Safe Stack Allocation in Standard ML

Thomas Schollenberger (tss2344@cs.rit.edu)
Advisor: Dr. Matthew Fluet

Introduction

Standard ML (SML) uses garbage collection to ensure safety, abstracting away manual memory management at the expense of low-level control.

Problem: Heavy heap allocation of temporary values triggers frequent, unpredictable “stop-the-world” GC pauses.

Existing Solutions: Escape analysis or ownership models are either invisible to the programmer or introduce significant complexity.

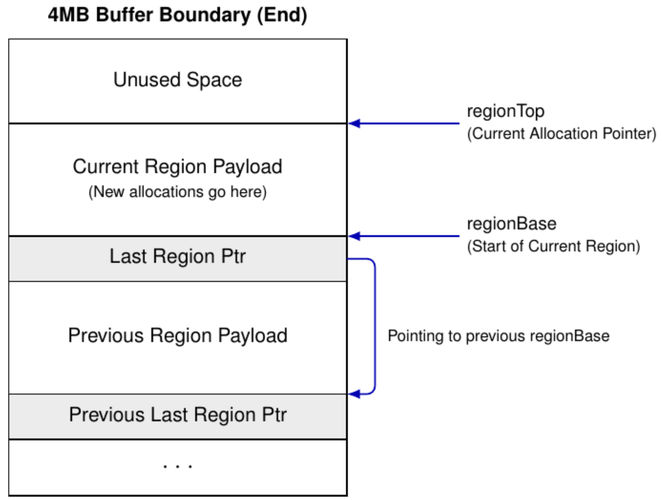
Goal: Give SML programmers explicit, type-safe control to safely stack allocate data, reducing GC pressure, while ensuring memory safety and backwards compatibility.

Example Code

```
fun map_exclave f [] = []
  | map_exclave f (x :: xs) =
      exclave_ (f x :- stack_)
        :: (map_exclave f xs :- stack_)

fun tabulate_exclave n f =
  if n <= 0 then []
  else
    exclave_
      (f n :- stack_) :: tabulate_exclave (n - 1) f

val list1 :- stack_ = List.tabulate_exclave n (fn x => x)
val list2 :- stack_ = List.map_exclave (fn x => x * 2) list
val sum = List.foldl op+ 0 list2
```



Modal Allocation

We extend SML to support the ability for programmers to allocate short-lived values on the stack. To enable this, we introduce a *modal type system*, which associates value bindings with modes describing where they may live.

Modes express where values live:

- stack: lives in the current function’s stack frame (lexical lifetime).
- heap: lives on the GC-managed heap (may outlive the defining function.
- constant: immediate values/literals that require no allocation,
- undetermined: initial mode for unannotated bindings during inference.

No-escape Invariant:

If a value is annotated or inferred to be stack, that value may not escape it’s defining function.

In particular, a stack value cannot be: stored in the heap, returned from the defining function, or stored in a longer-lived stack frame.

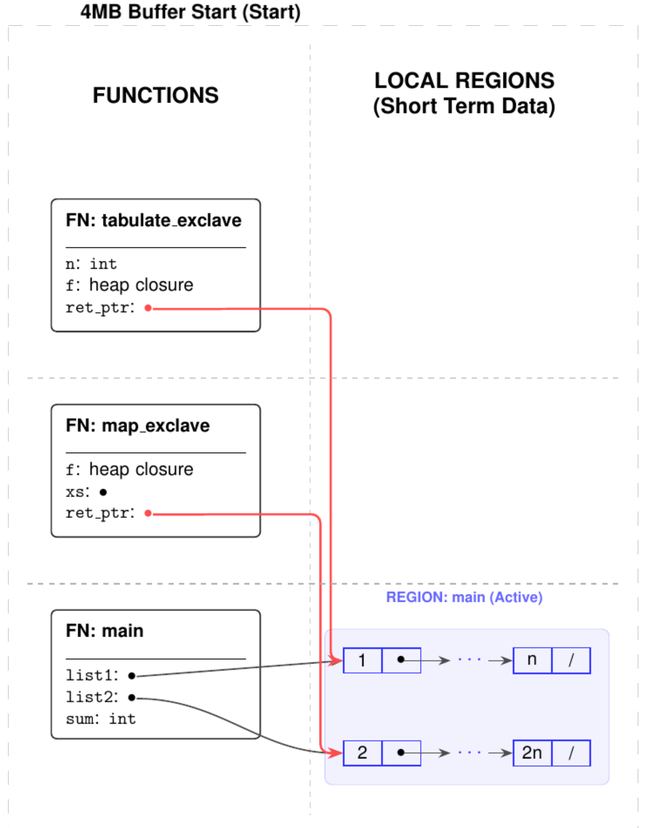
We implement a **region-based memory model** to handle allocating temporaries efficiently while preserving stack-like reclamation.

Region runtime mechanics:

- **Region stack:** Functions that may hold stack-allocated values are associated with a region. Entering such a function pushes a new region; returning pops it.
- **Reclamation:** Popping a region reclaims all of its stack-allocated values in constant time *O(1)* by resetting an allocation pointer.

The exclave primitive:

- exclave e: evaluates e such that allocations performed by e occur in the caller’s region instead of the current frame’s region.
- **Safety:** The type-and-mode system ensures that e does not capture stack values from the current (soon-to-be-popped) frame; all captured values must be valid for the caller’s region.



Results

We implement evaluate our modal allocation system on a synthetic allocation-heavy workload. The benchmark repeatedly constructs lists and performs simple list operations to simulate a hot path with high allocation pressure.

Iterations	Baseline Runtime	Stack Mode Enabled Runtime
1,000	27 ms	17 ms
5,000	702 ms	404 ms
10,000	2,939 ms	1,619 ms
50,000	77,681 ms	41,826 ms
100,000	284,801 ms	146,985 ms

These results show a consistent **speedup** of approximately 30%. This result is consistent with expectations. SML programs spend roughly 30% of execution time in garbage collection on this workload.

Future Work & Conclusion

- **Validation:** Apply modal stack allocation to larger SML projects, comparing against escape analysis and purely GC-based allocation..
- **Array Primitive Extensions:** Support modal allocation for the built-in array primitive.
- **Feasibility:** Extending SML to support stack allocation via modes and regions is feasible without breaking backwards compatibility.
- **The middle ground:** Modal memory management offers a balance between pure GC and fully manual memory control.