Reussir

A Memory Reuse IR Playground for FP Languages

Schrodinger Zhu

February 13, 2026

Overview

Introduction

Foundations

> “…This suggests an answer to our original question, at least within the framework of interaction nets: the fundamental laws of computation are commutation and annihilation.”
>
> — Yves Lafont, Interaction Combinators, 1997.

The FP Connection

- **Annihilation** ↔ Elimination (Pattern Matching / Destructors)
- **Commutation** ↔ Introduction (Constructors / Structural rules)

Functional programs are built from Intro/Elim pairs — each pair is a potential memory reuse opportunity. Reussir makes these interactions explicit at the IR level.

Examples

List Reversal

```
enum List<T> { Nil, Cons(T, List<T>) }

fn reverse_impl(list : List<i32>, acc : List<i32>) -> List<i32> {
    match list {
        List::Nil => acc,
        List::Cons(x, xs) => reverse_impl(xs, List::Cons{x, acc})
    }
}

pub fn reverse(list : List<i32>) -> List<i32> {
    reverse_impl(list, List::Nil)
}
```

Cons(x, xs) is eliminated; a new Cons{x, acc} is introduced — the eliminated cell can be reused in-place for the introduction.

RBTree: Balancing (1/2)

```
enum [value] Color { Red, Black }
enum Tree<T : Num> {
    Branch(Color, Tree<T>, T, Tree<T>),
    Leaf
}

fn balance_left<T : Num>(l : Tree<T>, k : T, r : Tree<T>) -> Tree<T> {
    match l {
        Tree::Branch(_, Tree::Branch(Color::Red, lx, kx, rx), ky, ry) =>
            Tree::Branch{
                Color::Red,
                Tree::Branch{Color::Black, lx, kx, rx},
                ky,
                Tree::Branch{Color::Black, ry, k, r}
            },
        // ... (continued)
```

RBTree: Balancing (2/2)

```
        // ... (continued)
        Tree::Branch(_, ly, ky, Tree::Branch(Color::Red, lx, kx, rx)) =>
            Tree::Branch{
                Color::Red,
                Tree::Branch{Color::Black, ly, ky, lx},
                kx,
                Tree::Branch{Color::Black, rx, k, r}
            },

        Tree::Branch(_, lx, kx, rx) =>
            Tree::Branch{
                Color::Black,
                Tree::Branch{Color::Red, lx, kx, rx},
                k,
                r
            },

        Tree::Leaf => Tree::Leaf
    }
}
```

Background

The Cost of Abstraction

FP relies heavily on heap-allocated objects. Memory management is both essential and expensive.

- **Haskell, OCaml**: sophisticated Garbage Collectors (GC).
- Tradeoffs: Latency, pauses, memory overhead.

Evolving Solutions

Languages are evolving to mitigate GC costs:

- **Haskell**: Introduced Linear Types for resource control.
- **OCaml (Jane Street)**: Exploring local/stack allocation strategies.

The RC Alternative

Lean4 and **Koka** adopt a different approach:

- Reference Counting (RC) system.
- Functional-But-In-Place (FBIP):
  - If an object has RC $= 1$ (exclusive ownership), mutate it in place!
  - Enables efficient mutation within functional purity.

Thesis: FBIP has more potential than prior work reveals.

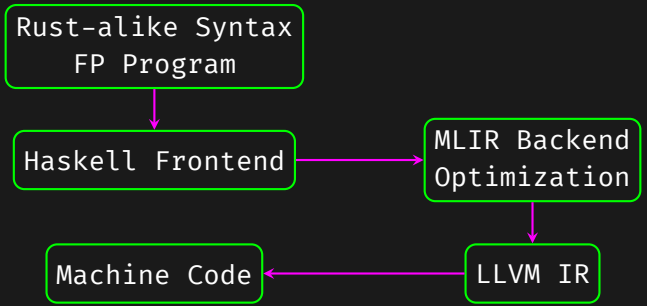
Reussir’s Design

What We Present

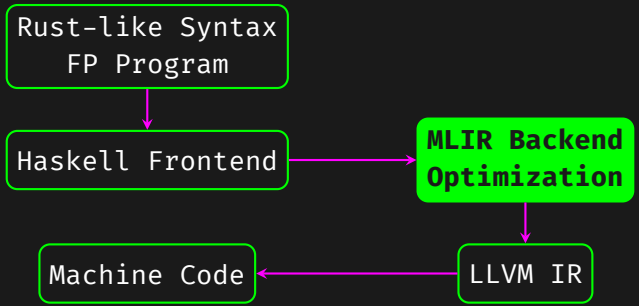
FBIP has more potential than existing work reveals.

1. **Structured Control Flow** — generalize FBIP reuse beyond pattern matching to loops, branches, early returns (analogous to MLIR's bufferization).
2. **Codegen Impact** — case study on how token reuse and inc/dec cancellation improve generated code quality (not emphasized in prior work).
3. **Lightweight FFI** — RC-based runtime embeds an ownership model into the imperative world (repr(C) headers, polymorphic FFI templates).
4. **Mixed Memory Management** — combine RC with region-based allocation for local mutable objects (flex → freeze → rigid).

Overall Design

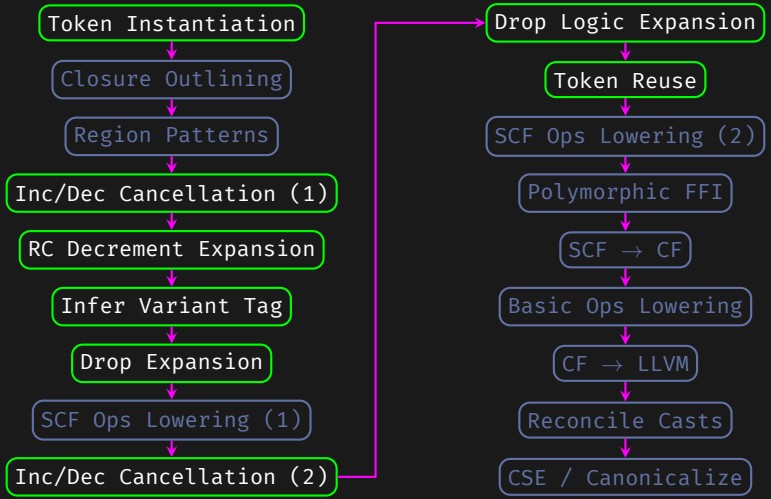


MLIR Backend Optimization



Compilation Pipeline

Detailed Compilation Pipeline



Pass Detailed

Running Example: List Reversal IR

A simplified view of the input IR (after frontend lowering):

```
func @reverse(%ls, %acc) {
  // Pattern matching on the list
  match %ls {
    Nil:
      return %acc
    Cons(%head, %tail):
      // Allocation: Create new Cons node
      %new = Cons(%head, %acc)

      // Recursive call
      return call @reverse(%tail, %new)
  }
}
```

Token Instantiation: Concept

- Similar to MLIR's bufferization, Reussir separates memory buffers from their consumers.
- We refer to available memory resources as tokens.
- **Token Instantiation** assigns tokens to token-irrelevant code by explicitly allocating new slots.
- Initially, new tokens are always allocated (no reuse yet).

Token Instantiation: Example

Before

```
// Value creation
%v = reussir.record.variant ...

// Implicit allocation?
%rc = reussir.rc.create value(%v)
```

After

```
// Value creation
%v = reussir.record.variant ...

// Explicit allocation!
%t = reussir.token.alloc
%rc = reussir.rc.create value(%v)
          token(%t)
```

Token Instantiation: Example (Producer)

Before

```
1 // RC decrement
2 reussir.rc.dec(%rc)
3 // No value returned
```

After

```
1 // RC decrement
2 // Returns a nullable token!
3 %t? = reussir.rc.dec(%rc)
```

RC Decrement Expansion: Concept

- **Inc/Dec Cancellation**: Removes redundant RC operations in straight-line code.
- **Expansion**: Lowers rc.dec into explicit control flow to handle reuse.
- Does the object have a unique reference count?
  - **Yes (1)**: We own the unique reference. Reuse the storage!
  - **No (>1)**: Others share it. Decrement and move on.

RC Decrement Expansion: Example

```
// Check reference count
%c = reussir.rc.fetch_dec(%rc)
%is_unique = arith.cmpi eq, %c, 1

// Branch on uniqueness
%t? = scf.if %is_unique {
  // Unique: reuse token logic...
  // Drop content first!
  reussir.ref.drop(%ref)
  // (e.g. reussir.rc.reinterpret)
  scf.yield %token
} else {
  // Shared: just decrement
  scf.yield null
}
```

Infer Variant Tag: Concept

- When distinct passes like Drop Expansion insert drop operations, they are generic.
- **Infer Variant Tag** uses context (e.g. match) to assign specific variant tags to these drops.
- **Benefit**: Critical for inlining the destructor and enabling further Inc/Dec Cancellation.

Infer Variant Tag: Example

Before

```
match %ls {
  Cons(...):
     // Generic drop
     // Unknown variant?
     reussir.ref.drop(%ls)
}
```

After

```
match %ls {
  Cons(...):
     // Tagged drop!
     // We know it's Cons (tag 1)
     reussir.ref.drop(%ls)
         variant[1]
}
```

Drop Expansion: Concept

- **Expansion**: We inline the destructor (one-fold) to expose individual rc.dec operations on fields. This may reintroduce new decrement operations.
- **Benefit**: These exposed decrements become visible to the Inc/Dec Cancellation pass, enabling further optimization.

Drop Expansion: Example

Inline Expansion of Cons (Variant 1):

Before

```
1 // Drop the cell
2 reussir.ref.drop(%ref)
3     variant[1]
```

After (Inlined)

```
1 // 1. Project fields
2 %tail_ref = reussir.ref.project
3     (%ref) [1]
4 %tail = reussir.ref.load(%tail_ref)
5
6 // 2. Decrement fields
7 // Exposed for cancellation!
8 reussir.rc.dec(%tail)
```

Inc/Dec Cancellation (2): Algorithm

Pushdown-Fusion Strategy

```
foreach inc(%ptr) in block:
  # Look ahead for expanded drop logic
  if next_op is scf.if (expanded_decrement):

    # Post-Dom: Is a dec op GUARANTEED to execute?
    %dec = find_post_dominating_dec(if.then, %ptr)

    if %dec exists:
       # Pushdown: Move inc to 'shared' path (else)
       move(%inc, if.else)

       # Fusion: Cancel inc with dec in 'unique' path
       erase(%dec)
```

Inc/Dec Cancellation (2): Explanation

- We attempt to fuse the inc with a matching dec hidden inside the expanded drop logic.
- If successful, we push the inc into the else branch, effectively cancelling the pair on the then path.

Drop Expansion (2): Recursive Outlining

- **Recursive Types**: We now handle recursive types by outlining the destructor to a separate function to avoid infinite inlining.
- **Cleanup**: This pass also expands any remaining rc.dec operations (including those re-introduced by the first expansion), ensuring all cleanups are finalized.

Example: Outlined Destructor

```
func @drop_in_place(%arg0) {
  match %arg0 {
    Cons(%head, %tail):
      // ... check uniqueness of tail ...
      // Recursive call!
      func.call @drop_in_place(%tail)
  }
}
```

Token Reuse: Concept

- **Goal**: Recycle memory buffers (tokens) released by destructive reads or decrements to satisfy new allocation requests.
- **Method**: A one-shot forward dataflow analysis that tracks available tokens and matches them with downstream consumers (acceptors).
- **Benefit**: Reduces heap allocations and improves locality by reusing hot cache lines.

Token Reuse: Algorithm (Dataflow)

Straight-line Analysis

```
available_tokens = {}

foreach op in block:
  # Producer: Generates a recyclable token
  if op is rc.dec or scf.if(expanded):
     available_tokens.add(op.result)

  # Acceptor: Needs a token
  if op is rc.create or token.alloc:
     # Heuristic: Match types, sizes, alignment
     best_token = find_best_match(available, op)

     if best_token:
        rewrite_as_reuse(op, best_token)
        available_tokens.remove(best_token)
```

Token Reuse: Algorithm (Control Flow)

Handling Branches & Cleanup

```
def analyze_region(region):
    # Recursively analyze sub-regions
    branch_results = [analyze(r) for r in op.regions]

    # Intersection: Only reuse if avail in ALL paths
    available = intersect(branch_results)

    # Cleanup: Tokens available but NOT taken by parent?
    # They confuse the parent scope, so free them here.
    foreach token in available_at_end_of_scope:
       if not token.dominates(parent_scope):
           insert_free(token)
```

Token Reuse: Algorithm (Safety Barriers)

Avoiding Indefinite Growth

- **Issue**: Accumulating tokens across loop iterations or unknown calls can lead to unbounded heap growth/leaks.
- **Solution**: Treat **Loops** and **Function Calls** as barriers.

```
foreach op in block:
  if op is Loop or Call:
     # Safety Barrier: Prevent indefinite growth
     free_all(available_tokens)
     available_tokens.clear()

     # Recursively analyze inside with fresh state
     analyze_regions(op, initial_tokens={})
```

Token Reuse: Example

Before

```
// Producer
%t? = reussir.rc.dec(%old)

// ... intermediate code ...

// Consumer (Allocating)
%alloc = reussir.token.alloc
%new = reussir.rc.create
    ... token(%alloc)
```

After

```
// Producer
%t? = reussir.rc.dec(%old)

// ... intermediate code ...

// Consumer (Reusing)
// No new allocation!
%reuse = reussir.token.ensure
    (%t?)
%new = reussir.rc.create
    ... token(%reuse)
```

Case Studies

Case Study: Fibonacci (Matrix Exponentiation)

Setup: Computing Fibonacci via $O(\log n)$ matrix exponentiation using RC-managed matrices.

- **Accumulation Loop**: The recursive structure optimization transforms the function into an efficient accumulation loop with minimal memory churn.
- **Operand Reuse**: In matmul(A, B), the buffer for matrix A is successfully reused for the result (or kept live), avoiding deallocation.
- **Limitation**: matmul(B, B) requires one allocation per step because B is shared (non-unique) during the call.
- **Future Work**: This could be eliminated with Inter-Procedural Analysis (IPA) or a local-free-list allocator.

Case Study: Fibonacci (Code Comparison)

Matmul Logic (Pseudo-C)

```c
Matrix* matmul(Matrix* A, Matrix* B) {
    // ---- load fields (SSA) ----
    uint64_t A_m00 = A->m00; /* ... */

    // ---- consume A and B (dec) ----
    uint64_t oldA = A->rc--;
    uint64_t oldB = B->rc--;

    // ---- decide output storage ----
    Matrix* out;
    if (oldA == 1) {
        out = A; // reuse A
    } else {
        out = (Matrix*) alloc(8, 40);
    }

    // ---- compute matrix multiply ----
    // ... (omitted) ...

    // ---- initialize output ----
    out->rc = 1; out->m00 = ...;

    // ---- free A if uniquely owned ----
    if (oldB == 1) dealloc(B, 8, 40);
```

Generated Loop (Pseudo-C)

```c
uint64_t fib_impl(uint64_t n, Matrix* acc, Matrix*
      base) {
    while (1) {
        if (n == 0) {
            uint64_t result = acc->m01;
            // consume base
            if (--base->rc == 0) free(base);
            return result;
        }

        if (n & 1) {
            // odd: acc = acc * base
            base->rc++; // prepare for alias-safe
                  consume
            Matrix* new_acc = matmul(acc, base);
            acc = new_acc;
        }

        // base = base * base
        base->rc++;     // prepare self-alias
        Matrix* new_base = matmul(base, base);
        base = new_base;
        n >>= 1;
    }
}
```

Case Study: Tree Mirror (Non-Tail Recursion)

Original (Reussir)

```
fn mirror(t : Tree) -> Tree {
    match t {
        Tree::Leaf(x) =>
            Tree::Leaf{x},
        Tree::Node(l, r) =>
            Tree::Node{
                mirror(r),
                mirror(l)
            }
    }
}
```

Generated Logic (Pseudo-C)

```
Tree* mirror(Tree* t) {
  if (t->tag == LEAF) {
    if (--t->rc == 0) {
       t->rc = 1; return t; // Reuse!
    }
    return alloc_leaf(t->val);
  }

  // NODE case
  Tree *l = t->left, *r = t->right;

  // Smart Ownership Transfer
  if (--t->rc == 0) {
     free(t); // Free shell only
     // Children moved to recursive calls
  } else {
     l->rc++; r->rc++; // Share children
  }

  // Recursive calls (Barrier for reuse)
  Tree* new_r = mirror(r);
  Tree* new_l = mirror(l);

  return alloc_node(new_r, new_l);
}
```

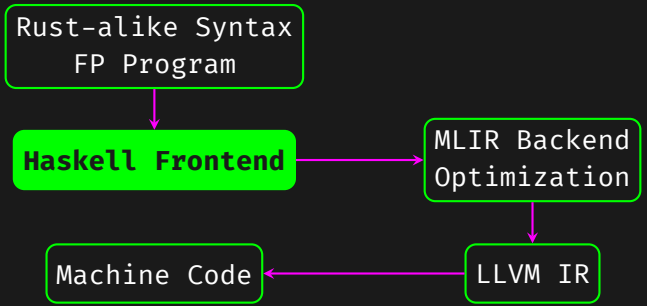
Case Study: Tree Mirror (Discussion)

Key Insights

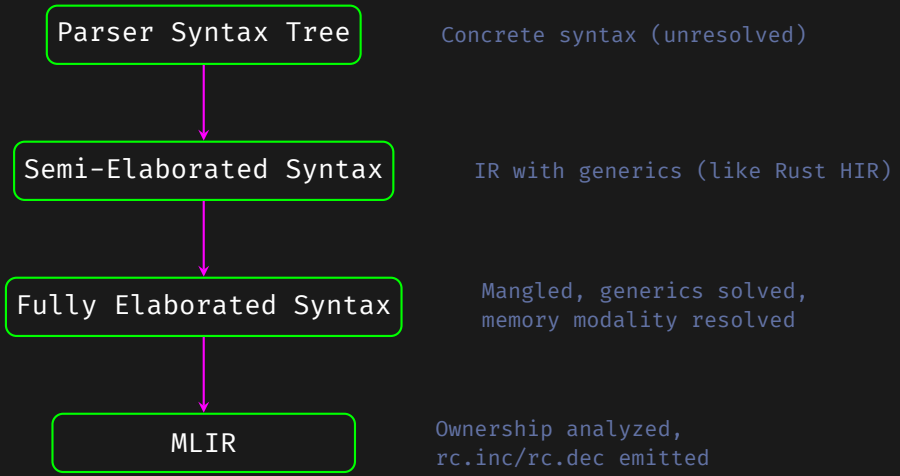
- **Fused Ownership**: Compared with Rust's Rc<Tree>, no separate "deep free" pass. Destruction of the tree structure is fused with the recursive traversal.
- **Efficiency**: RC management prevents the "double traversal" (drop then recurse) often seen in other systems.
- **Reuse Truncation**:
  - The `Node` buffer itself is freed and re-allocated, rather than reused.
  - **Reason**: The recursive calls to `mirror(r)` and `mirror(l)` act as barriers. We cannot keep the token alive across these calls due to potential heap growth.

Frontend and Language Features

Haskell Frontend



Frontend Pipeline



Why Two-Stage Elaboration?

Records in Reussir carry a memory modality annotation:

- [value] — copy semantics, no RC, stored inline.
- [shared] — immutable, reference-counted (Rc<T, Shared>).
- [regional] — mutable in region, reference-counted (Rc<T, Flex/Rigid>).

The modality determines **how memory operations are generated**. Value types need no RC ops; Shared/Regional types need rc.inc/rc.dec inserted by ownership analysis.

⇒ We must fully resolve modality and generics **before** ownership analysis can decide which operations to emit.

Stage 1: Semi-Elaboration

- Parse surface syntax into a **Semi-Elaborated IR**.
- Bidirectional type inference with unresolved generics.
- Types carry flexivity (Irrelevant, Flex, Rigid, Regional) — not yet concrete memory decisions.
- Collect all generic instantiations across the program.

Analogy: Similar to Rust's HIR — types are known but monomorphization has not happened yet.

Stage 2: Full Elaboration

1. **Monomorphize**: Instantiate all generic records and functions with concrete types.
2. **Resolve modality**: Map flexivity to concrete capabilities:
   - [value] → TypeRecord (bare, no RC wrapper)
   - [shared] → TypeRc(T, Shared)
   - [regional] + [flex] → TypeRc(T, Flex)
   - [regional] + [rigid] → TypeRc(T, Rigid)
3. **Mangle names**: e.g. Cell<u64> → _RIC4CellyE

After this stage, every type is concrete — ready for ownership analysis.

PST → Semi: Bidirectional Type Checking

We use bidirectional type checking to elaborate the parse tree:

- **Infer mode** ($\Gamma \vdash e \Rightarrow A$): Synthesize the type from the term. Used for literals, variables, projections, function calls.
- **Check mode** ($\Gamma \vdash e \Leftarrow A$): Verify the term against an expected type. Used for let bindings with annotations, return positions.

Generics remain as unresolved holes (?T) during this stage — each call site records which concrete types flow into generic parameters.

Semi → Full: Flow-Based Generic Solver

Between semi and full elaboration, a flow-based generic solver resolves all generic instantiations:

- Collect all type flows $\tau \rightsquigarrow ?T_i$ at call sites.
- Build a constraint graph and propagate concrete types.
- Produce a **GenericSolution**: GenericID → [Type].

Based on the algorithm from:

"Flow-Sensitive Type Inference for Scripting Languages"
OOPSLA 2025 doi:10.1145/3720472

The solution drives monomorphization — each unique instantiation produces a separate fully-elaborated function and record.

Demo: Value Modality

```
struct [value] Matrix {
    m00: u64, m01: u64,
    m10: u64, m11: u64
}

fn matmul(a : Matrix, b : Matrix) -> Matrix {
    let m00 = a.m00 * b.m00 + a.m01 * b.m10;
    let m01 = a.m00 * b.m01 + a.m01 * b.m11;
    let m10 = a.m10 * b.m00 + a.m11 * b.m10;
    let m11 = a.m10 * b.m01 + a.m11 * b.m11;
    Matrix { m00: m00, m01: m01, m10: m10, m11: m11 }
}
```

[value] ⇒ **No RC overhead**. Matrix is passed and returned by value (copy semantics).

Demo: Shared Modality

```
struct [value] Box<T> { val : T }
struct [shared] RcBox<T> { val : T }

pub fn rc_project(rc : RcBox<u32>) -> u32 {
    rc.val
}

pub fn rc_project_chained(rc : RcBox<RcBox<u32>>) -> u32 {
    rc.val.val
}
```

[shared] ⇒ **Immutable, reference-counted**. RcBox<u32> becomes Rc<RcBox<u32>, Shared> in Full IR. Projections borrow the inner value — no copies needed.

Demo: Regional Mutation ([field])

```
struct [regional] DLLink<T> {
    val: T,
    next: [field] DLLink<T>,
    prev: [field] DLLink<T>
}

regional fn push_back<T>(
    cursor : [flex] DLLink<T>, elem : T
) {
    let new_node = new(elem);
    new_node->prev := Nullable::NonNull{cursor};
    cursor->next := Nullable::NonNull{new_node};
}
```

[field] marks mutable fields within regional structs. The → operator enables in-place mutation on [flex] references.

Ownership Analysis

After full elaboration, **Ownership Analysis** emits rc.inc/rc.dec:

- Each resource-relevant (RR) parameter starts with ownership $= 1$.
- **Consuming uses** (function args, returns) transfer ownership.
- **Borrowing uses** (projections, match scrutinees) do not consume.
- **Early dec**: rc.dec placed at earliest point after last use.
- **Branch reconciliation**: All branches must reach same ownership state.

Polymorphic FFI

The FFI Challenge in FP Languages

RC-based runtime naturally embeds an ownership model into the imperative world:

- repr(C) headers — ABI-compatible RC layout.
- Clone ↔ rc.inc, Drop ↔ rc.dec.
- Rust code can hold and manipulate Reussir objects directly.

Problem: This only works for monomorphic types.

For **polymorphic** types (Vec<T>, HashMap<K,V>), Lean/Koka resort to boxed values — uniform representation with runtime type dispatch, losing type-specific layout and performance.

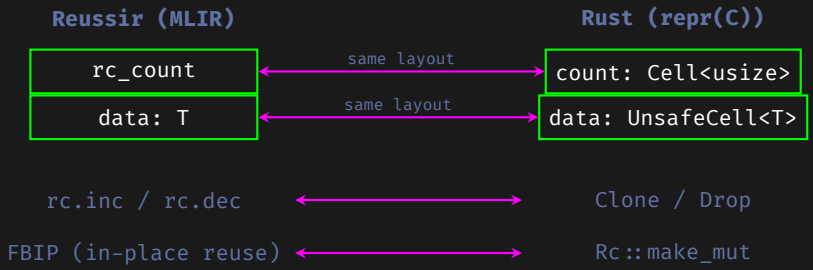
Reussir's Approach: Monomorphic Instantiation

Key insight: Since Reussir already monomorphizes generics during full elaboration, we can do the same for FFI types.

- FFI libraries provide template Rust source with [:typename:] placeholders.
- When generics are solved, substitute concrete types and compile each instantiation separately.
- No boxing — Vec<i64> and Vec<f64> get their own specialized code, just like native Rust.

Advantage: Zero-overhead interop with Rust's type-specialized collections and data structures.

RC Layout: Bridging Two Worlds



repr(C) ensures identical memory layout. Rust's make_mut is the same idea as FBIP: mutate in-place when uniquely owned, copy otherwise.

Polymorphic FFI: Template Mechanism

```
// MLIR: Before compilation
reussir.polyffi
    texture("
        extern crate reussir_rt; // shipped as HIR Rlib
        use reussir_rt::collections::vec::Vec;

        pub unsafe extern "C"
        fn vec_new() -> Vec<[:typename:]> {
            Vec::new()
        }

        pub unsafe extern "C"
        fn vec_push(v: Vec<[:typename:]>,
                    e: [:typename:])
            -> Vec<[:typename:]> {
            v.push(e)
        }
    ")
    substitutions({"typename" = "f64"})
```

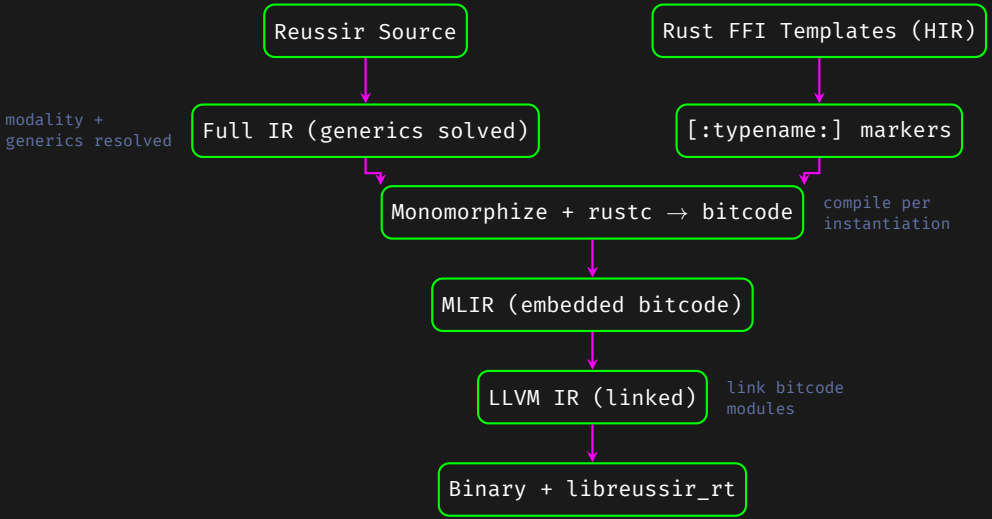
Polymorphic FFI: After Compilation

```
// Opaque FFI type with cleanup hook
!Vecf64 = !reussir.rc<!reussir.ffi_object<
    "::reussir_rt::collections::vec::Vec<f64>",
    @__polyffi_Vec_f64_drop
>>

// MLIR: After CompilePolymorphicFFI pass
reussir.polyffi
    compiled(dense_resource<blob> : tensor<...xi8>)
    // ^ Embedded LLVM bitcode from rustc

// Generated declarations are available
func.func private @vec_f64_new() -> !Vecf64
func.func private @vec_f64_push(
    !Vecf64, f64) -> !Vecf64
```

Compilation Pipeline for FFI



Boxed vs Monomorphic: Comparison

Lean / Koka

Vec<T> → Array Object

- Uniform boxed representation
- Runtime type dispatch
- Extra indirection for primitives
- Cannot inline type-specific ops

Reussir

Vec<T> → Vec<f64>, Vec<i64>, …

- Type-specialized code per instantiation
- No boxing overhead
- Native Rust Vec layout
- Full LLVM optimization across FFI

Thank you!

Reussir

A Memory Reuse IR Playground for FP Languages