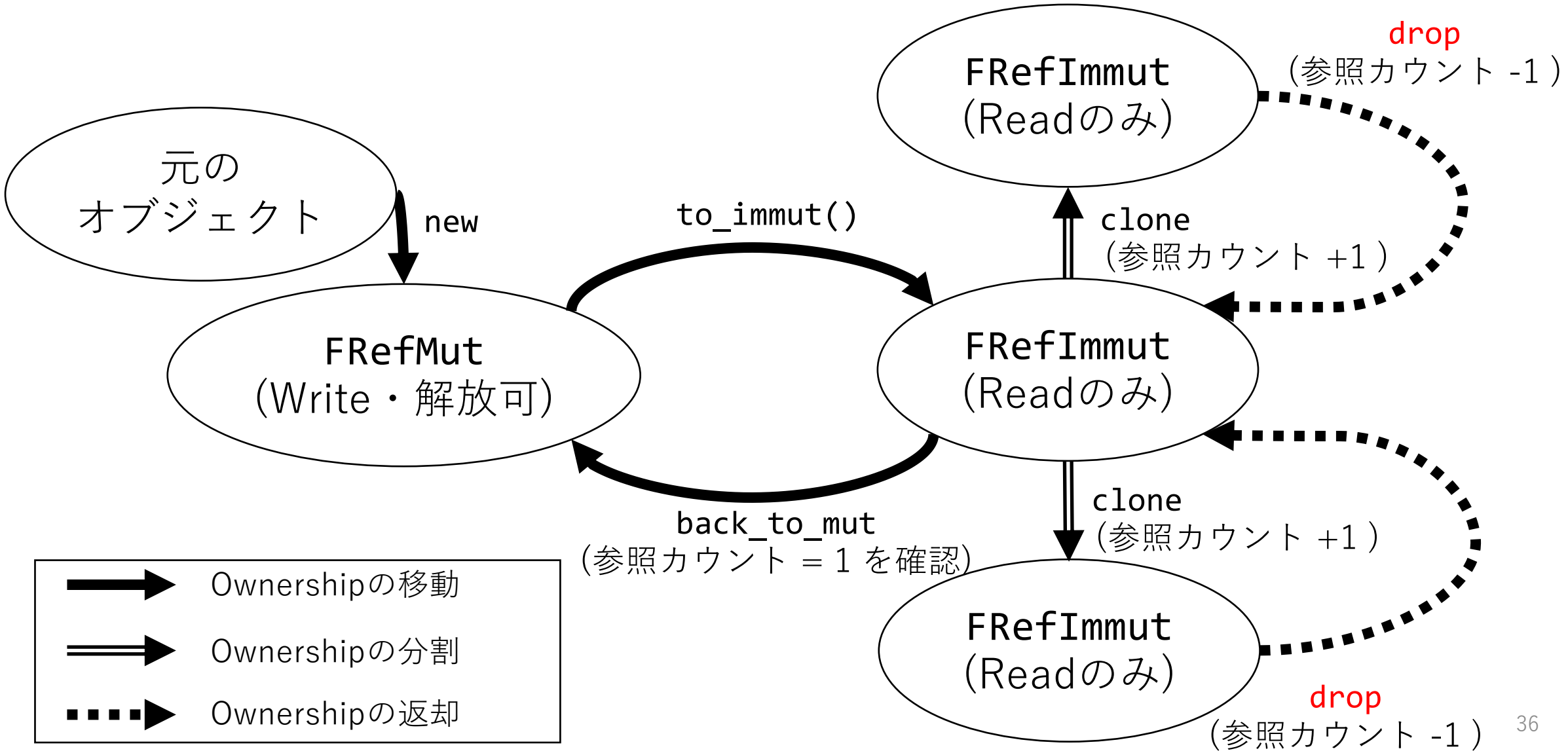
実装

- FRefImmutのownershipを分割する関数clone_immut
  - Rcのcloneに類似




```
pub fn clone_immut(&self: &FRefImmut<T>)->FRefImmut<T> { // moveはしない
  let old_size = self.inner().ref_count.fetch_add(1, Release); // カウント増加
  if old_size > MAX_REFCOUNT {
    abort();
  }
  FRefImmut { ptr: self.ptr,  // Rustのownershipは無視して生ポインタを複製
              phantom: PhantomData }
}
```

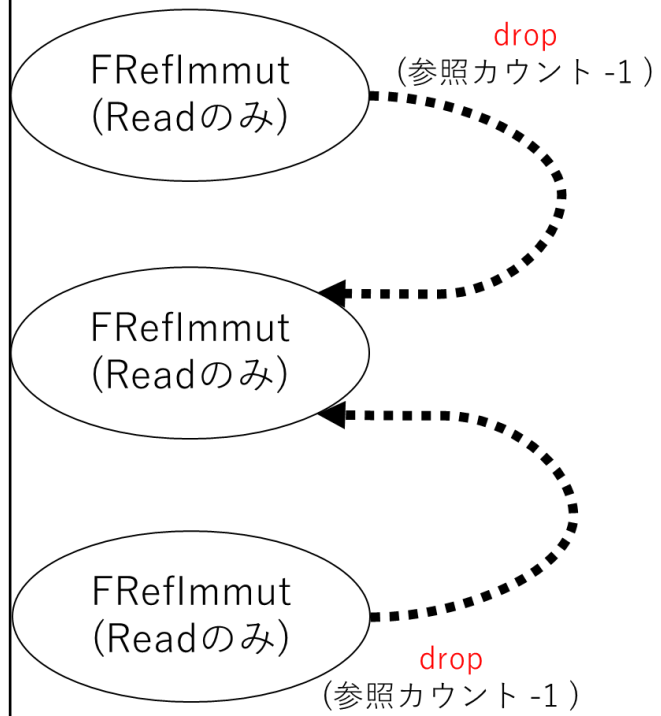
新たな参照オブジェクトの概要



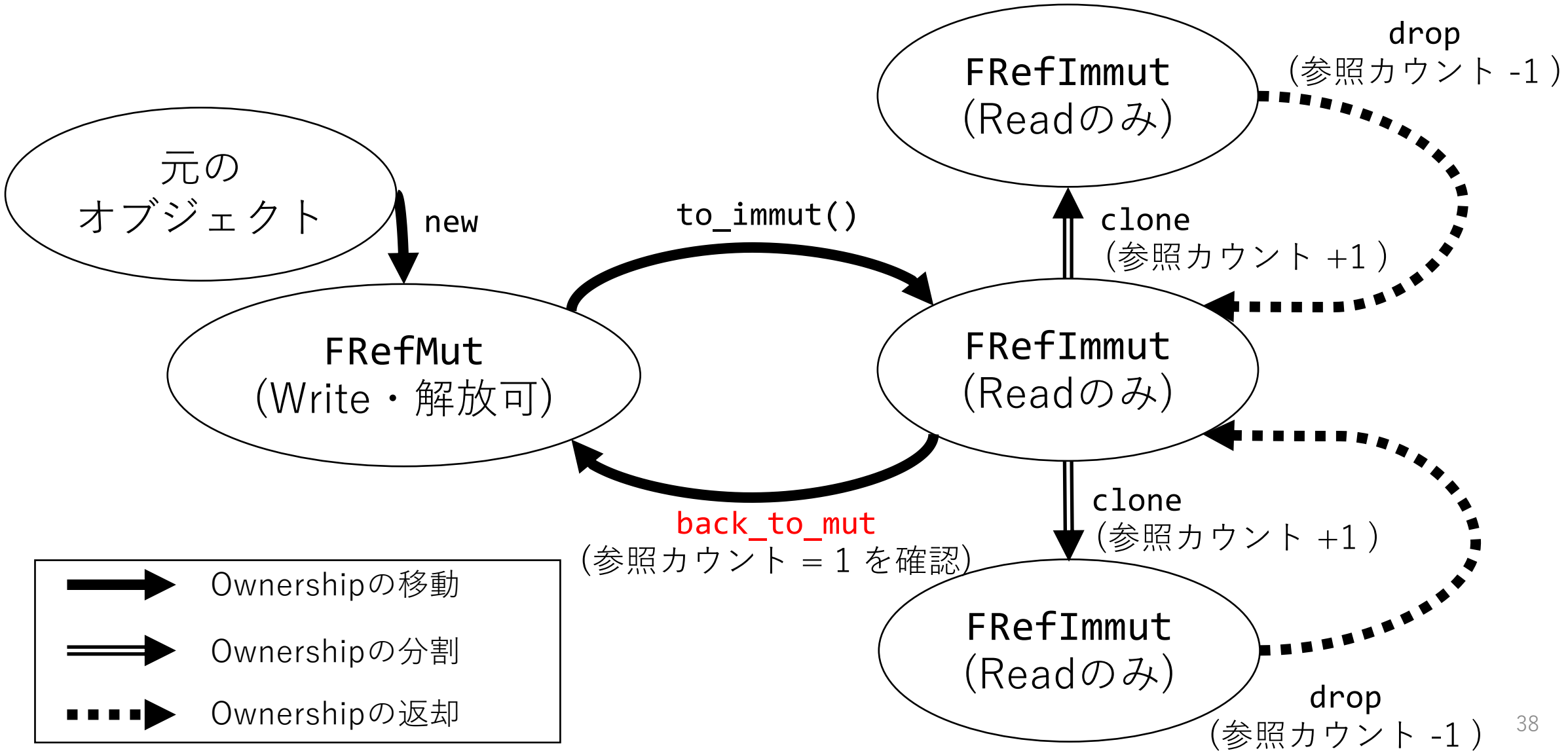
実装

- FRefImmutのdrop

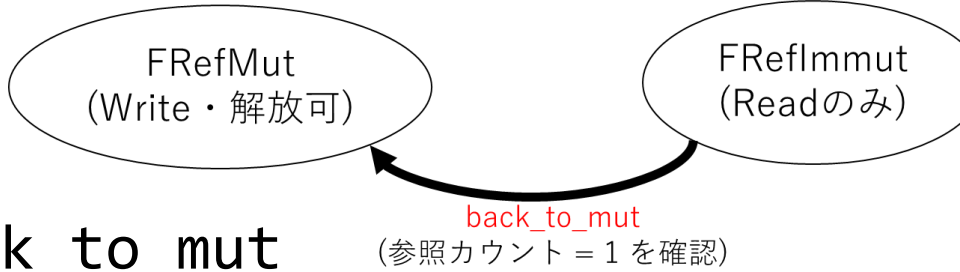
```
fn drop(&mut self) {
  let count =
    self.inner().ref_count.fetch_sub(1, Relaxed); // カウント減少
  atomic::fence(Acquire);
  if count > 1 {
    return
  } else if count == 1 { // back_to_mutせずカウントが0に
    unsafe { ptr::drop_in_place(self.ptr.as_ptr()) };
    if !panicking() {
      panic!("memory leak"); // 解放忘れとみなして実行時エラー
    }
  } else {
    abort();
  }
}
```



新たな参照オブジェクトの概要



実装



- FRefImmutからFRefMutな参照に変換する関数back_to_mut
  - Rcを通常のオブジェクトに戻すときの処理（into_inner）に類似

```
pub fn back_to_mut(self) -> FRefMut<T> { // moveする
  if self.inner().ref_count.load(Acquire) != 1{
    panic!("cannot back to mut"); // 参照カウントが1でなければ実行時エラー
  }
  let mut this = ManuallyDrop::new(self);
  let inner =
    unsafe { ptr::read(&mut (*this.ptr.as_ptr()).data)};
  unsafe {
    dealloc(this.ptr.cast().as_mut(),
            Layout::for_value_raw(this.ptr.as_ptr()))
  }
  FRefMut { data: inner }
}
```

関連研究：Fractional Ownership

[Boyland 03]

- Fractional ownershipの提案
  - 有理数計画法を用いて静的に検査

[Suenaga+ 09]

- C言語プログラムのメモリ関連のエラーをfractional ownershipにより検証

[Nakayama+ 23]

- Fractional ownershipとRustのownershipを組み合わせた
  新たな所有権型システムの提案

関連研究：Rustの安全性の理論

[Jung+ 17]

- Rustで**unsafe**を用いたライブラリ（Rcなど）のメモリ安全性を検証

[Weiss+ 19]

- Rustのコア言語の型システム（borrow checkerを含む）を形式化

関連研究：並列処理

[Courtois+ 71]

- Readers-Writer Lockにより競合状態を防止

- 本研究では、ロックではなくFRefMutとFRefImmutの区別により競合状態を防止できる（論文参照）
  - Fractional ownership [Boyland 03]の本来の目的

まとめと今後の研究課題

- Fractional ownershipの考え方を応用し、オブジェクトの解放し忘れを実行時に検出する参照オブジェクトを提案
- 並列処理にも応用可能
  - 詳しくは論文を参照

今後の研究課題：

- メモリ安全性の形式的証明(cf. [Jung+ 17])
- 実際の例で評価
- Weak referenceの導入

Rustでの並列処理 1/3

- Rustの通常のスレッド生成では、ownerは高々一つのスレッドにmoveする

```
fn main() {
    let s = String::from("Hello");
    thread::spawn(move || {
        println!("{}", s);  // ownerは生成されたスレッドにmoveする
    });
}
```

Rustでの並列処理 2/3

- Rustの通常のスレッド生成では、ownerは高々一つのスレッドにmoveする

```
fn main() {
    let s = String::from("Hello");
    thread::spawn(move || {
        println!("{}", s); // ownerは生成されたスレッドにmoveする
    });
    thread::spawn(move || {
        println!("{}", s); // ownershipがないのでエラー
    });
}
```

Rustでの並列処理 3/3

- スレッドのライフタイムは無限とみなされるため、ownershipが返却されない

```
fn main() {
    let s = String::from("Hello");
    let t1 = thread::spawn(|| {
        println!("{}", &s);  // スレッドをまたいだborrowingはエラー
    });
    let t2 = thread::spawn(|| {
        println!("{}", &s);
    }
    t1.join();
    t2.join();
}
```

スコープ付きスレッド

- スレッドに静的スコープを付与
- 通常のスレッドと異なり、共有オブジェクトに対するborrowingが可能
- 静的スコープに従ってスレッドの合流が起きる

```
fn main () {
    let mut s = String::from("Hello");
    thread::scope(|sc| {
        sc.spawn(|| {
            println!("{}", &s);
        });
        sc.spawn(|| {
            println!("{}", &s);
        });
    }); // いずれのスレッドも合流する
    s.push_str("World");
}
```

スコープ付きスレッドで表せない例

- threadの合流が静的スコープに従わない場合

```
fn my_spawn (a: &String) -> JoinHandle<()>{
    let t = thread::spawn(|| {
        println!("{}", &a);
    });
    return t
}
fn main() {
    let mut a = String::from("Hello");
    let t1 = my_spawn(&a);
    let t2 = my_spawn(&a);
    t1.join();
    t2.join(); // この時点でownershipは返却済みであるはず
    a.push_str(“World”); // Rustではエラー
}
```

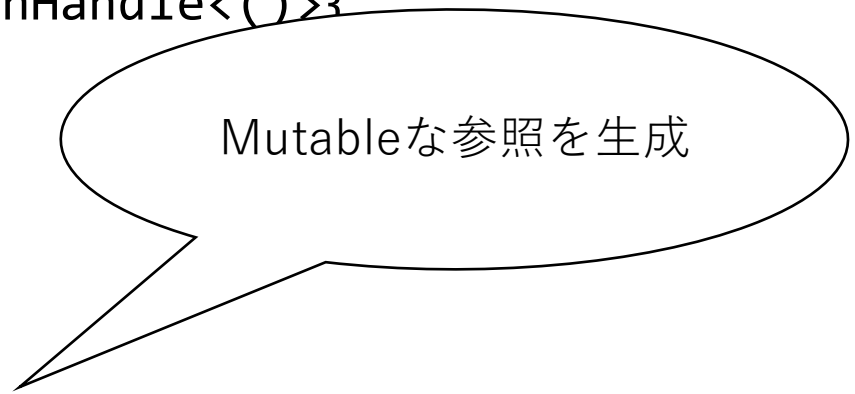
Rustでの共有オブジェクト処理の問題点のまとめ

- 共有オブジェクトはスレッドをまたいでborrowingできない

- スコープ付きスレッドは提供されているが、スレッドの合流が静的スコープに従う場合にしか用いることができない

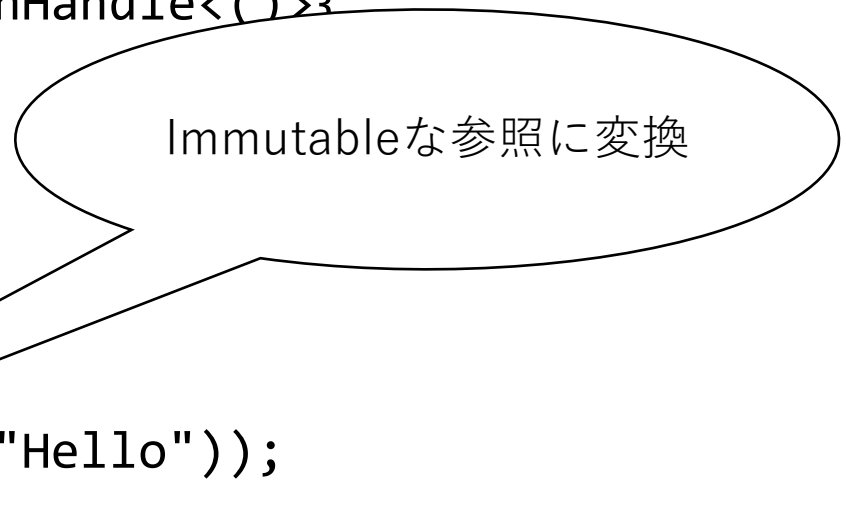
コード例

```
fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
    let t = thread::spawn(||{
        println!("{}", a);
    });
    return t
}
fn main() {
    let mut s = RefMut::new(String::from("Hello"));
    let s1 = s.to_immut();
    let s2 = s1.clone_immut();
    let t1 = my_spawn(&s1);
    let t2 = my_spawn(&s2);
    t1.join();
    t2.join();
    drop(s2); // s2を明示的に解放
    let mut s3 = s1.back_to_mut();
    s3.push_str("World");
}
```



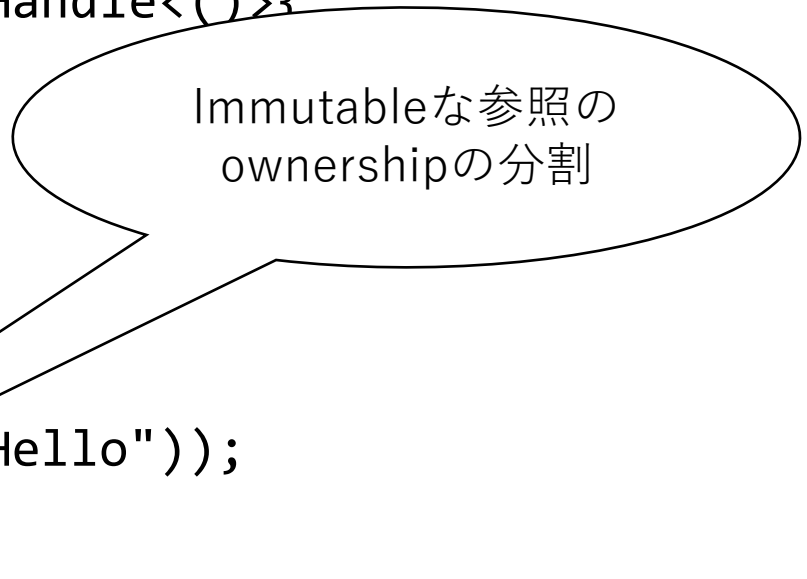
コード例

```
fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
    let t = thread::spawn(||{
        println!("{}", a);
    });
    return t
}
fn main() {
    let mut s = RefMut::new(String::from("Hello"));
    let s1 = s.to_immut();
    let s2 = s1.clone_immut();
    let t1 = my_spawn(&s1);
    let t2 = my_spawn(&s2);
    t1.join();
    t2.join();
    drop(s2); // s2を明示的に解放
    let mut s3 = s1.back_to_mut();
    s3.push_str("World");
}
```



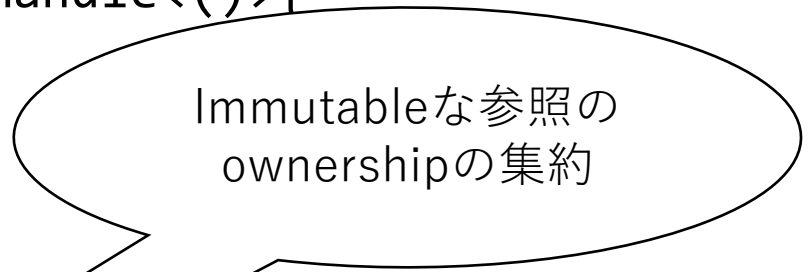
コード例

```
fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
    let t = thread::spawn(||{
        println!("{}", a);
    });
    return t
}
fn main() {
    let mut s = RefMut::new(String::from("Hello"));
    let s1 = s.to_immut();
    let s2 = s1.clone_immut();
    let t1 = my_spawn(&s1);
    let t2 = my_spawn(&s2);
    t1.join();
    t2.join();
    drop(s2); // s2を明示的に解放
    let mut s3 = s1.back_to_mut();
    s3.push_str("World");
}
```



コード例

```
fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
    let t = thread::spawn(||{
        println!("{}", a);
    });
    return t
}
fn main() {
    let mut s = RefMut::new(String::f    ("Hello"));
    let s1 = s.to_immut();
    let s2 = s1.clone_immut();
    let t1 = my_spawn(&s1);
    let t2 = my_spawn(&s2);
    t1.join();
    t2.join();
    drop(s2); // s2を明示的に解放
    let mut s3 = s1.back_to_mut();
    s3.push_str("World");
}
```



コード例

```
fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
    let t = thread::spawn(||{
        println!("{}", a);
    });
    return t
}
fn main() {
    let mut s = RefMut::new(String::from("Hel    );
    let s1 = s.to_immut();
    let s2 = s1.clone_immut();
    let t1 = my_spawn(&s1);
    let t2 = my_spawn(&s2);
    t1.join();
    t2.join();
    drop(s2); // s2を明示的に解放
    let mut s3 = s1.back_to_mut();
    s3.push_str("World");
}
```