A Mechanically Verified Garbage Collector for OCaml

KC Sivaramakrishnan
kcsrk.info

IARCS Verification Seminar Series
18th November 2025

IIT MADRAS

Tarides

OCaml

📜 Language

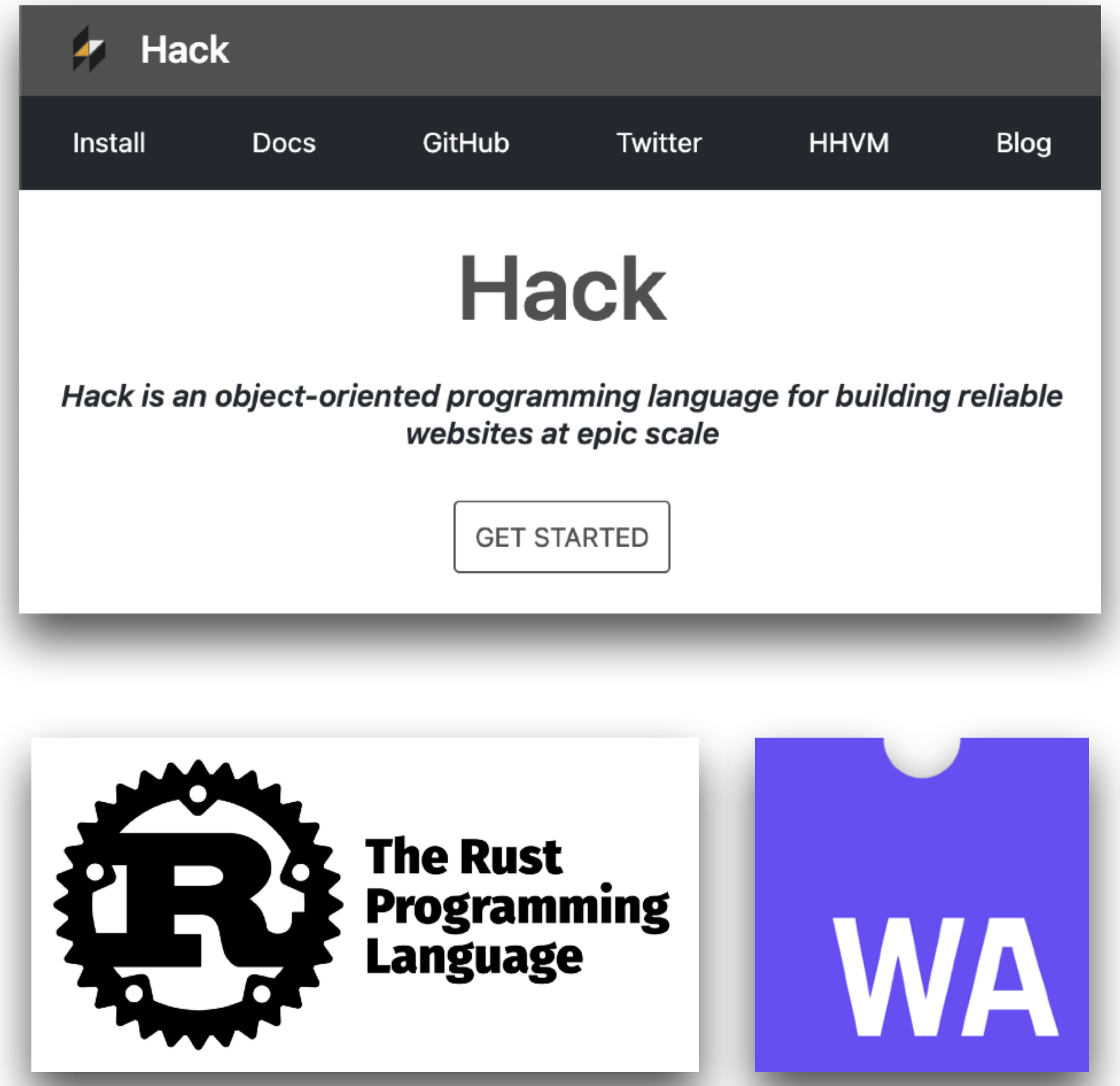
- Functional-first but multi-paradigm (imperative, OO)
- Static-type system with Hindley-Milner type inference
- Advanced features — powerful module system, GADTs, Polymorphic variants
- Multicore support and effect handlers

⚙️ Platform

- Fast, native code— x86, ARM, RISC-V, etc.
- JavaScript and WebAssembly (using WasmGC) compilation

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure



Compilers

Verification & Static Analysis

High dynamic range

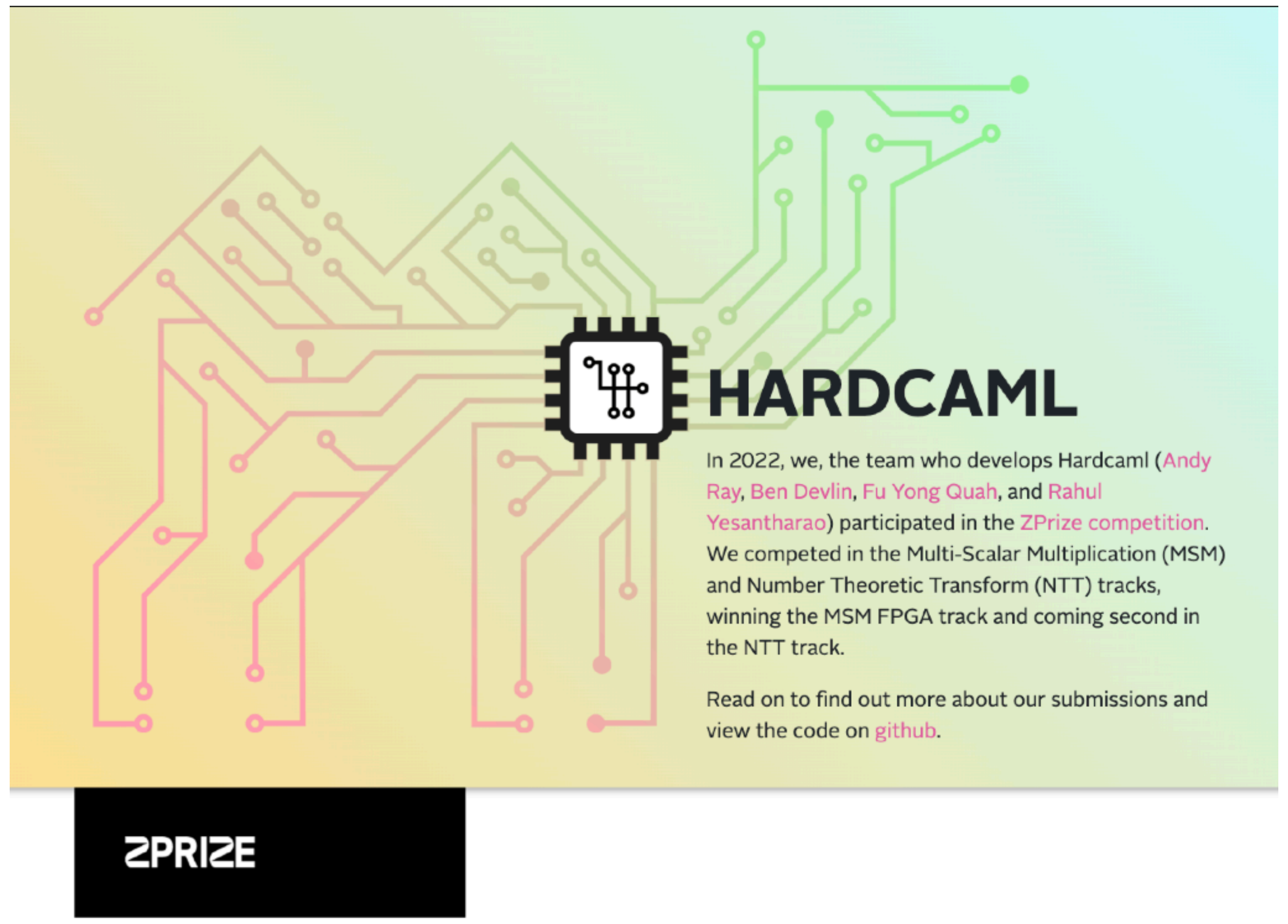
From scripts to scalable systems, research prototypes to production infrastructure

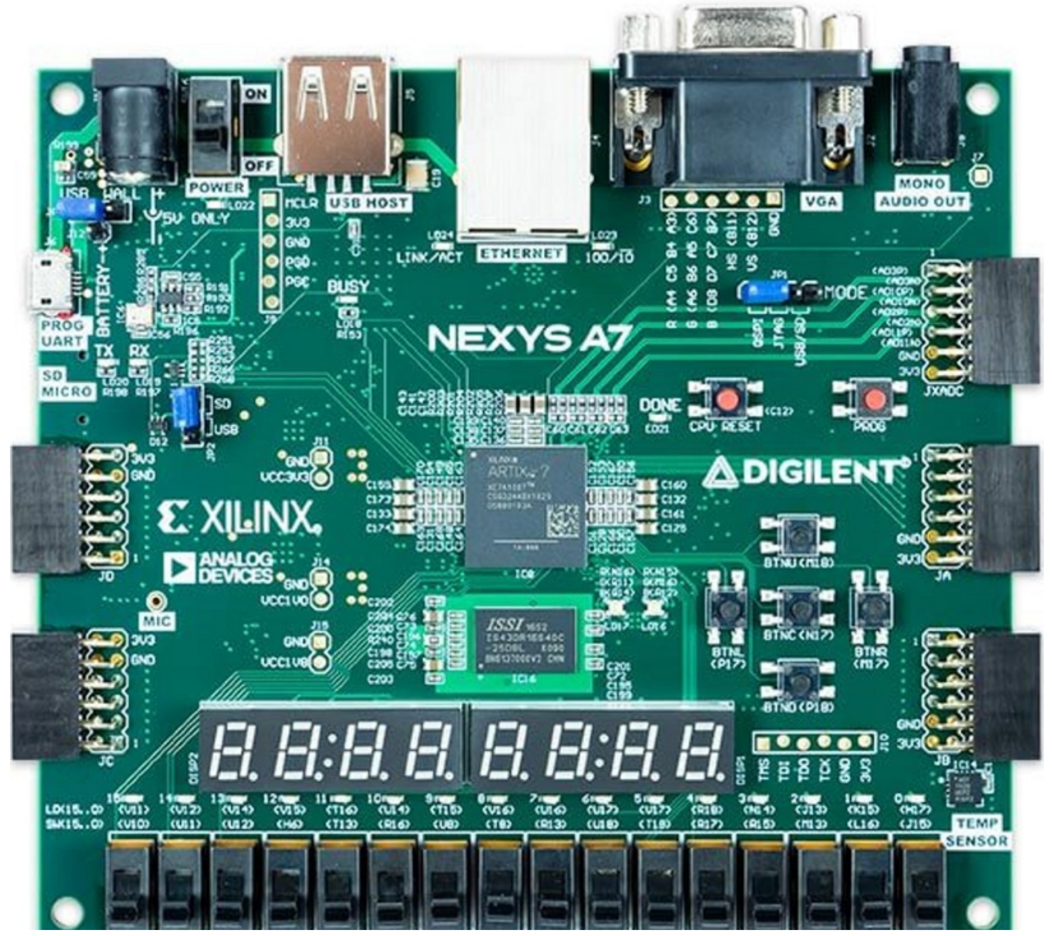
Jane Street

60+M lines of OCaml code!

Bloomberg

Finance

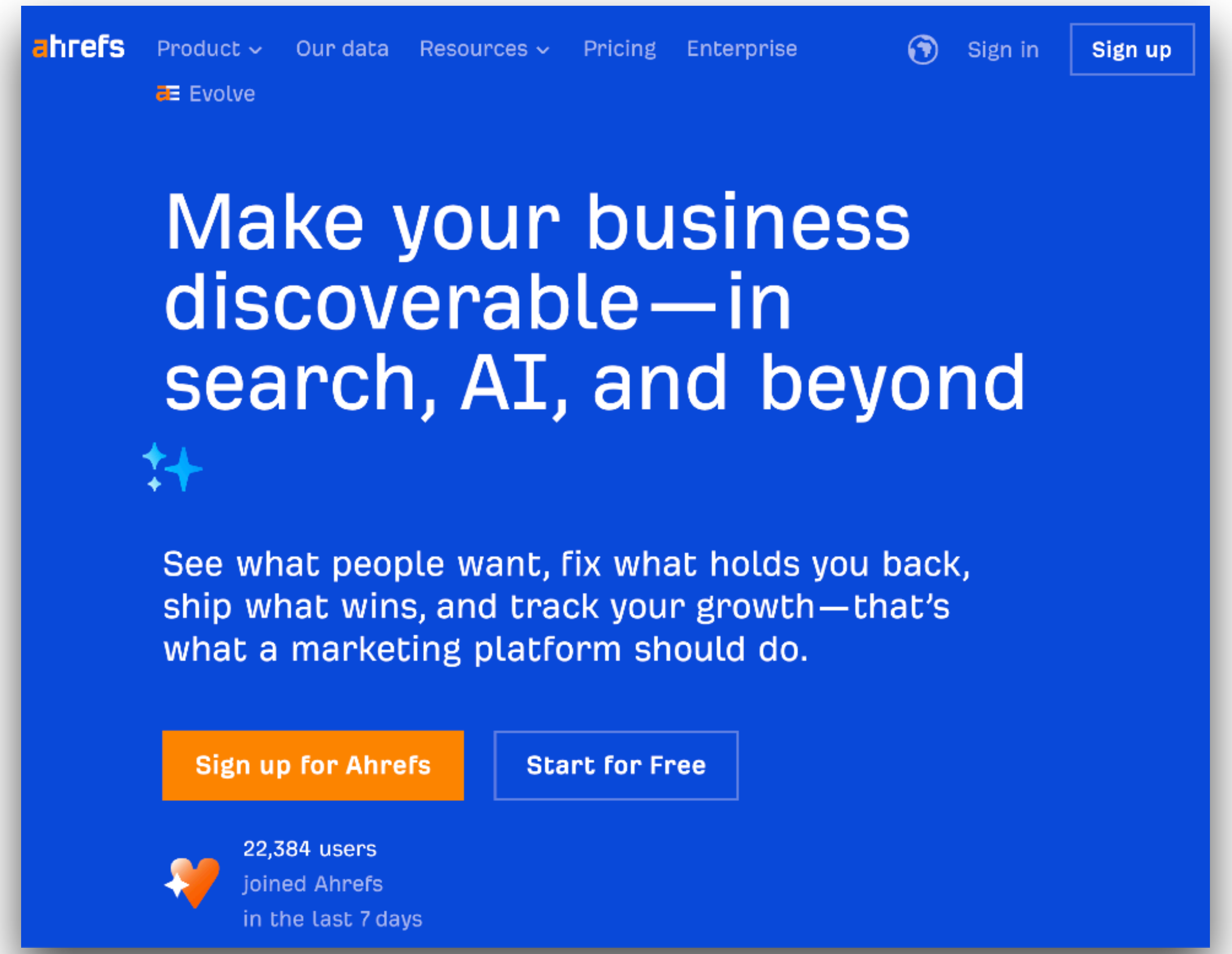




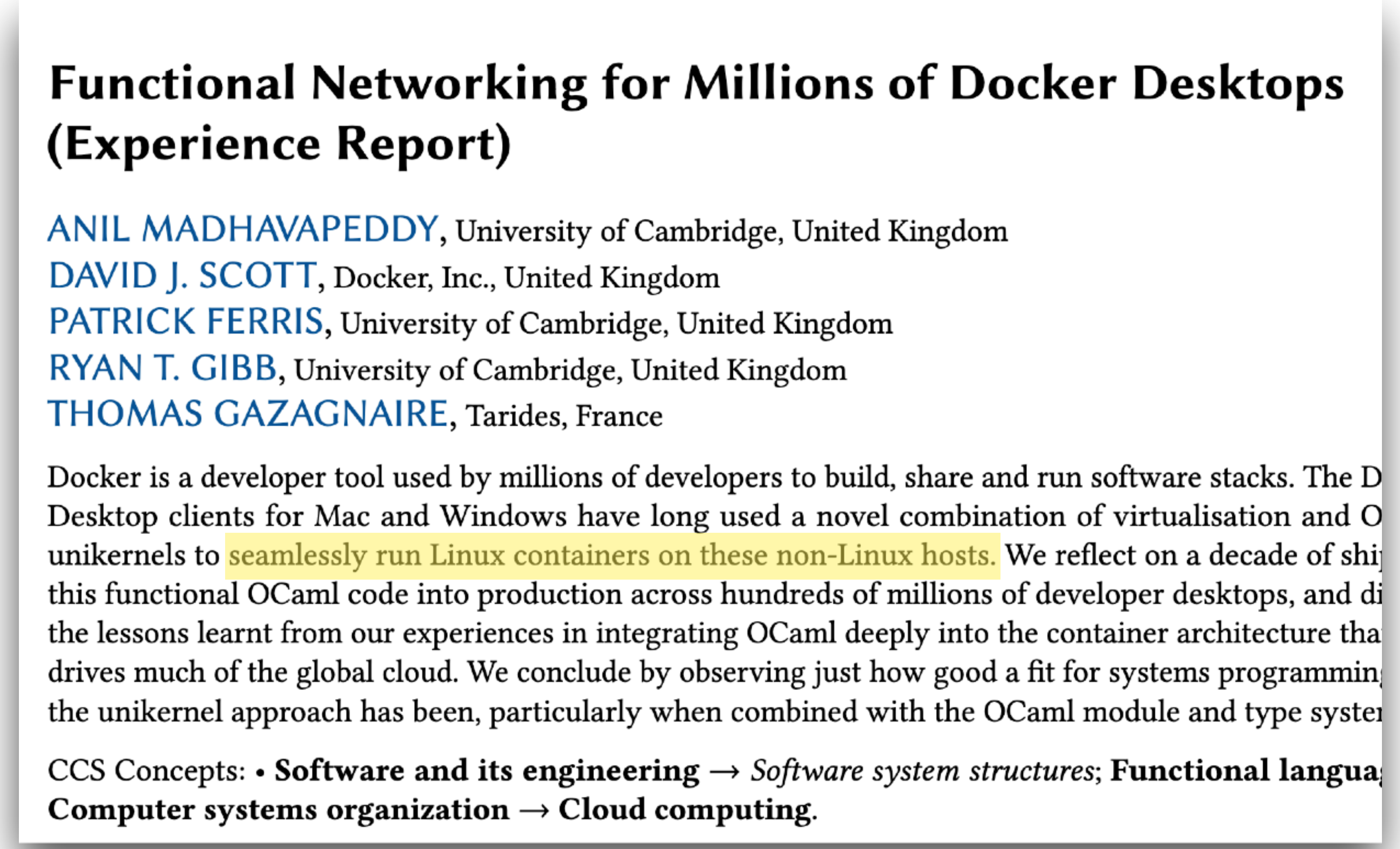
Hardware design

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure



Web Frontend



Functional Networking for Millions of Docker Desktops (Experience Report)

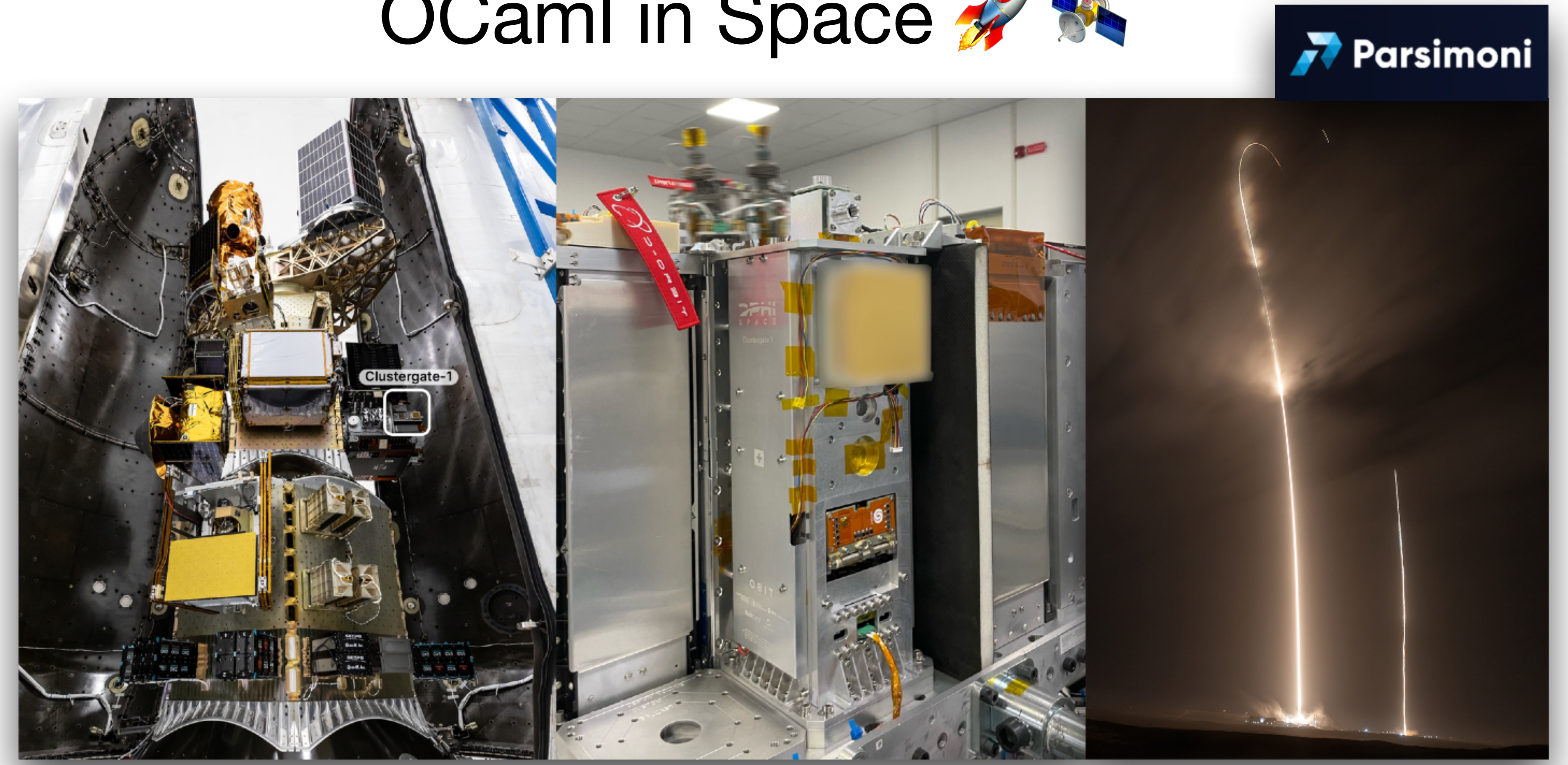
ANIL MADHAVAPEDDY, University of Cambridge, United Kingdom
DAVID J. SCOTT, Docker, Inc., United Kingdom
PATRICK FERRIS, University of Cambridge, United Kingdom
RYAN T. GIBB, University of Cambridge, United Kingdom
THOMAS GAZAGNAIRE, Tarides, France

Docker is a developer tool used by millions of developers to build, share and run software stacks. The D
Desktop clients for Mac and Windows have long used a novel combination of virtualisation and O
unikernels to seamlessly run Linux containers on these non-Linux hosts. We reflect on a decade of shi
this functional OCaml code into production across hundreds of millions of developer desktops, and di
the lessons learnt from our experiences in integrating OCaml deeply into the container architecture tha
drives much of the global cloud. We conclude by observing just how good a fit for systems programmin
the unikernel approach has been, particularly when combined with the OCaml module and type syste

CCS Concepts: • **Software and its engineering** → *Software system structures*; **Functional langua**
Computer systems organization → **Cloud computing**.

Networking

OCaml in Space 🚀 🛰️



Virtualisation

Trusted Computing Base

- Unsurprisingly, the OCaml compiler is written in OCaml
  - But includes ***a runtime system written in C***

```
[kc@KCs-MacBook-Pro-2 ocaml % cloc .
    6167 text files.
    4025 unique files.
    4354 files ignored.

github.com/AlDanial/cloc v 2.02  T=2.41 s (1669.2 files/s, 314367.8 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
OCaml                         2974          62693         115033         393861
C                              308           8629          10315          48137
Bourne Shell                    72           7502           8900          45563
m4                              12           1489            111          12448
Assembly                        18            561           2194           5869
C/C++ Header                    91           2034           3580           5600
make                            23            922            610           3633
Markdown                        29            940             34           2828
D                              410              0              0           2168
AsciiDoc                        12            518              0           1682
CSV                              1              0              0           1512
```

~400k lines of OCaml

~60k lines of ***C & Assembly***

All of it in the runtime system, much of it in the GC

OCaml GC

- OCaml is a garbage-collected (GCed) language.
  - Low-latency, high-throughput with good space-time tradeoff
- Memory management subsystem = Allocator + GC
- GC in OCaml 5
  - …. is a ***complex piece of software***
  - A generational, concurrent and parallel!
  - Support for weak references, finalisers (2 kinds), ephemerons, etc.
- ***Tons of bugs*** during development (see Multicore OCaml project)
  - C is excellent for writing unsafe, hard-to-reason-about code! :-(

Can we build a correct-by-construction GC for OCaml?

Verified GC desiderata

- **Correct-by-construction**
  - End-to-end proof that the GC preserves safety and liveness
- **Pluggable**
  - Should be able to ***slot in*** for the existing GC
- **Extensible**
  - Accommodate improvements without rearchitecting from scratch
- **Efficient**
  - Be competitive with existing GC

Everything?

Our work

- A verified stop-the-world ***mark-and-sweep*** GC for OCaml
- Written in F* and its subset Low*, proof-oriented programming languages
- Extracted to memory-safe C and integrated with OCaml 4.14.1 (non-multicore)
- Competitive performance with vanilla OCaml


Home › Journal of Automated Reasoning › Article

A Mechanically Verified Garbage Collector for OCaml

Open access | Published: 14 May 2025

Volume 69, article number 11, (2025) Cite this article

Download PDF

You have full access to this open access article

Sheera Shamsu, Dipesh Kafle, Dhruv Maroo, Kartik Nagar, Karthikeyan Bhargavan & KC Sivaramakrishnan

997 Accesses Explore all metrics →

Abstract

The OCaml programming language finds application across diverse domains, including systems programming, web development, scientific computing, formal verification, and symbolic mathematics. OCaml is a memory-safe programming language that uses a

Journal of Automated Reasoning, 2025

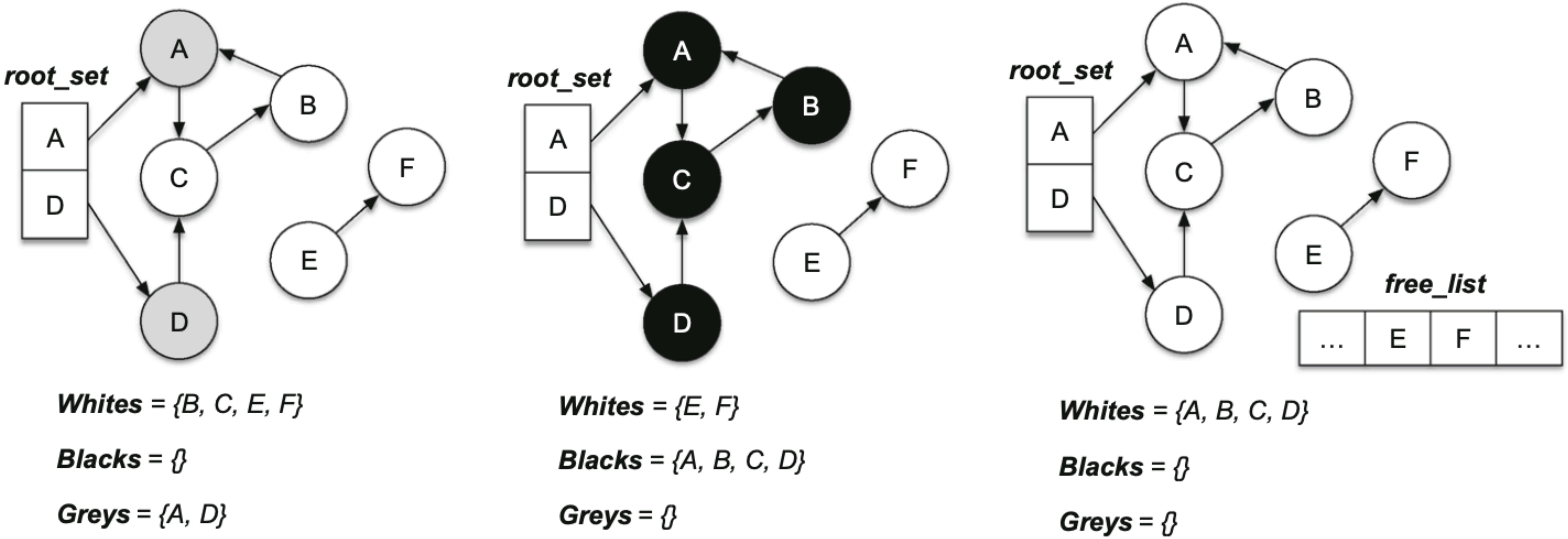
GC Correctness

- Correctness properties
  - Safety — GC preserves all ***reachable*** objects
  - Liveness — GC frees all ***unreachable*** objects
- How do we approach this?
  - Reachability is a property of the ***graph*** formed by objects in the heap.
  - Reason about correctness by connecting ***reachability*** to the ***GC operations***

GC verification — Layered approach

Tricolour Mark-and-sweep GC

White = Unmarked, Grey = Marking, Black = Marked



(a) Start of mark (b) After mark (c) After sweep

Layer 1 — Graph and reachability

```
noeq type graph (#a:eqtype) = {
  vertices : v: vertex_set #a;
  (* [vertices] are a sequence of type a with no duplicates *)
  edges : e: edge_set #a {edge_ends_are_vertices vertices e}
  (* [edges] are a sequence of type (a,a) with no duplicates *)
}

type reach: (g:graph) → (x:vertex) → (y:vertex) → Type =
  (* reachability is reflexive *)
  | ReachRefl  : (g:graph) → (x:vertex) → (reach g x x)
  (* reachability is transitive *)
  | ReachTrans : (g:graph) → (x:vertex) → (z:vertex) →
                 (reach g x z) →
                 (* [edge g z y] is a type refinement which mandates
                    that [(z,y)] is an edge in [g] *)
                 (y:vertex {edge g z y}) → (reach g x y)
```

Layer 1 — Functional Depth-first search

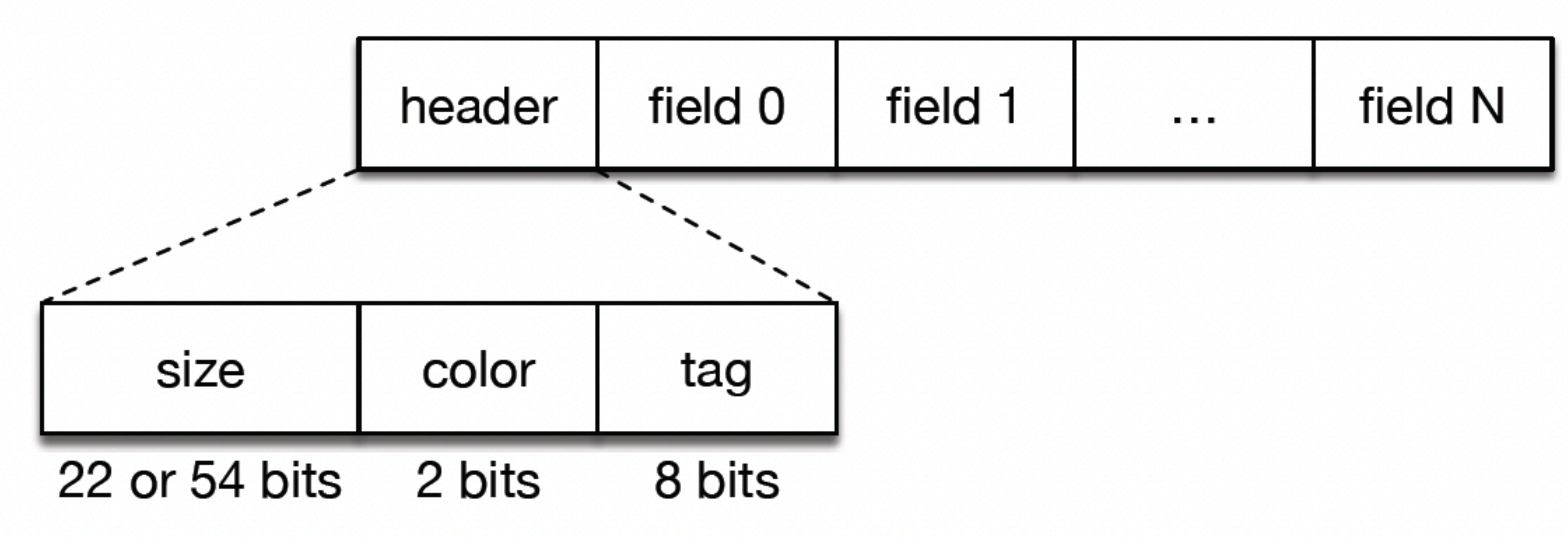
```
(* dfs calls dfs_body until stack empty.
   Inputs are graph, stack and visited set.  *)
let rec dfs (g:graph) (st:seq U64.t) (vl:seq U64.t)
 : Pure (seq U64.t)
  (requires ...)
  (ensures (λ res → ...)
  (decreases (length g.vertices - length vl; length st)) =
  if length st = 0 then vl
  else
    let st₁ ,vl₁ = dfs_body g st vl in
    dfs g st₁ vl₁

let dfs_body g st vl
 : Pure ...
  (requires ...)
  (ensures (λ res → ...) =
     let x   = stack_top st in
     let xs  = stack_rest st in
     let s   = successors g x in
     let vl₁ = set_insert x vl in
     let st₁ = push_unvisited s xs vl1 in
     (st₁, vl₁)
```

Layer 1 — Reachability ≡ DFS

```
(* r_list is the root set, stack is filled with r_list initially *)
val dfs_reachability_lemma (g:graph) (st:seq obj_addr)
                           (vl:seq obj_addr) (r_list:seq obj_addr)
  : Lemma
   (requires
     (* Pre-conditions required to prove forward direction *)
     (*F1*) mutually_exclusive_sets st vl ∧
     (*F2*) (∀ y.y ∈ st ⟹ (∃ x.x ∈ r_list ∧ reach g x y) ∧
     (*F3*) (∀ y.y ∈ vl  ⟹ (∃ x.x ∈ r_list ∧ reach g x y) ∧

     (* Pre-conditions required to show the backward direction *)
     (*B1*) (∀ x y.x ∈ r_list ∧ reach g x y  ⟹
            (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)
     (*B2*) (∀ x y.x ∈ vl ∧ reach g x y  ⟹
            (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)

   (ensures (∀ y.y ∈ (dfs g st vl)  ⟺ (∃ x.x ∈ r_list ∧ reach g x y))
```

Layer 2 — Heap

- A single contiguous buffer packed with objects
- Objects follow OCaml object layout

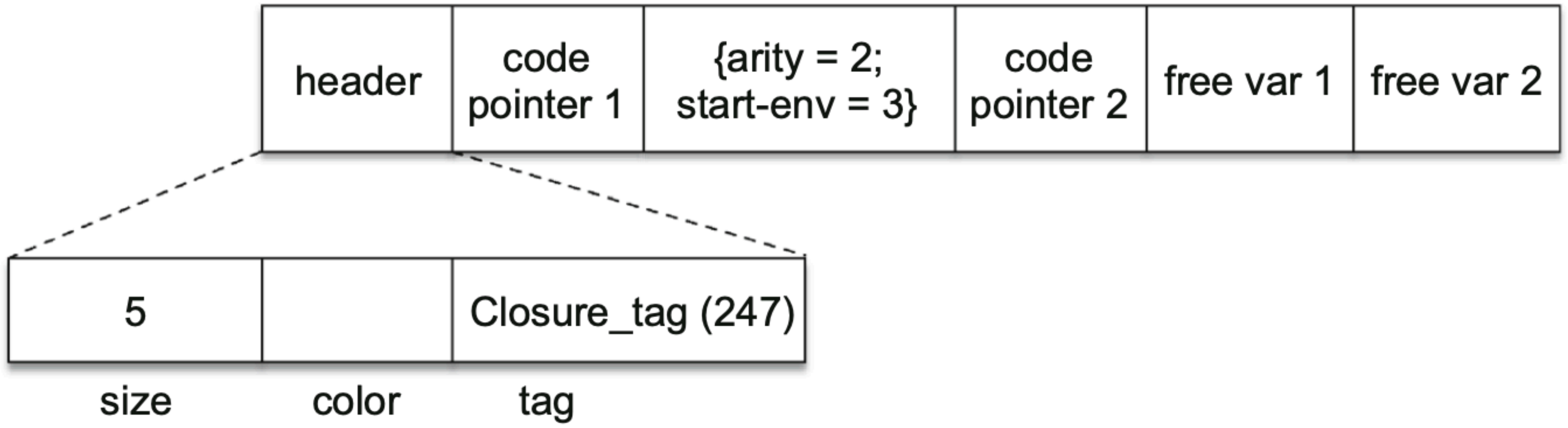



```
00 — White — Unmarked
01 — Gray  — Marking
10 — Blue  — Free
11 — Black — Marked
```

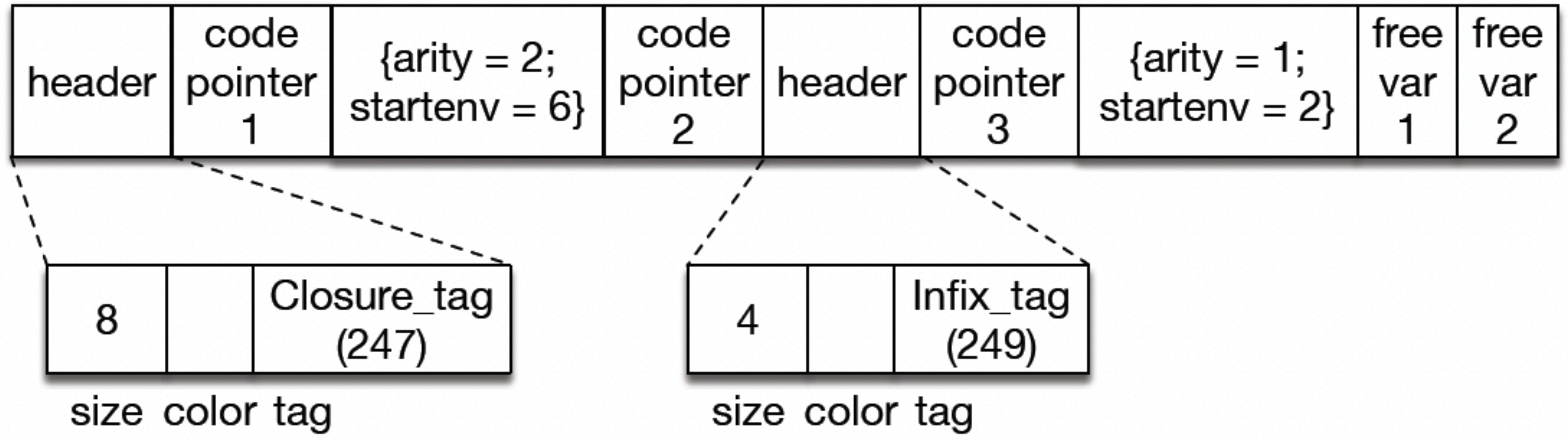
- Uniform value representation — Tagged ints
  - Makes it easy for GC to scan the object

Layer 2 — Heap

- Special Objects
  - NO SCAN objects — Strings, Float Arrays, Abstract, etc; contain no pointers.
  - Closure Objects



  - Mutually-recursive closure objects



Layer 2 — Well-formed heap $\omega(h)$

- A well-formed heap satisfies a few properties.
  - Objects are non-overlapping
  - Objects are within the heap
  - Objects fill the heap
  - Pointers in the objects point to other allocated objects (non-blue)
  - Objects satisfy their layout requirements (think Closure and Infix objects).

Layer 2 — We're still functional!

```
//Machine integers
module U64 = FStar.UInt64
module U8 = FStar.UInt8
let mword = 8UL
val heap_size : n:int{n `mod` U64.v mword == 0 ∧ n >=16 ∧
                      n < 1099511627776}
(* heap is a sequence of 8 bit unsigned machine integers *)
type heap = h:seq U8.t{length h == heap_size}
(* A valid heap address *)
type hp_addr = addr:U64.t {U64.v addr < heap_size  ∧
                           is_multiple_of_mword addr}
```

Layer 2 — DFS and Mark

```
(* dfs calls dfs_body until stack empty.
   Inputs are graph, stack and visited set. *)
let rec dfs (g:graph) (st:seq U64.t) (vl:seq U64.t)
 : Pure (seq U64.t)
  (requires ...)
  (ensures (λ res → ...)
  (decreases (length g.vertices - length vl; length st)) =
  if length st = 0 then vl
  else
    let st₁ ,vl₁ = dfs_body g st vl in
    dfs g st₁ vl₁

let dfs_body g st vl
 : Pure ...
  (requires ...)
  (ensures (λ res → ...) =
     let x   = stack_top st in
     let xs  = stack_rest st in
     let s   = successors g x in
     let vl₁ = set_insert x vl in
     let st₁ = push_unvisited s xs vl1 in
     (st₁, vl₁)
```

```
(* mark calls mark body until stack empty.
   Inputs are heap and stack *)
let rec mark (h:heap) (st:seq obj_addr)
 : Pure (heap)
  (requires ...)
  (ensures (λ res → ...)
  (decreases (length allocs h - length blacks h;
              length st)) =
  if length st = 0 then h
  else
    let st₁, h₁ = mark_body h st in
    mark h₁ st₁

let mark_body (h:heap) (st:seq obj_addr)
  : Pure ...
    (requires ...)
    (ensures (λ res → ...) =
  let x    = stack_top st in
  let xs   = stack_rest st in
  let h₁   = colorHeader h x black in
  let st1  = darken h₁ xs x 1UL in
  (st₁, h₁)
```

Layer 2 — Mark specification

```
val mark (h:heap) (st:seq obj_addr)
  : Pure (heap)
  (requires (* Only core conditions shown *)
  (*1*) well_formed_heap (h) ∧
  (*2*) (∀ x.x ∈ st ⟺ hd_address x ∈ greys(h)))
  (ensures  (* Only core conditions shown *)
  (*1*) (λ h₁ → well_formed_heap (h₁) ∧
  (*2*) (∀ x i. (hd_address x) ∈ h_objs(h) ⟹
                      field x h i = field x h₁ i) ∧
  (*3*) (∀ x.x ∈ h_objs(h₁) ⟹ (color (hd_address x h)₁ ≠ grey)))
```

Relating Layer 2 and Layer 1 — Graph(Heap)

Heap h

Object graph, $G(h) = (V, E)$

$V =$ Set of allocated object addresses in h

$E =$ Set of pairs of the form (x, y), where
$x,\ y \in V$
and y is a pointer field of x

Layer 2 — DFS ≡ Mark

```
val dfs_mark_equivalence_lemma (h:heap) (st:seq obj_addr)
                               (vl:seq obj_addr) (h_list:seq obj_addr)
      : Lemma
        (* Only important properties shown *)
        (requires (*1*) mutually_exclusive_sets st vl ∧
                  (*2*) well_formed_heap(h) ∧
                  (* stack invariant *)
                  (*3*) (∀ x.x ∈ st  ⟺  (hd_address x) ∈ greys(h))
                  (* visited-list invariant *)
                  (*4*) (∀ x.x ∈ vl  ⟺  (hd_address x) ∈ blacks(h))

        (ensures (∀ x. x ∈ (dfs (graph_from_heap h) st vl) ⟺
                       (hd_address x) ∈ blacks(mark h st)))
```

- Proof strategy
  - At every recursive call to dfs and mark, the stack ***st*** remains the same
  - Both terminate when the stack ***st*** is empty

Layer 2 — Reachability ≡ Mark

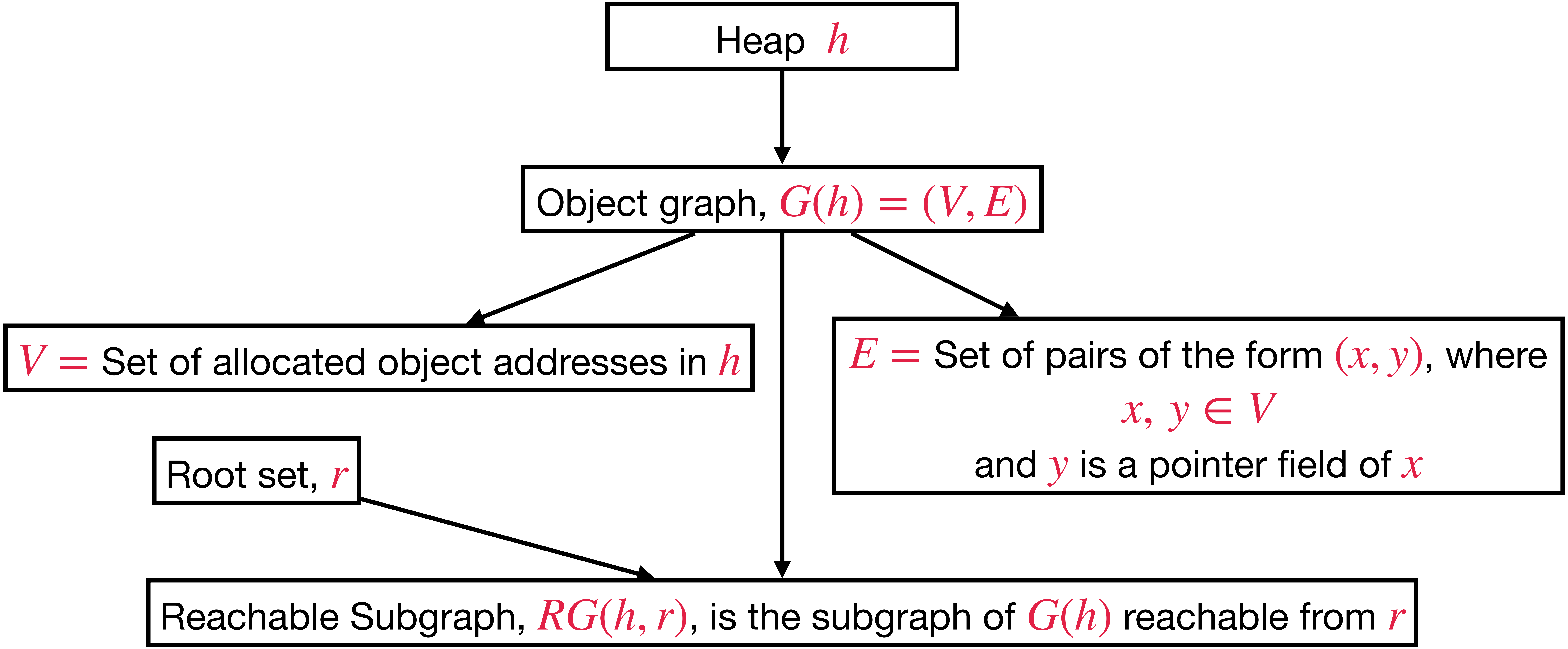
```
val mark_reachability_lemma (h:heap) (st:seq obj_addr)
                            (r_list:seq obj_addr)
  : Lemma
    (requires
    (*1*) well_formed_heap (h)
    (*2*) (∀ x. x ∈ st  ⟺  hd_address x ∈ greys(h)) ∧
    (*3*) well_formed_heap (mark h st))

    (ensures
    (*1*) (graph_from_heap (mark h st) = graph_from_heap h) ∧
    (*2*) (∀ x y.y ∈ r_list ∧
              reach (graph_from_heap h) y x  ⟺  x ∈ blacks(mark h st)))
```

- Already proved Reachability ≡ DFS and DFS ≡ Mark
  - Hence, Reachability ≡ Mark

Layer 2 — Sweep specification

```
val sweep_subgraph_lemma (h:heap) (r_list:seq obj_addr)
                         (curr_ptr:obj_addr) (fp:obj_addr)
  : Lemma
    (requires
    (*1*) well_formed_heap (h) ∧
    (*2*) (∀ x.x ∈ h_objs(h)  ⟹  (color (hd_address x h) ≠ grey)) ∧
    (*3*) well_formed_heap (sweep h curr_ptr fp))

    (ensures
    (*1*) (∀ x. x ∈ graph_from_heap (sweep h curr_ptr fp).vertices  ⟺
                x ∈ graph_from_heap (h).vertices ∧
                (∃ y. y ∈ r_list ∧ reach (graph_from_heap h) y x)) ∧

    (*2*) (∀ x y.
            (x,y) ∈ graph_from_heap (sweep h curr_ptr fp).edges  ⟺
                x ∈ graph_from_heap (sweep h curr_ptr fp).vertices ∧
                y ∈ graph_from_heap (sweep h curr_ptr fp).vertices ∧
                (x,y) ∈ graph_from_heap (h).edges))
```

Relating Layer 2 and Layer 1 — Graph(Heap)



GC Correctness

A well-formed heap $\omega(h_1)$

$\omega(h_2)$

$G(h_2) = RG(h_1, r)$
where r is the root-set

$$\text{Initial heap, } h_1 \xrightarrow{\text{GC}} \text{Final heap, } h_2$$

$\forall x\,.\, x \in G(h_2) \implies data(x, h_1) = data(x, h_2)$
where $data$ denotes the non-pointer fields

GC Correctness

```
val end_to_end_correctenss_theorem
        (* Initial heap *)
        (h_init:heap{well_formed_heap h_init})
        (* mark stack - contains grey objects *)
        (st: seq Usize.t {pre_conditions_on_stack h_init st })
        (* root set *)
        (roots : seq Usize.t{pre_conditions_on_root_set h_init roots})
        (* free list pointer *)
        (fp:hp_addr{pre_conditions_on_free_list h_init fp})
: Lemma
( requires
  (* Pre-conditions elided for brevity. Important ones are:
     + The mark stack [st] contains all the [roots].
     + All the grey objects in the heap are in the mark stack [st].
   *) )
( ensures
   (* heap after mark *)
   let h_mark = mark h_init st in
   (* heap after sweep *)
   let h_sweep = fst (sweep h_mark mword fp) in
   (* graph at init *)
   let g_init = graph_from_heap h_init in
   (* graph after sweep *)
   let g_sweep = graph_from_heap h_sweep in

(* GC preserves well-formedness of the heap *)
(* 1 *) well_formed_heap h_sweep ∧

(* GC preserves reachable object set *)
(* 2 *) ((∀ x. x ∈ g_sweep.vertices ⟺
           (∃ o. mem o roots ∧ reach g_init o x))) ∧

(* GC preserves pointers between objects *)
(* 3 *) ((∀ x. mem x (g_sweep.vertices) ⟹
           (successors g_init x) ==
           (successors g_sweep x))) ∧

(* The resultant heap objects are either white or blue only *)
(* 4 *) (∀ x. mem x (h_objs h_sweep) ⟹
           color x h_sweep == white ∨
           color x h_sweep == blue) ∧

(* No object field (either pointer or immediate) is modified *)
(* 5 *) field_reads_equal h_init h_sweep )
```

Layer 3 — Imperative GC

- Layer 3 is in Low*, a low-level explicit memory subset of F*
  - Explicitly reason about lifetime and aliasing
    - Tedious (Low* does not use separation logic) but mechanical
  - Low* programs are memory-safe by construction
  - Can be extracted to C using KaRaMel
- Our Idea is to have 1:1 correspondence between the functions in layer 2
  - Tail-recursive functions → loops
    - Invariants on tail-recursive functions → loop invariants
  - Mark Stack is a list → Mark stack is a fixed-sized buffer
    - Assumed to be the size of the heap (*limitation*)

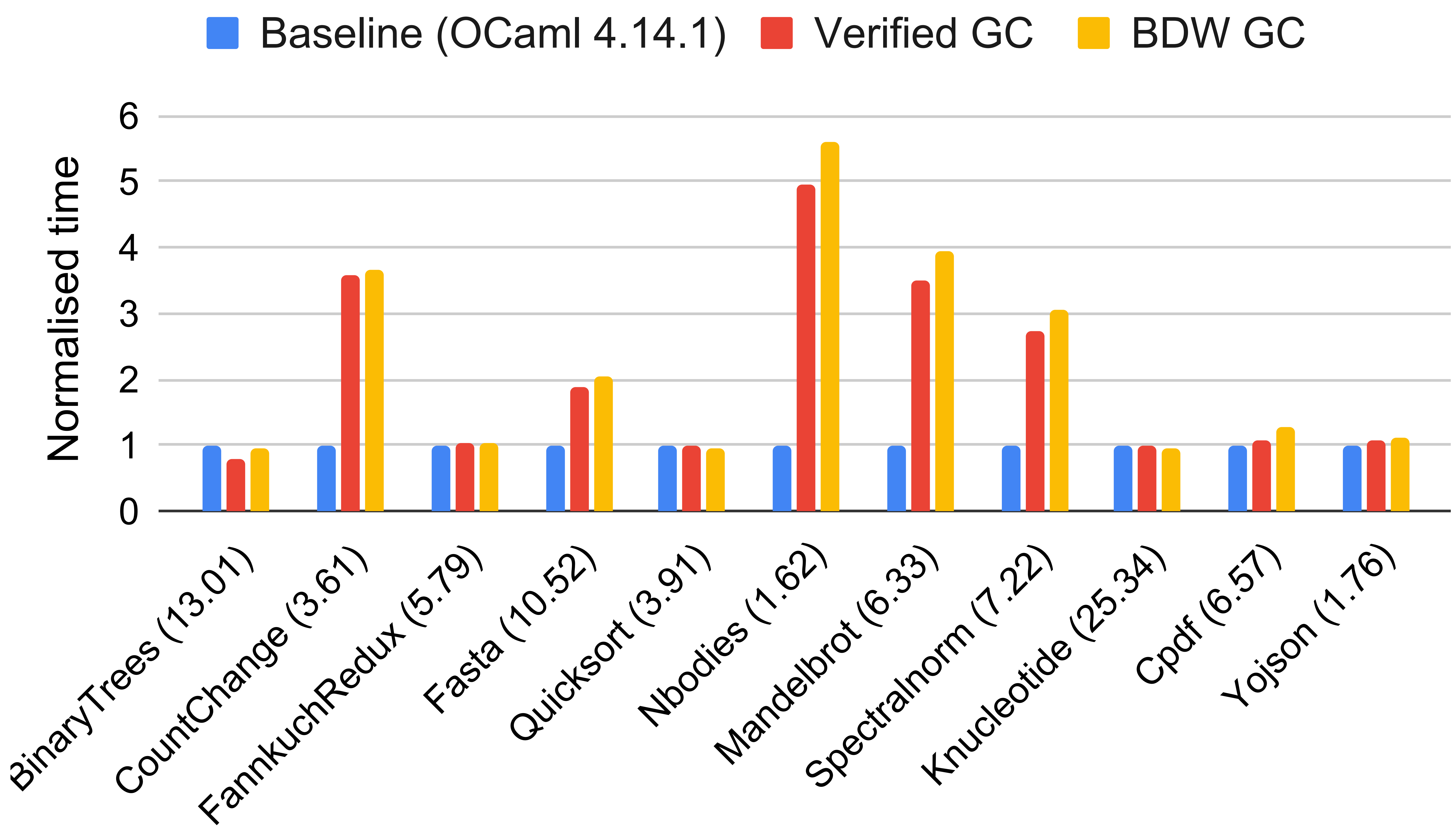
Layer 3 — Extracted Sweep

```
void sweep (uint8_t *g, uint64_t *sw, uint64_t *fp,
            uint64_t limit, uint64_t mword) {
  while (*sw < limit) {
    uint64_t curr_obj_ptr = *sw;
    uint64_t curr_header = hd_address(curr_obj_ptr);
    uint64_t wz = wosize_of_block(curr_header, g);
    uint64_t next_header =  curr_header + (wz + 1ULL) * mword;
    uint64_t next_obj_ptr = next_header + mword;
    sweep_body (g, sw, fp);
    sw[0U] = next_obj_ptr;
  }
}

void sweep_body (uint8_t *g, uint64_t *sw, uint64_t *fp) {
  uint64_t curr_obj_ptr = *sw;
  uint64_t curr_header = hd_address(curr_obj_ptr);
  uint64_t c = color_of_block(curr_header, g);
  uint64_t wz = wosize_of_block(curr_header, g);

  if (c == white || c == blue) {
    colorHeader(g, curr_header, blue);
    uint64_t fp_val = *fp;
    uint32_t x1 = fp_val;
    store64_le(g + x1, curr_obj_ptr);
    fp[0U] = curr_obj_ptr;
  } else {
    colorHeader(g, curr_header, white);
  }
}
```

Verification effort

Modules	#Lines	#Defns	#Lemmas	Time
Graph	4653	72	81	2m3s
DFS	657	1	9	2m5s
Functional GC	18401	65	218	120m
Imperative GC	2734	19	19	27m43s

Performance



Extensions

- Extended to support coalescing during sweep
  - No two adjacent free/blue objects after sweep
  - Removes fragmentation
- In the paper,
  - Specification sketches for incremental and copying collectors
  - Not implemented fully

Limitations and Future Work

- Mark stack is assumed to be the size of the heap
  - Mark stack overflow handling is tricky!
- Allocator is still unverified
- No support for weak references, ephemerons, finalisers
  - Even specifying their correctness is challenging
- OCaml 5 is multicore but the verified GC is sequential only
  - Need a framework with concurrent separation logic (?)
- Although specifications are extensible, proof burden still very large
  - Transplant ideas to reusable GC framework such as MMTk (?)

Summary

- A verified stop-the-world ***mark-and-sweep*** GC for OCaml
- Written in F* and its subset Low*, proof-oriented programming languages
- Extracted to memory-safe C and integrated with OCaml 4.14.1 (non-multicore)
- Competitive performance with vanilla OCaml
- All the artifacts have been open-sourced

https://github.com/prismlab/verified_ocaml_gc

Home > Journal of Automated Reasoning > Article

A Mechanically Verified Garbage Collector for OCaml

Open access | Published: 14 May 2025

Volume 69, article number 11, (2025) Cite this article

Download PDF You have full access to this open access article

Sheera Shamsu, Dipesh Kafle, Dhruv Maroo, Kartik Nagar, Karthikeyan Bhargavan & KC Sivaramakrishnan

997 Accesses Explore all metrics →

Abstract

The OCaml programming language finds application across diverse domains, including systems programming, web development, scientific computing, formal verification, and symbolic mathematics. OCaml is a memory-safe programming language that uses a

Journal of Automated Reasoning, 2025