A “caller-roots” calling convention for the OCaml foreign function interface

Guillaume Munch-Maccagnoni

Inria

June 1st 2022

OCamlRust:

- Gabriel Scherer (Inria)
- Bruno Deferrari (SimpleStaking)
- Jacques-Henri Jourdan (CNRS)
- Myself

Later works:

- Gabriel Scherer
- Myself

Interfacing Rust and OCaml

- Strongly typed (memory safety)
- Low-level, system
- Different resource-management mechanisms
- Building upon existing OCaml-C interface

Earlier works:

- Rust & GC: Manish Goregaokar, Alan Jeffrey (Josephine)
- OCaml & Rust: Stephen Dolan, Frédéric Bour, Zach Shipko

1: Static assurances

2: Language abstractions

3: Computational behaviour

1: Static assurances

2: Language abstractions

3: Computational behaviour

- 2: Manipulate OCaml values from Rust?
- 1: Respect OCaml GC discipline (values move)
- 3: Competitive performance?

OCaml:

- Uniform memory representation allowing polymorphism
- Memory handled with a generational tracing GC

Rust (C++ model):

- Low-level memory representation that can accomodate OCaml value representation
- Destructors called in timely fashion
- Well-suited for interoperability

Different memory management methods:

- Freeing memory: graph traversal to find live values
- Freeing memory: graph traversal to find dead values

Rust: pointer manipulation, imperative, concurrent.

Example: Dynamic arrays (Vectors) & iterator invalidation

```
fn main() {
    let mut vec = vec!['a', 'b', 'c'];
    let first = &mut vec[0];
    vec.push('e');
    let mut _vec2 = vec!['f', 'g', 'h'];
    println!("vec[0] contains {}", first);
}
```

```
error[E0499]: cannot borrow ‘vec‘ as mutable more than once at a time
 --> destructeur10.rs:6:5
  |
5 |     let first = &mut vec[0];
  |                      --- first mutable borrow occurs here
6 |     vec.push('e');
  |     ^^^ second mutable borrow occurs here
7 |     let mut _vec2 = vec!['f', 'g', 'h'];
8 |     println!("vec[0] contains {}", first);
  |                                    ----- first borrow later used here
```

- Either several concurrent reads or one concurrent write
- iterator invalidation ~ data race

What else moves values?

What else moves values?

Vector	GC heap
Vec	Runtime capability
iterator	pointer to the heap
insert element	allocate
index	root

cf. works by Goregaokar, Jeffrey, Dolan…

- Root is usually a low-level notion (GC implementation detail)

Callee-roots

OCaml C API: the callee registers values as root

```
value f(value x)
{
  CAMLparam(x);
  CAMLlocal(y);
  ...
  CAMLreturn(z);
}
```

Implementation: stack-allocated linked list

Callee-roots

Callee-roots in Rust

- Previous works: Dolan, Shipko
- Get rid of Rust macro tricks which were hard to manage about

```
pub fn f<'a, 'id>(gc: RuntimeAndValues<'a, 'id>,
                  x: ValueArg<'id>) -> Value<'a> {
    let gc_immut : &Runtime = gc.immut_cap();
    let x : Value = gc.extract_temp_value(&x);
    ...
    let gc : &'a mut Runtime = gc.into_cap();
    ...
}
```

Callee-roots

- Making sense of OCaml's FFI discipline (type safety for CAMLparam/CAMLlocal)
- Heavy to manipulate in practice (boilerplate erased at compilation)
- Could be used to provide a thin layer above the OCaml FFI

Caller-roots

Alternative: the caller is responsible for registering values as roots

```
value * f(value *x)
{
  value *y = root_create(...);
  ... *x, *y ...
  return z;
}
```

e.g. Bour's CAMLroot

- Easier to write buggy programs in C

Caller-roots

But fits safely with Rust type system

```
pub fn f(gc: &'a mut runtime, x: ValueRef<'b>) ->
  ...
}
```

- ValueRef: a polymorphic container (rooted, unrooted or immediate)
- root_create?

Boxroot

- Language abstraction: GC root as a smart pointer (cf. Rust's Box), derefs into a ValueRef
- Computational behaviour: takes advantage of good cache locality, lock-free deallocation
- Typing: supports a safe caller-roots calling convention

Boxroot

Inspired by standard, modern concurrent allocator technology.
Hooks into the OCaml runtime.

- Pools of ~512 allocatable roots
- Domain-local caches of pools
- Easy to scan during GC

Boxroot

Generational optimisation

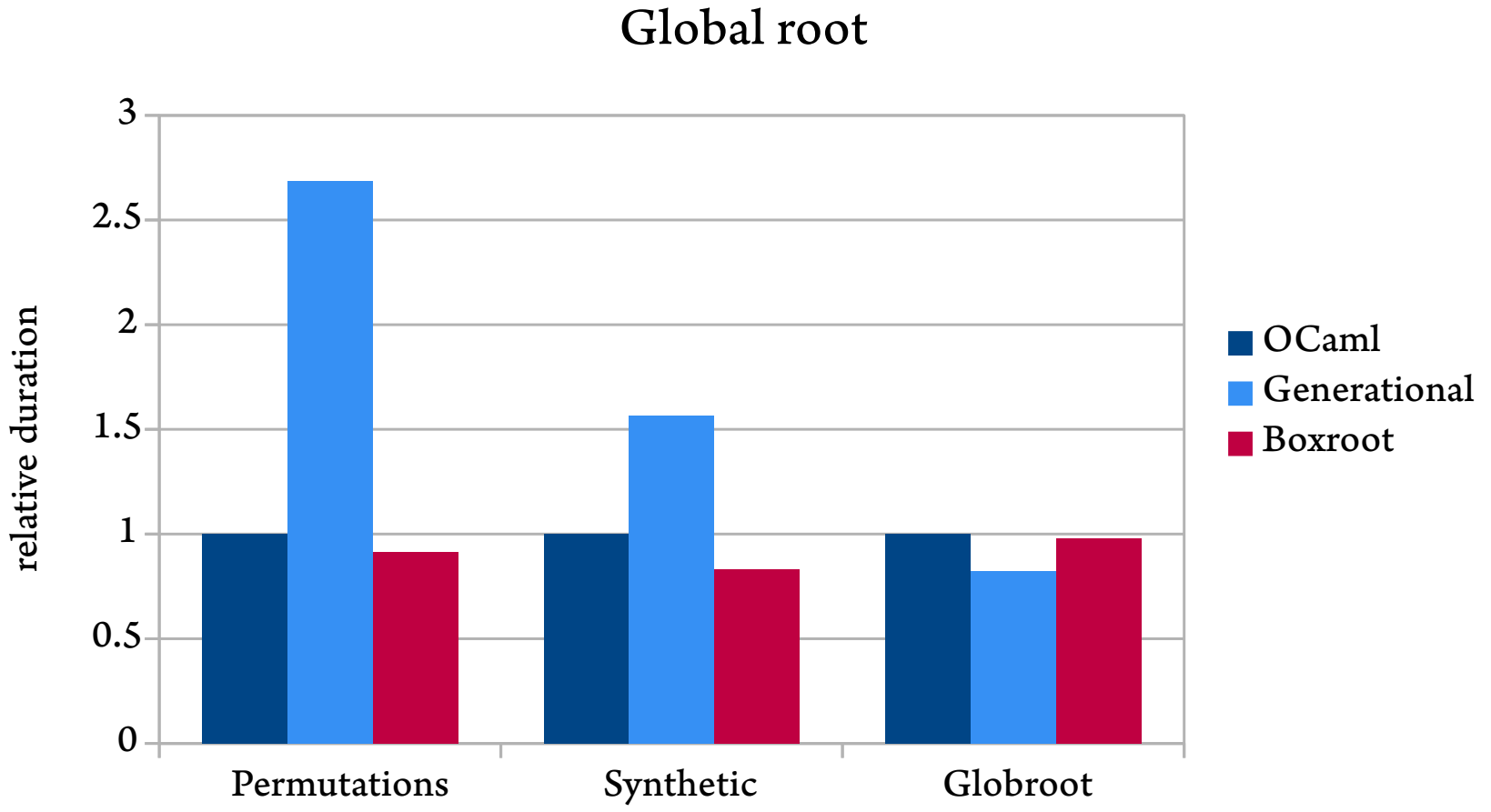
- Only scan “young” roots during minor collection
- Lesson: the simplest solution wins (inlining)
- In the end, there is nothing more in boxroot than an efficient allocator

Benchmarks (qualitative)

- OCaml 4.12
- Qualitative (micro-benchmarks, artificial situations)
- Limitation: Single-threaded
- *OCaml*: Pure OCaml implementation
- *Boxroot*: via C, using boxroot
- *Generational*: via C, using OCaml API generational global roots
- *Local*: via C, using OCaml API local roots (CAMLparam, etc.)

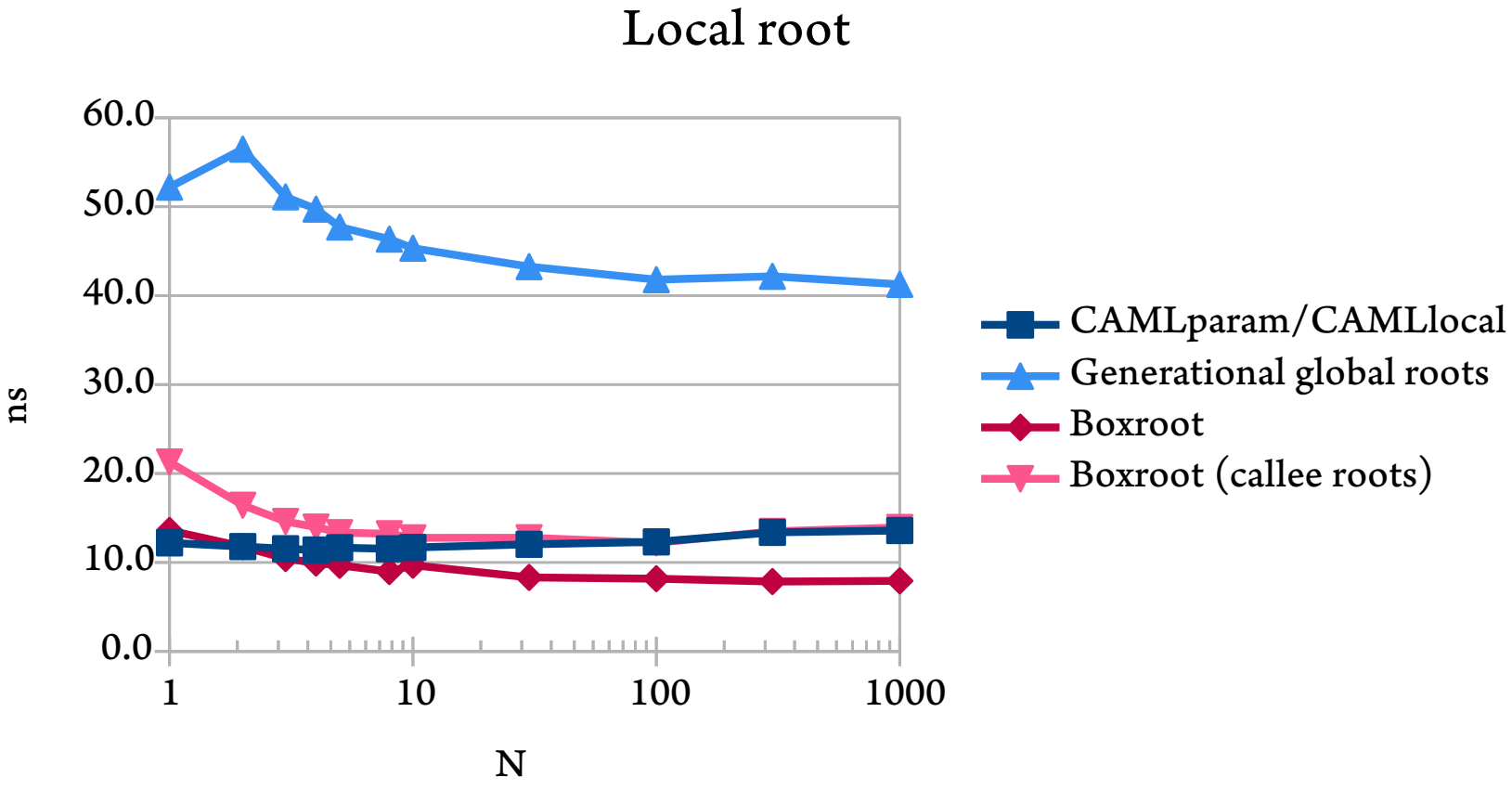
Boxroot vs. Global roots

Let's do various computations using an OCaml value stored inside an OCaml or a foreign value. If inside a foreign value, we need to register it as a root.



Boxroot vs. Local roots

Let's simulate an OCaml->C function call which does N function calls. Each function call manipulates the OCaml heap and so it needs to register the local variables as roots.



Conclusion

Academic

- Ideas can be re-used for other languages
- We discovered artificial limitations of Rust's type system
- Future: Contributes to merging functional and systems programming

Practical

- Some ideas have been put into practice in ocaml-rs, which uses Boxroot.
- Future of the OCaml/Rust interface: additional and sustained maintainer work needed.

Conclusion

Thank you

References I

Alan Jeffrey. 2018. Josephine: Using JavaScript to safely manage the lifetimes of Rust data. (2018). arXiv:cs.PL/1807.00067