Simulating Modal Memory Management in Standard OCaml 5

A Library-Based Approach to Locality and Ownership

Vishrut Shukla

December 2, 2025

The Context: The Need for Safety

The Problem

- OCaml 5 introduces Multicore (parallelism), making memory safety critical to avoid race conditions.
- Developers want **Rust-like control** over memory:
  - **Locality:** Allocating on the Stack (Fast, GC-free).
  - **Uniqueness:** Single ownership (Safe in-place mutation).

The Current Gap

Achieving this currently requires **modifying the OCaml compiler** (Ref: *Oxidizing OCaml*, Lorenzen et al., ICFP 2024). This is a high barrier to entry.

The Objective

Core Question: Can we enforce modal memory safety without a custom compiler?

- **Goal:** Implement a library (`OxidizeLib`) that simulates compiler "Modes" using standard OCaml types.
- **Strategy: Type-Level Programming**
  - **Phantom Types:** To label data (e.g., `stack` vs `heap`).
  - **GADTs:** To enforce these labels at compile time.
- **Differentiation:**
  - Unlike `linocaml` (which uses rigid Monads), this project aims for a **Direct Style** API that feels like normal OCaml.

Details: The "Phantom" Technique

We use types that don't exist at runtime to "tag" our data.

1. Define Empty Tags

```
type stack
type heap
```

2. Define Tagged Pointer

```
(* 'mode is a Phantom Type *)
type 'mode pointer = int
```

3. Enforce with GADTs

```
type _ ptr =
  | Stack : int -> stack ptr
  | Heap  : int -> heap ptr

(* Logic Error = Compile Error *)
let free (p : heap ptr) = ...
(* Passing 'Stack' here fails! *)
```

Details: Enforcing Locality

How do we prevent a stack pointer from leaking?

- **Mechanism:** Rank-2 Polymorphism.
- **Logic:** We generate a unique ”Region ID” that is valid *only* inside the function scope.

```
(* The type system ensures ’r cannot escape *)
type ’a region_scope = {
  run : ’r. ((’r, ’a) t -> ’a)
}

let result = with_region 5 {
  run = fun ptr ->
    ptr.data + 1 (* Safe to return int *)
    (* ptr       (* ERROR: Cannot return ptr! *) *)
}
```

Details: Handling Uniqueness

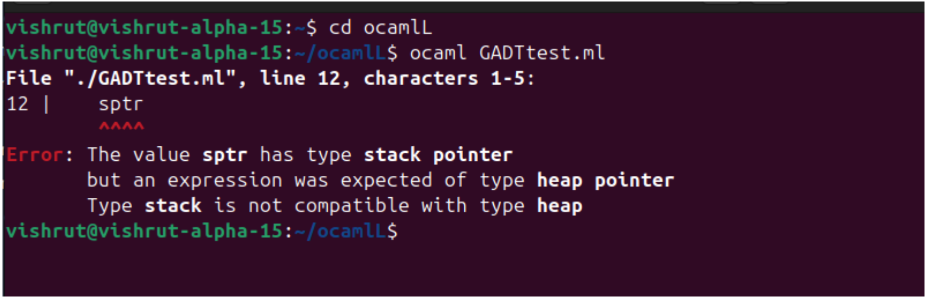
Challenge: OCaml allows variable aliasing (`let y = x`), so static uniqueness is impossible.

My Solution: Hybrid Enforcement

1. **Static Guidance (GADTs):** We model the resource lifecycle as a State Machine in the type system. Functions require an `Alive` type.
2. **Runtime Safety (Destructive Moves):** Since we cannot stop aliasing at compile time, we use a **Destructive Move**.
   - When `consume(x)` is called, the underlying reference is nullified.
   - If aliased variable `y` is used later, it hits a safe runtime error instead of a Segfault.

Results: Proof of Concept

Scenario: Creating a Stack pointer and attempting to pass it to a Heap deallocator.



```
vishrut@vishrut-alpha-15:~$ cd ocamlL
vishrut@vishrut-alpha-15:~/ocamlL$ ocaml GADTtest.ml
File "./GADTtest.ml", line 12, characters 1-5:
12 |     sptr
         ^^^^
Error: The value sptr has type stack pointer
       but an expression was expected of type heap pointer
       Type stack is not compatible with type heap
vishrut@vishrut-alpha-15:~/ocamlL$
```

Conclusion: The standard OCaml compiler successfully enforces the modal constraints defined in our library.

Evaluation & Current Status

Current Status

- ✓ Core Phantom/GADT Types defined.
- ✓ **Locality Module:** Working (Rank-2 Polymorphism).
- → **Unique Module:** In progress (Implementing destructive move logic).

Performance Analysis

- **Locality:** Zero Runtime Overhead (Types are erased).
- **Uniqueness:** Low Overhead (Option check).
- **Ergonomics:** Direct style code is cleaner than Monadic alternatives.

Conclusion

Summary

This project bridges the gap between theoretical safety research (*Oxidizing OCaml*) and practical engineering. We demonstrate that **Standard OCaml 5** is powerful enough to enforce advanced memory disciplines without compiler modifications.

Deliverable: The full `OxidizeLib` library will be delivered on the 9th, containing the complete Region (Locality) and Unique (Ownership) modules.

Thank you. Questions?