11 OxCaml: Safe Control over Program Behaviour

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

OxCaml

- **A performance-oriented extension of the OCaml programming language**
  - A super set of OCaml
  - Every OCaml program is a valid OxCaml program
- Extensions
  - **Control:** over allocations and memory layout
  - **Safe:** data race freedom and memory safety
- New ingredient — Modes
  - Types are ***what*** values are, modes are ***how*** values may be used
  - Types prevent — string value to be added to an int value
  - Modes prevent — a plain reference to be accessed concurrently by two different domains

Revisiting lecture with OxCaml

- Lecture 2 — mutual exclusion with mutex
  - Forgot to lock, we get a data race
  - OxCaml's ***contention*** mode statically prevents this
- Lecture 5 — spin lock implementation
  - Used TSAN to detect data races dynamically
  - OxCaml's ***portability and contention*** mode statically prevent this
- Lecture 9 & 10 — effect handlers
  - Unhandled effects are detected dynamically
  - OxCaml catches them statically through ***locality*** mode

OxCaml Lecture Handout

- See cs6868_s26/lectures/11_oxcaml/handout.md
- We will only cover a small part of it.

GenSym

```
let gensym =
  let count = ref 0 in
  fun prefix ->
    count := !count + 1;
    prefix ^ "_" ^ string_of_int !count

let _ = Domain.Safe.spawn (fun () -> gensym "x")
```

```
File "./gensym_unsafe.ml", line 9, characters 37-43:
9 | let _ = Domain.Safe.spawn (fun () -> gensym "x")
                                         ^^^^^^
Error: The value gensym is nonportable but is expected to be portable
       because it is used inside the function at File "./gensym_unsafe.ml", line 9, characters 26-48
       which is expected to be portable.
```

Locality Mode

Locality Mode

- Locality mode
  - Values with @ local mode may be allocated “on the stack”
  - **Local** values cannot escape their region!
  - **Global** values may be treated as local values
    - Global values can be used more freely, and we can take away rights
  - The default mode is global

```
let use_locally (r @ local) = !r + 1

let _test_use_locally () =
  let r = stack_ (ref 41) in
  use_locally r


let _test_use_locally2 () =
  let r = ref 41 in
  use_locally r
```

Will this compile?

Yes, global values can be considered local

Local point

```
(* Stack allocation with stack_. *)

type point = { x : float; y : float }

let distance (a @ local) (b @ local) =
  let dx = a.x -. b.x in
  let dy = a.y -. b.y in
  Float.sqrt (dx *. dx +. dy *. dy)

let test_distance () =
  let a = stack_ { x = 0.0; y = 0.0 } in
  let b = stack_ { x = 3.0; y = 4.0 } in
  let d = distance a b in
  d
```

Escaping store

```
type point = { x : float; y : float }

let storage : point ref = ref { x = 0.0; y = 0.0 }

let store_local () =
  let p = stack_ { x = 1.0; y = 2.0 } in
  storage := p

File "./part1_escape_store.ml", line 9, characters 13-14:
9 |   storage := p
                 ^
Error: This value is local
       because it is stack_-allocated.
       However, the highlighted expression is expected to be global.
```

Escaping Return

```
type point = { x : float; y : float }

let escape_demo () =
  let p = stack_ { x = 1.0; y = 2.0 } in
  p

File "./part1_escape_return.ml", line 7, characters 2-3:
7 |   p
      ^
Error: This value is local
       because it is stack_-allocated.
       However, the highlighted expression is expected to be local to the parent region or global
       because it is a function return value.
       Hint: Use exclave_ to return a local value.
```

Exclave

```
(* Returning local values with exclave_. *)

let midpoint (a @ local) (b @ local) : point @ local =
  exclave_ { x = (a.x +. b.x) /. 2.0; y = (a.y +. b.y) /. 2.0 }

let translate (p @ local) (dx : float) (dy : float) : point @ local =
  exclave_ { x = p.x +. dx; y = p.y +. dy }

let[@zero_alloc] [@inline never] rec translate_polyline
    (poly : point list @ local) dx dy : point list @ local =
  match poly with
  | [] -> exclave_ []
  | p :: rest ->
      exclave_ (translate p dx dy :: translate_polyline rest dx dy)
```

Working with lists — zero alloc

```
type point = { x : float; y : float }

let distance (a @ local) (b @ local) =
  let dx = a.x -. b.x in
  let dy = a.y -. b.y in
  Float.sqrt (dx *. dx +. dy *. dy)

let[@zero_alloc] [@inline never] rec path_length
    (poly : point list @ local) : float =
  match poly with
  | a :: ((b :: _) as rest) -> distance a b +. path_length rest
  | _ -> 0.0
```

Working with lists — zero alloc fail!

```
% ocamlopt part1_path_length_alloc_fail.ml
File "part1_path_length_alloc_fail.ml", line 14, characters 5-15:
14 | let[@zero_alloc] [@inline never] rec path_length
          ^^^^^^^^^^

Error: Annotation check for zero_alloc failed on function Part1_path_length_alloc_fail.path_length
(camlPart1_path_length_alloc_fail__path_length_1_3_code).

File "part1_path_length_alloc_fail.ml", line 17, characters 29-41:
17 |   | a :: (b :: _ as rest) -> distance a b +. path_length rest
                                 ^^^^^^^^^^^^

Error: called function may allocate (direct call camlPart1_path_length_alloc_fail__distance_0_2_code)

File "part1_path_length_alloc_fail.ml", line 17, characters 29-61:
17 |   | a :: (b :: _ as rest) -> distance a b +. path_length rest
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error: allocation of 16 bytes for float

File "part1_path_length_alloc_fail.ml", line 17, characters 45-61:
17 |   | a :: (b :: _ as rest) -> distance a b +. path_length rest
                                                 ^^^^^^^^^^^^^^^^

Error: called function may allocate (direct call camlPart1_path_length_alloc_fail__path_length_1_3_code)
```

Working with lists — Unboxed Floats

```
let[@zero_alloc] [@inline never] distance_u
    (a @ local) (b @ local) : float# =
  let open Float_u in
  let dx = of_float a.x - of_float b.x in
  let dy = of_float a.y - of_float b.y in
  sqrt (dx * dx + dy * dy)

let[@zero_alloc] [@inline never] rec path_length_u
    (poly : point list @ local) (acc : float#) : float# =
  let open Float_u in
  match poly with
  | a :: (b :: _ as rest) -> path_length_u rest (acc + distance_u a b)
  | _ -> acc
```

Summary — Locality Mode

- Allocation on the stack
- Modes — local and global
  - Global is default
  - Local values may be allocated on the stack
  - Global values may be considered local
- stack_ and exclave_ to control allocations on the stack
- float is a value on the heap!
- @zero_alloc to verify that functions don’t allocate on the heap!

Portability and Contention Mode

Two model axes

- **Contention**
  - Whether a value is accessed by 2+ domains
  - Modes are **uncontended** (default) ⊑ **shared** ⊑ **contended**
  - **Shared** is shared, but reads only
  - **Contended** is shared with at least one being a write
- **Portability**
  - Whether a (usually, function) value can *cross* domain boundaries
  - Modes are **portable** ⊑ **nonportable** (default)

Contention Axis

- **Rule 1:** If a value is contended, you cannot read or write to its mutable fields
- **Rule 2:** Everything inside a contended value is also contended. You can't extract an uncontended component from a contended container.

```
type mood = Happy | Neutral | Sad

type thing = { price : float; mutable mood : mood }
```

Reading and immutable field is *fine* even when contended

```
let price_contended (t @ contended) = t.price
```

Writing to a mutable field on a contended field is *rejected*

```
let cheer_up_contended (t @ contended) = t.mood <- Happy

25 | let cheer_up_contended (t @ contended) = t.mood <- Happy
                                               ^

Error: This value is contended but is expected to be
       uncontended because its mutable field mood is
       being written.
```

Contention Axis

```
type mood = Happy | Neutral | Sad

type thing = { price : float; mutable mood : mood }

let read_mood_contended (t @ contended) = t.mood

File "./part2_slides.ml", line 34, characters 42-43:
34 | let read_mood_contended (t @ contended) = t.mood
                                              ^

Error: This value is contended but expected to be shared.
  Hint: In order to read from the mutable fields,
  this record needs to be at least shared.
```

Even reading a mutable field on a contented value is *rejected*

Portability axis

- A portable (function) value can safely cross domain boundaries
- **Important:** inside a portable function, all captured values are treated as contended

A pure function is portable

```
let test_portable () =
  let (f @ portable) = fun x y -> x + y in
  f 1 2
```

Capturing a mutable ref makes a function non-portable

```
let test_nonportable () =
  let r = ref 0 in
  let (counter @ portable) () = incr r; !r in
  counter ()

File "./part2_slides.ml", line 63, characters 37-38:
63 |   let (counter @ portable) () = incr r; !r in
                                         ^

Error: This value is contended but expected to be uncontended.
```

Ruling out data races with Portability and Contention

- Data races prevented with a two-step argument

    1. **Portability:** ensures that closures crossing a domain boundary are portable $\Longrightarrow$ they treat captured values as contended
    2. **Contention:** ensures that contended values can’t have their mutable fields read or written

Portability — capture vs arguments

- Portability only expects ***captured*** valued to be contended
  - Function arguments can be used at their declared mode

```
let factorial_portable n =
  let a = ref 1 in
  let rec (loop @ portable) (a @ uncontended) i =
    if i > 0 then begin
      a := !a * i;
      loop a (i - 1)
    end
  in
  Domain.spawn (fun () -> loop a n);
  !a
```

First working program

```
open Portable

let gensym =
  let count = Atomic.make 0 in
  let res @ portable = fun prefix ->
    let n = Atomic.fetch_and_add count 1 in
    prefix ^ "_" ^ string_of_int n
  in
  res

let d = Domain.Safe.spawn (fun () -> gensym "y")
let s1 = gensym "x"
let s2 = Domain.join d
let () = Printf.printf "%s %s\n" s1 s2
(* prints:  x_0 y_1 *)
```

- **Portable.Atomic** crosses both Contention and Portability axes
  - Since it has synchronisation, it can freely be used in cases where one of the modes is expected

No compile time safety in vanilla OCaml

```
(* Standard OCaml — no compile-time safety *)
let mutex = Mutex.create ()
let shared_table = Hashtbl.create 16

let safe_insert k v =
  Mutex.lock mutex;
  Hashtbl.add shared_table k v;
  Mutex.unlock mutex

(* Nothing stops you from forgetting the lock: *)
let unsafe_insert k v =
  Hashtbl.add shared_table k v   (* DATA RACE — compiles fine! *)
```

Capsules: Compile-time Lock Discipline

- A capsule is a ***branded*** container for mutable state
  - Brand is a type-parameter (implicit) that connects data to its lock
  - Cannot access the data without ***proving*** you hold the lock at the type-level
- Components
  - **Capsule.Mutex.t** — a mutex carrying a brand
  - **Capsule.Data.t** — the encapsulated data, sharing the same brand
  - **access token** — proof that you hold the lock; required for unwrapping the data

Capsules

```
let gensym =
  (* `P mutex` introduces a fresh existential brand $k tied to this mutex. Anything we want this mutex
     to protect must end up branded with $k, and the only way to obtain a $k-branded `access` token is
     to call with_lock on this very mutex. *)
  let (P mutex) = Await_capsule.Mutex.create () in
  (* The ref is created INSIDE Capsule.Data.create — it has no name in scope outside, so the only
     handle to it is `counter`, which is branded $k. The bare `ref` cannot be aliased out. *)
  let counter = Capsule.Data.create (fun () -> ref 0) in
  let fetch_and_incr (w : Await.t) =
    (* with_lock acquires the mutex and hands the body an `access` token branded $k. `w : Await.t` is
       the awaiter — needed because acquiring the lock may suspend the fiber. *)
    Await_capsule.Mutex.with_lock w mutex
      ~f:(fun access ->
        (* Capsule.Data.unwrap requires a brand-matching access token. Outside this body, no $k token
           exists, so no one can reach `c`. *)
        let c = Capsule.Data.unwrap ~access counter in
        incr c;
        !c)
  in
  fun w prefix -> prefix ^ "_" ^ Int.to_string (fetch_and_incr w)
```

Summary

- **Portability** and **Contention** modal axes work together to prevent data races statically
- **Portable.Atomic.t** ***crosses*** portability and contention axes
- OxCaml has
  - Other modes
    - Linearity — how many times a value can be used $\Longrightarrow$ memory-safety of linear resources
    - Uniqueness — aliasing
  - Higher-level building block libraries — capsules, immutable arrays, fork-join parallelism
  - Performance-oriented extensions — SIMD, unboxed types
- See **handout.md**