Resource polymorphism : A proposal for integrating first-class resources into ML

Guillaume Munch-Maccagnoni

Inria

ML workshop, September 8th 2023

Resource management

Values with a clean-up operation

```
let read_line name =
  let f = open_in name in
  print_endline (input_line f);
  flush stdout;
  close_in f
```

Examples in Multicore OCaml: continuations/fibers, resources in scheduling frameworks. Discussions with industrial users of OCaml about problems with resource leaks.

Resource management

Even in case of errors

```
let read_line name =
  let f = open_in name in
  try
    print_endline (input_line f);
    flush stdout;
    close_in f
  with e ->
    close_in_noerr f;
    raise e
```

(example from Real World OCaml, Minsky et al., 2013)

Resource management

Resource: value which is hard to copy or dispose of.

- large or shared data structures
  (⇒ memory management)
- low-level abstractions
- anything that needs to be cleaned-up (file handle, sockets, locks, values from a foreign runtime...)
- anything for which aliasing is harmful
- emerging abstractions (e.g. poison guards for fault-tolerance in Rust)
- Programming with resources: any data structure containing the above (lists of resources, closures of resources...)

Resource management

- Memory management (when avoiding a GC)
- Usage of a value respects a protocol (e.g. file, network connection, value from a foreign runtime)
- Fault tolerance (correct exception handling)

$\Rightarrow$ Correctness, efficiency, interoperability, expressiveness

Resource management

What Techniques to reason about the state of the program

How Language support for linear values (**linearity as a safety property**)

Resource Management in C++11/Rust

C++11/Rust represent a breakthrough for all these questions

- C++11: “move semantics”: resources as first-class values in C++. A notion of ownership based on linear values becomes core to the language. Connections with linear logic anticipated by Baker (1994a, 1995).
- Rust: extends the C++11 model with fancy types to avoid certain classes of bugs at compilation, shows it relevance for concurrent/parallel programming.

Resource Management in C++11/Rust

Key concept: Destructors (aka RAII: “Resource acquisition is initialization”)

- Function that is called when a variable goes out of scope (*predictably*)
- Including when an exception is raised (*stack unwinding*)

cf. unwind-protect

But:

- A type-level abstraction: algebra of types
- Move semantics (since C++11): passing and returning a resource along with the responsibility of calling its destructor (*ownership*); data types (e.g. vectors) can now manage resources.

Resource Management in C++11/Rust

Linearity and ownership are *emergent phenomena* of types with destructors.
In Rust, other notions follow intuitively:

1. Region typing ("borrowing"),
2. Uniqueness ("linear/mutable borrows"),
3. External uniqueness/linear abstract data types ("interior mutability").

A semantic reconstruction of RAII

jww G. Combette *"A resource modality for RAII"* (LOLA 2018)
Understanding RAII/destructors via Curry-Howard.

- Interpret types with destructors as objects $(A, \delta : A \to TI)$ of the slice category $C_{/TI}$
- The exists a resource modality (like linear logic "!") for types with destructors
- Unlike "!" (which enriches linear logic with a kind for intuitionistic logic), it enriches intuitionistic logic with a kind for *ordered logic* (because the order of destruction matters)

A semantic reconstruction of RAII

- *A type-based abstraction.* Attach a destructor to a type, to create a new type.
- *Ordered* data types (rather than linear or affine)

$$A \otimes B \not\cong B \otimes A$$

- Still affine at the level of provability! (move is an effect)

$$A \otimes B \leftrightarrow B \otimes A$$

- Solves the open question of combining linearity and control effects (e.g. exceptions)

A semantic reconstruction of RAII

“Are types in Rust linear or affine?”

Our model is clear:

- *Linear* for values
- *Ordered* for types
- *Affine* in terms of provability/static type system

Proposal: First-Class Resources in ML

Resource polymorphism, 2018 (CoRR)
Propositions in language design and implementation
See a language 3 layers:

1. Type system (*challenge, lots of relevant literature*)
2. Language abstractions (*this proposal*)
3. Runtime/implementation (*this proposal, but not this talk*)

The proposal is also meant as a survey of resource management

Proposal: First-Class Resources in ML

Goals:

- Looking for an ML-like *"sweet spot"*: between simplicity, modularity, expressiveness…
- Be rooted in programming practice and experience (the goal is *not* to turn a mathematical concept into a language feature)
- Learn from the C++11 move semantics proposal: make resources *"first-class"* (enable gradual migration, avoid creating a new dialect that mixes poorly with other code)
- Backwards-compatibility at all costs including performance (e.g. do not change how preexisting code has to be compiled)

⇒ Add one concept of resource, enriched with various notions of polymorphism pertaining to resources

Proposal: First-Class Resources in ML

New types

```
type t = Res of u with destructor f
                            (* f : u -> unit      *)
                            (* f must not raise *)
t &
```

- *Runtime* kind distinction **O**wned/**U**nrestricted, inspired by the notion of polarity in linear logic (Girard, 1991, 1993), similar to Rust's special traits (Drop/Copy)

New terms

```
&v
*x (* on bound variables *)
```

Everything else follow from there.

Proposal: First-Class Resources in ML

```
type file_in = File of Stdlib.in_channel
               with destructor Stdlib.close_in_noerr

let open_file name : file_in =
  File (Stdlib.open_in name)
```

Proposal: First-Class Resources in ML

```
let drop *x = ()
(* val drop : (^a : O) -> unit = <fun> *)

let fancy_drop *x =
  try
    let *y = x in raise Exit
  with
    Exit -> ()
```

Proposal: First-Class Resources in ML

```
let create_and_move name =
  let *x = open_file name in
  f x (* move resource *)
```

Proposal: First-Class Resources in ML

```
let twice1 name =
  let *f = open_file name in
✗ (f,f) (* expected: error: f cannot be copied *)
```

Proposal: First-Class Resources in ML

```
let open_list =
  List.map (fun name ->
              (name, open_file name))
(* (string * file_in) list : O *)
```

Proposal: First-Class Resources in ML

```
let open_list =
  List.map (fun name ->
              (name, open_file name))
(* (string * file_in) list : O *)

open_list l
(* Clean-up after Exception:
   Sys_error "No such file or directory". *)
```

Proposal: First-Class Resources in ML

Parametric resource polymorphism

```
let rec map f = function
    [] -> []
  | *a::*l -> let *r = f a in r :: map f l
(* map : (^a -> ^b) -> ^a list -> ^b list *)
```

Compiling **U** <: **O**

Monomorphisation of polarities (there are only 2 per distinct type variable)

- **U** Compiled as usual
- **O** Compiled with destructor calls & moves, receiving destructor implicitly as an argument (e.g. implicit first-class module)

Proposal: First-Class Resources in ML

Experiment: adapting OCaml Stdlib modules to respect linearity.
E.g. the List module

- It is already linear (one just needs to add * and **&** where needed, in the notations of this talk). Only the List.sort function needed to be made linear; a PR to do so has been accepted because it made it objectively better.
- One of the most polymorphic library
  - 16 functions have two parameters and would compiled up to 4 times after kind monomorphisation, e.g. List.map
  - 4 functions have three parameters and would be compiled up to 8 times, e.g. List.map2
  - No function has more than 3 parameters
  - All the possible kind combinations make sense

Proposal: First-Class Resources in ML

(Notations allude to, but the proposal leaves as an open question, an expressive kind system for affine typing)

Practical Affine Types *

Jesse A. Tov Riccardo Pucella

Northeastern University

{tov,riccardo}@ccs.neu.edu

Abstract

Alms is a general-purpose programming language that supports practical affine types. To offer the expressiveness of Girard's linear logic while keeping

Borrows

We have added alongside ML a copy of it made of linear values. How do they mix?

Borrows

&

```
let read_line name =
  let *f = open_file name in
  print_endline (input_line &f);
  flush stdout
```

Borrows

```
let read_line name =
  let *f = open_file name in
  let File g : file_in = &f in
  drop f;
  print_endline (input_line g)
    (* Sys_error "Bad file descriptor" *)
```

Borrows

Linear Abstract Data Types

```
module File : sig
  type t : O
  val open : string -> t
  val input_line : t & -> string
end

let read_line name =
  let f = File.open name in
  let g : File.t & = &f in
  drop f;
  print_endline (File.input_line g)
(* Compilation error: g outlives its resource *)
```

Borrow polymorphism

Expressiveness argument:
Assuming all resources are fixed, the expressiveness with borrows should be that of entire ML

```
filter : ('a & -> bool) -> 'a list & -> 'a list &

let x = &l in let y = filter f x in ...

(*  (string * File.t) list &
                vs.
    (string * File.t) & list
                 ?                  *)
```

Borrow polymorphism

& a homomorphism

```
  (string * File.t) list &
= (string * File.t) & list
= (string & * File.t &) list
```

(interpretation: forgetful functor from **O** to **U**)

Summary

- New types
  `type t = Foo of t with destructor f`
  `t &`
- New term formers
  `&x`
  `*x`
- *Parametric resource polymorphism* (with monomorphisation of kinds at runtime)
- *Borrow polymorphism* (**&** as a forgetful functor)
- Linear abstract data types

Not discussed here: *type-dependent polarities, linear mutable state, linear borrows, types of closures, borrow modality, linear continuations, tail calls,* and more

Further experiments

1. Implementing destructors with typed pointer reversal (jww. Rémi Douence, ML 2019)
2. Semantics of asynchronous exceptions & failure recovery (OCaml workshop 2021)

Advantages of GC and Linear Allocation

Automatic memory management with RAII (C++11/Rust)

- Stack allocation & memcpy (orthogonal to the present proposal)
- Unique pointers
  - Ownership & borrowing discipline
  - “As efficient” as raw malloc/free
- Reference-counted pointers
  - Copiable
  - Many costs & limitations (e.g. cycles)
  - Baker: minimise cost by moving, borrowing and deferred copying (more recently, see works on Perceus at this workshop)

Advantages of GC and Linear Allocation

"tracing operates on live objects, while reference counting operates on dead objects"

A Unified Theory of Garbage Collection

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Tracing and reference counting are uniformly viewed as being fundamentally different approaches to garbage collection that possess very distinct performance properties. We have implemented high-performance collectors of both types, and in the process observed that the more we optimized them, the more similarly they behaved — that they seem to share some deep structure.

We present a formulation of the two algorithms that shows that they are in fact duals of each other. Intuitively, the difference is that tracing operates on live objects, or "matter", while reference counting operates on dead objects, or "anti-matter". For every operation

1. INTRODUCTION

By 1960, the two fundamental approaches to storage reclamation, namely tracing [33] and reference counting [18] had been developed.

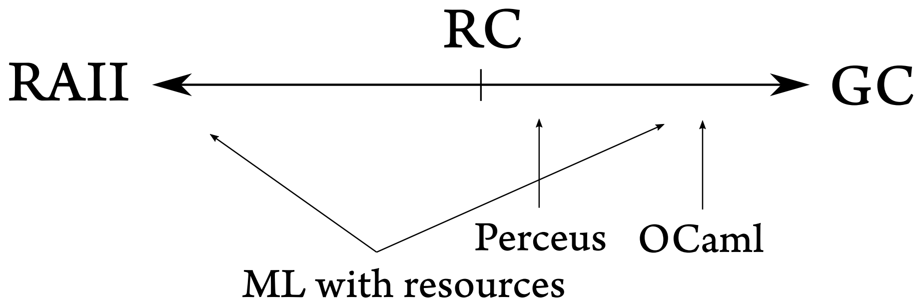
Since then there has been a great deal of work on garbage collection, with numerous advances in both paradigms. For tracing, some of the major advances have been iterative copying collection [15], generational collection [41, 1], constant-space tracing [36], barrier optimization techniques [13, 45, 46], soft real-time collection [2, 7, 8, 14, 26, 30, 44], hard real-time collection [5, 16, 23], distributed garbage collection [29], replicating copying collection [34], and

Advantages of GC and Linear Allocation

Reference-counting

- ✓ Values do not move (interoperability)
- ✓ Timely resource destruction
- ✓ Possible to re-use cells
- ✗ Costs of count updates & synchronisation
- ✗ Cycles leak
- ✗ Latency due to upfront deallocation cost, sometimes cascading
- ✗ Upfront allocation cost

Advantages of GC and Linear Allocation



Advantages of GC and Linear Allocation

Generational GC (tracing live)

- ✓ No discipline needed
  - Shared data structures & shared mutable state
  - Cycles
- ✓ Allocation almost free for short-lived values
- ✗ Cost of read/write barriers, synchronisation
- ✗ Allocation can cause latency
- ✗ Unable to deal with resources
- ✗ Values can move

Advantages of GC and Linear Allocation

Allocate with RAII, e.g. RC=1 (tracing dead)

- ✓ Timely resource destruction
- ✓ Purely static re-use of cells (Baker, 1992; Lafont, 1988)
- ✓ No reference count to update
- ✓ Values do not move (interoperability)
- ✓ No read/write barrier, no synchronisation
- ✓ No cycles
- ✓ During life: no cost & no interruption
- ✗ Upfront allocation cost (but no need to stop for GC)
- ✗ Ownership & borrowing discipline
- ✗ Latency due to upfront deallocation cost (but in a controlled way)

Advantages of GC and Linear Allocation

RAII allocation suitable for:

- large data with epochal behaviour (see Nguyen et al., 2016)
- interoperability with systems languages (efficiently and expressively)
- performance-sensitive paths
    - To avoid long-lived GC allocations
    - To avoid all allocations (e.g. pre-allocate a free list, re-use cells during hot path, and clean-up after)

(cf. ML 2020 informed position talk)

Combining GC and Linear Allocation

Language design : expressiveness vs. concision

"RAII hypothesis"

- RAII-allocated types ⊆ types with destructors (obviously)
- Anybody using destructors already pay most of the costs (ownership & borrowing discipline, traversing the whole structure on destruction: no benefit to expect from a generational hypothesis anyway)
- Heuristic: types with destructors ⊆ RAII-allocated types

Combining GC and Linear Allocation

$$\downarrow^{\mathbf{U}}_{\mathbf{O}} : \mathbf{U} \to \mathbf{O}$$

Embed GC-allocated values into resources

Register GC root; set destructor to unregister root.

Boxroot experiment: rooting is free (= not more expensive than what is already paid for) (jww G. Scherer, ML 2022)

Combining GC and Linear Allocation

$$\uparrow^{\mathbf{B}}_{\mathbf{O}} : \mathbf{O} \rightarrow \mathbf{B}$$

Embed linearly-allocated values inside GC-allocated ones

Borrowing as a forgetful functor

Implemented at runtime with:

- Uniform representation of values between GC & RAII.
- Treat borrowed values as if GC allocated.

A runtime classification of pointers makes the GC go faster (OCaml workshop 2022)

Towards typing

- Rich literature (e.g. Tov and Pucella 2011, and now more recent works, e.g. Radanne et al. 2019)
- But still unsufficient (cf. Rust limitations, active area of research)
- Key design constraint: do not guess linearity from use count
  - ✓ Force making clear when a function is designed to be compatible with RAII (backwards-compatibility & no surprise)
  - ✓ Separate linearity & borrow checking from type inference (ease of implementation, preserve performance of type inference)
- Forces (and motivates) to have a clear story about ownership. Stepping stone towards better data race control in parallel programming.

Summary

1. Types with destructors ⇒ ordered logic
2. Mixing GC and linear allocation with re-use
3. Various notions of resource polymorphism
   - 3.1 parametric resource polymorphism
   - 3.2 borrow polymorphism
   - 3.3 indifference to allocation methods
4. Opportunities for a better story about ownership (including control of aliasing)

Difference with linear types:

- Inspired by Baker's essays on linear logic
- Solves the interaction with control (identified as the next obstacle as far back as Tov & Pucella)
- Linearity phrased as a safety property
- Linear types do not have runtime contents
- Linear types + typeclasses would end up reconstructing something like this (but this is not a story told in the literature)

Thank you

References I

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *ICSE '16*. https://doi.org/10.1145/2889160.2889229

David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 50–68. https://doi.org/10.1145/1028976.1028982

Henry G Baker. 1992. Lively linear lisp: "Look ma, no garbage!". *ACM Sigplan notices* 27, 8 (1992), 89–98.

Henry G. Baker. 1994a. Linear logic and permutation stacks - the Forth shall be first. *SIGARCH Computer Architecture News* 22, 1 (1994), 34–43. https://doi.org/10.1145/181993.181999

References II

Henry G. Baker. 1994b. Minimum Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. *SIGPLAN Notices* 29, 9 (1994), 38–43. https://doi.org/10.1145/185009.185016

Henry G. Baker. 1995. "Use-Once" Variables and Linear Objects - Storage Management, Reflection and Multi-Threading. *SIGPLAN Notices* 30, 1 (1995), 45–52. https://doi.org/10.1145/199818.199860

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science)*, Luca Cardelli (Ed.), Vol. 2743. Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. *A resource modality for RAII (abstract)*. Technical Report. INRIA. https://hal.inria.fr/hal-01806634

References III

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A New Deconstructive Logic: Linear Logic. *Journal of Symbolic Logic* 62 (3) (1997), 755–807.

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 525–539. `https://doi.org/10.1145/3062341.3062357`

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 7–21. `https://doi.org/10.1007/11693024_2`

References IV

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

Jean-Yves Girard. 1991. A new constructive logic: Classical logic. *Math. Struct. Comp. Sci.* 1, 3 (1991), 255–296.

Jean-Yves Girard. 1993. On the Unity of Logic. *Ann. Pure Appl. Logic* 59, 3 (1993), 201–217.

Howard E. Hinnant, Peter Dimov, and Dave Abrahams. 2002. A Proposal to Add Move Semantics Support to the C++ Language. (2002). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm`

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. `https://doi.org/10.1145/3158154`

Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180.

References V

Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Paul-André Melliès and Nicolas Tabareau. 2010. Resource modalities in tensor logic. *Ann. Pure Appl. Logic* 161, 5 (2010), 632–653.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml - Functional Programming for the Masses*. O'Reilly.

Guillaume Munch-Maccagnoni. *Probabilistic resource limits, or: Programming with interrupts in OCaml*. Technical Report. INRIA.

Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. (2018). `https://arxiv.org/abs/1803.02796`

Guillaume Munch-Maccagnoni. 2020. *Towards better systems programming in OCaml with out-of-heap allocation*. Technical Report. INRIA, Jersey City, United States. 1–6 pages. `https://hal.inria.fr/hal-03142386`

Guillaume Munch-Maccagnoni. 2022. *Efficient "out of heap" pointers for multicore OCaml*. Technical Report. INRIA.

References VI

Guillaume Munch-Maccagnoni and Rémi Douence. 2019. *Efficient Deconstruction with Typed Pointer Reversal (abstract)*. Technical Report. INRIA. `https://hal.archives-ouvertes.fr/hal-02177326`

Guillaume Munch-Maccagnoni and Gabriel Scherer. 2022. *Boxroot, fast movable GC roots for a better FFI*. Technical Report. INRIA.

Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 349–365. `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen`

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2019. Kindly Bent to Free Us. (2019). arXiv:cs.PL/1908.09681

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

References VII

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436