A proposal for a resource-management model for OCaml

Guillaume Munch-Maccagnoni

Inria, LS2N CNRS

25th June 2018
Gallium Seminar

Long: the goal is not to get to the end of the talk
Interrupt me if anything sounds unclear/dubious

Manual resource management

```
void f () {
  X * p = new X;
  // ...
  g(p);
}

void g(X * p) {
  // ...
  delete p;
}
```

Manual resource management

```
void f () {
  X * p = new X;
  // ...
  g(p);
}

void g(X * p) {
  // ...

}
```

Leak

Manual resource management

```
int f () {
  X * p = new X;
  // ...
  return 0;
  // ...
  delete p;
}
```

Leak

Manual resource management

```
void f () {
  X * p = new X;
  // ...
  g(p);
  // ...
  delete p;
}

void g(X * p) {
  // ...
  throw std::runtime_error("error");
}
```

Leak

Manual resource management

```
void f () {
  X * p = new X;
  // ...
  g(p);
  // ...
  delete p;
}

void g(X * p) {
  // ...
  delete p;
}
```

Double-free

Manual resource management

```
void f () {
  X * p = new X;
  // ...
  g(p);
  // ...
  h(p);
}

void g(X * p) {
  // ...
  delete p;
}
```

Use-after-free

Manual resource management

```
void f (std::vector<std::string> vec) {
  std::string const & x = vec[0];
  // ...
  vec.push_back("resize");
  // ...
  g(x);
}
```

Iterator invalidation

Manual resource management

Resource: value which is hard to copy or dispose of

- large or shared data structures
  (⇒ memory management)
- low-level abstractions (continuations...)
- anything that needs to be cleaned-up (file handle, sockets, locks, values from a foreign runtime...)
- anything that restricts aliasing
- ...any data structure containing the above (lists of resources, closures of resources...)

Automatic resource management

Garbage collection

Automatic resource management

Thanks

Questions?

Automatic resource management

Garbage collection

A run-time optimisation that anticipates or delays the collection of resources that can be trivially disposed of.

Automatic resource management

Well done! Now what about the rest?

Automatic resource management

Destructors

“Resource Acquisition Is Initialisation” (RAII)

Stroustrup

Automatic resource management

```
void f () {
  X a;
  // ...
  g(a);
  // ...
  // <- a.~X()
}
```

Automatic resource management

```
void f () {
  X a;
  // ...
  g(a); // <- a.~X()
  // ...
}

void g (X const & a) {
  // ...
  throw std::runtime_error("error");
}
```

Automatic resource management

Basic exception-safety (Stroustrup):
Leave data in a valid state, do not leak

Not GC-based finalizers
(need predictability and reliability)

Automatic resource management

Move semantics

Baker (1994), Hinnant et al. (2003)

Ownership/affine types

Automatic resource management

```
void f () {
  auto a = make_unique<X>();
  // ...
  g(move(a));
  // ...
}

void g (std::unique_ptr<X> a) {
  // ...
  // <- a.~X(); free(a);
}
```

Automatic resource management

```
void f () {
  auto a = make_unique<X>();
  std::vector<std::unique_ptr<X>> vec{move(a)};
  // ...
  g(move(vec));
  // ...
}

void g (std::vector<std::unique_ptr<X>> vec) {
  // ...
  // <- ~X(); delete vec;
}
```

Automatic resource management

```
void f () {
  auto a = make_unique<X>();
  Mutex<X> m{move(a)};
  // ...
  g(m);
  // ...
  // <- ~X()
}

void g (Mutex const & m) {
  Lock l = m.lock();
  X & x = l.access();
  // ...
  // <- l.~Lock();
}
```

Automatic resource management

```
void f () {
  auto a = make_unique<X>();
  // ...
  g(move(a));
  // ...
✗ h(a); // We want a compile error
}

void g (std::unique_ptr<X> a) {
  // ...
  // <- a.~X(); free(a);
}
```

Automatic resource management

Borrowing (regions)

Tofte-Talpin-Birkedal…

Cyclone

Automatic resource management

```
int f () {
  auto p = make_unique<X>();
  X & val = *p;
  // ...
  g(move(p));
  // ...
✗ h(val); // We want a compile error
}
```

Automatic resource management

Linear borrows (control of aliasing)
Rust

Automatic resource management

```
void f (std::vector<std::string> vec) {
  std::string const & x = vec[0];
✗ vec.push_back("resize");
  g(x); // We want a compile error
}
```

Girard’s polarisation

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1443

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

A NEW CONSTRUCTIVE LOGIC : CLASSICAL LOGIC

Jean-Yves GIRARD

Juin 1991

RR-1443

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1467

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

ON THE UNITY OF LOGIC

Jean-Yves GIRARD

Juin 1991

RR-1467

Girard’s polarisation

$$[\![A \vee B]\!] =^{?} !A \oplus !B$$

$$!A \oplus !(!B \oplus !C) \neq !(!A \oplus !B) \oplus !C$$

Girard’s polarisation

- Assign *polarities* to formulae corresponding to the structural rules they satisfy
- Only introduce modalities where needed to force a polarity

Girard’s polarisation

A	B	$A\wedge B$	$A\vee B$	$A\Rightarrow B$	$A\supset B$	$\forall xA$	$\exists xA$
$+1$	$+1$	$A\otimes B$	$A\oplus B$	$A\multimap ?B$	$A\multimap B$	$\bigwedge x?A$	$\bigvee xA$
0	$+1$	$!A\otimes B$	$!A\oplus B$	$!A^{\perp}\oplus B$	$!A\multimap B$	$\bigwedge x?A$	$\bigvee x!A$
-1	$+1$	$!A\otimes B$	$A\parr ?B$	$A^{\perp}\oplus B$	$!A\multimap B$	$\bigwedge xA$	$\bigvee x!A$
$+1$	0	$A\otimes !B$	$A\oplus !B$	$A\multimap ?!B$	$A\multimap B$		
0	0	$A\& B$	$!A\oplus !B$	$!A^{\perp}\oplus !B$	$!A\multimap B$		
-1	0	$A\& B$	$A\parr ?!B$	$A^{\perp}\oplus !B$	$!A\multimap B$		
$+1$	-1	$A\otimes !B$	$?A\parr B$	$A\multimap B$	$A\multimap B$		
0	-1	$A\& B$	$?!A\parr B$	$?!A^{\perp}\parr B$	$!A\multimap B$		
-1	-1	$A\& B$	$A\parr B$	$!A\multimap B$	$!A\multimap B$		

tableau 3 : classical and intuitionitic connectives definition in terms of linear logic

Girard's polarisation

$$!A \oplus \underbrace{(!B \oplus !C)}_{+1} = (!A \oplus !B) \oplus !C$$

Girard's polarisation

Goal: minimise the number of modalities to maximise type isomorphisms, valid η expansions...

Girard’s polarisation

You know:

- `Nullable`
- `lazy`
- Reference-counted pointers

Girard's polarisation

Features from Girard & co:

- *Polarity*: type of types that share a computational behaviour (Rust's built-in traits, see also Eisenberg & Peyton Jones's *"kinds as calling conventions"*)
- Coercions between polarities which can possess a computational contents
- Standard set of connectives & automatic inference of polarities and coercions (polarity tables)

Girard's polarisation

The proposal

Polarity = Resource management mode

- **U** (Unrestricted) GC
- **O** (Ownership) RAII + move semantics
- **B** (Borrow) Regions, allocation-method agnostic

A notion of *resource polymorphism* inspired by the C++98 → C++11 transition (Hinnant et al.) for mixing polarities and ensuring backwards-compatibility

A resource modality for RAII

Joint work with G. Combette:

A resource modality for RAII

(talk at LOLA 2018 next month)

A resource modality for RAII

Template for the Ownership modality

A resource modality for RAII

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Delta, \Gamma' \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C}$$

$$\frac{}{\vdash \mathbf{1}} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \mathbf{1}, \Delta \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, A \multimap B, \Delta, \Gamma' \vdash C}$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash B \multimapinv A} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, B \multimapinv A, \Gamma' \vdash C}$$

$$\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C} \qquad \frac{\Gamma, \Gamma' \vdash C}{\Gamma, A, \Gamma' \vdash C}$$

A resource modality for RAII

Attach a destructor to a type,
to create a new type

A resource modality for RAII

Affine typing is not at odds
with the linear logic narrative,
but arises from it

A resource modality for RAII

Ordered data types

$$(A, \delta^{A \to TI}) \otimes (B, \delta'^{B \to TI}) = (A \otimes B, \lambda(a,b).(\delta(a);\delta'(b))^{A \otimes B \to TI})$$

(unless the monad T is commutative)

A resource modality for RAII

Exceptions

A resource modality for RAII

Destructors cannot raise

A resource modality for RAII

Moving performs an effect

This proposal

Propositions in language design and implementation

Looking for the *"sweet spot"*: between simplicity, modularity, expressiveness...

Three levels

1. Type system
2. Language abstractions (here)
3. Runtime (here)

This proposal

Moving and erasure perform effects

Key design point: 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)

Ownership polarity

Naive approach

- A special *drop* (typeclass | trait | modular implicit) baked into the compiler
- Two types of types: **O**wnership (with drop and move semantics) and **U**nrestricted (as usual)
- **U** <: **O** for parametric polymorphism
- Assume for now everything is GC-allocated

Ownership polarity

```
type u = t with destructor f
                (* must not raise *)

type file_in = in_channel
  with destructor close_in_noerr
```

Ownership polarity

```
let open_file name : file_in =
  new file_in (open_in name)
```

Ownership polarity

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

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

Ownership polarity

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

Ownership polarity

```
let twice1 name =
  let f = open_file name in
✗ (f,f) (* typing error: f is affine *)
```

Ownership polarity

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

Ownership polarity

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

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

Ownership polarity

```
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** (abstract type)

Compile twice (monomorphisation of polarities)

- **U** Compiled as usual
- **O** Compiled according to RAII and move semantics, receives destructor in argument (modular implicit)

Ownership polarity

(What I do not speak about: Types of closures)

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 the type system light and convenient, Alms uses expressive kinds that minimize notation while maximizing polymorphism between affine and unlimited types.

A key feature of Alms is the ability to introduce abstract affine types via ML-style signature ascription. In Alms, an interface can impose stiffer resource usage restrictions than the principal usage restrictions of its implementation.

Borrow polarity

&

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

Borrow polarity

Cf. Real World OCaml

```
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
```

Borrow polarity

```
type t = u with destructor f

x : t & ⇒ x : u &
```

Borrow polarity

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

Borrow polarity

Linear Abstract Data Types (Baker)

```
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 polarity

Operating on borrowed values

```
filter : ('a -> bool) -> 'a list -> 'a list
⇒ 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 polarity

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

Borrow polarity

```
(t * u) & = t & * u &
(t list) & = (t &) list
(t : G) & = t
```

Borrow polarity

Mild case of iterator invalidation:

```
(* x : (string * File.t) list *)
let y = &x in
(* y : (string * File.t &) list *)
drop x;
✗print_endline (match hd y with (x,y) -> x)
```

Borrow polarity

No access to data
after destructors have been called

Borrow polarity

A new polarity: the Borrow polarity

- Attach lifetime/region annotation to the polarity
- The lifetime/region annotation is inherited

```
t &@a : B@a
G <: B@a
t : B@a ∧ u : B@a ⇒ t * u : B@a
```

(annotation inspired by Leo White's region-based resource management with the type-and-effect system)

Borrow polarity

The same design lets us consider managing memory using RAII

Summary

Discussed here:

- New types: `affine(M : Droppable) | t &`
  with a built-in module type definition

```
Droppable = sig
  type t
  val drop : t -> unit
end
```

- New terms: `new t (e) | &x`
  Optional ownership annotation for polymorphic bound variables

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

The essence of RAII allocation

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

- Stack allocation & memcpy
- Unique pointers
  - Ownership & borrowing discipline
  - "As efficient" as raw malloc/free
- Reference-counted pointers
  - Copiable
  - Many costs
  - Baker: minimise cost by moving, borrowing and deferred copying

The essence of RAII 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 performed by the tracing collector, there is a precisely correspond-

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 multiprocessor concurrent collection [21, 22, 27, 28, 39].

The essence of RAII allocation

Issues with reference-counting

- ✗ Count-update is costly and inefficient
- ✗ Cycles leak
- ✗ Upfront allocation cost
- ✗ Latency due to upfront deallocation cost, sometimes cascading

The essence of RAII allocation

RAII allocation

Trace dead cells (with destructors)

= RC restricted to a unique reference
(old idea, see Baker)

Cyclone's dynamic regions

The essence of RAII allocation

Allocate with RAII

- ✓ No reference count to update
- ✓ No cycles
- Automatic re-use of cells
- Allocator informed as soon as cells are freed, but can delay / do it in a separate thread

Mixing tracing GC and RAII

Set lowest bit to distinguish traced pointers
from untraced RAII pointers

Mixing tracing GC and RAII

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

Register GC root; set destructor to unregister root.

Mixing tracing GC and RAII

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

Forgetful functor
Uniform representation of values between GC&RAII.

Mixing tracing GC and RAII

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

Stop propagation of region information in the type

$$\left(\downarrow^{\mathbf{B}}_{\mathbf{U}}(t\,\boldsymbol{\&}) \otimes \downarrow^{\mathbf{B}}_{\mathbf{U}}(u\,\boldsymbol{\&}) \neq (\downarrow^{\mathbf{B}}_{\mathbf{U}} t \otimes \downarrow^{\mathbf{B}}_{\mathbf{U}} u)\boldsymbol{\&}\right)$$

Mixing tracing GC and RAII

A	B	A * B	A list	A &
$\mathbf{U}$	$\mathbf{U}$	$A \otimes_{\mathrm{GC}} B$	$\mu X^{\mathbf{U}}.(\mathbf{1} \oplus_{\mathrm{GC}} (A \otimes_{\mathrm{GC}} X))$	A
$\mathbf{O}$	$\mathbf{U}$	$A \otimes_{\mathrm{RAII}} \downarrow^{\mathbf{U}}_{\mathbf{O}} B$	$\mu X^{\mathbf{O}}.(\mathbf{1} \oplus_{\mathrm{RAII}} (A \otimes_{\mathrm{RAII}} X))$	$\uparrow^{\mathbf{B}}_{\mathbf{O}} A$
$\mathbf{B}$	$\mathbf{U}$	$A \otimes_{\mathrm{G/R}} B$	$\mu X^{\mathbf{B}}.(\mathbf{1} \oplus_{\mathrm{G/R}} (A \otimes_{\mathrm{G/R}} X))$	A
$\mathbf{O}$	$\mathbf{O}$	$A \otimes_{\mathrm{RAII}} B$		
$\mathbf{B}$	$\mathbf{O}$	$\downarrow^{\mathbf{U}}_{\mathbf{O}} \downarrow^{\mathbf{B}}_{\mathbf{U}} A \otimes_{\mathrm{RAII}} B$		
$\mathbf{B}$	$\mathbf{B}$	$A \otimes_{\mathrm{G/R}} B$		

Semantics

Mixing tracing GC and RAII

A	B	A * B
U	**U**	**U**
O	**U**	**O**
B	**U**	**B**
O	**O**	**O**
B	**O**	**O**
B	**B**	**B**

A list	A &
U	**U**
O	**B**
B	**B**

Types (resulting polarity)

Mixing tracing GC and RAII

A	B	A * B	A list	A &
U	**U**	0	0	0
O	**U**	1	1	1
B	**U**	0	0	0/1
O	**O**	1		
B	**O**	1		
B	**B**	0		

Runtime (tag for newly-introduced values, 0=traced)

Mixing tracing GC and RAII

Generational GC (tracing live)

✓ No discipline
  - Shared data structures & shared mutable state
  - Cycles

✓ Cheap on allocation

✓ Almost free for short-lived values

Mixing tracing GC and RAII

RAII (tracing dead)

- ✓ During life: no cost & no interruption
- ✓ Pointers do not move
  - No read/write barrier
  - Can by given or lent to foreign runtimes
- ✓ No synchronisation

Mixing tracing GC and RAII

RAII allocation suitable for

- very-long-lived and large data (no GC load)
- interoperability with systems languages (efficiently and expressively)
- performance-sensitive paths (pre-allocate a free list, re-use cells during hot path, and clean-up after)

Mixing tracing GC and RAII

Implementation: a design space for the allocator to explore.

How to best take advantage of the statically-known re-usability and timeliness?

Mixing tracing GC and RAII

Language design : expressiveness vs. concision

"RAII hypothesis"

(cf. generational hypothesis)

- RAII-allocated types $\subseteq$ types with destructors (obviously)
- Anybody using destructors already pays most of the costs (ownership & borrowing discipline, traversing the whole structure on destruction)
- Heuristic: types with destructors $\subseteq$ RAII-allocated types

Mixing tracing GC and RAII

- ✓ Leaves the door open to affine types without destructors, still using GC (e.g. mutable borrows)
- ✗ Could still greatly benefit from a better support for stack allocation/unboxing
- ✓ Will be able to compare GC-allocation and RAII-allocation for **O** types, all other things remaining equal (meaningful benchmarks)

Mixing tracing GC and RAII

Resources can be explored FP-style
with GC-allocated structures by borrowing

cf. Rust's borrow splitting, slice patterns

Example: the borrowed zipper (blackboard)

Towards a type system

Nourished from discussions with Leo White and integrating contributions from him.

3 separate components

1. Type inference & type checking:
   - Main novelty: structural functors
     `(t * u) & = (t &) * (u &)`, etc.
   - Abstract types: type-dependent polarities
     `type +'a t : <'a>` (cf. Tov & Pucella)
2. Linearity and borrow checking: integration with the type-and-effect system
   - Accessing a value of polarity **B@a** performs an effect @a (non-lexical lifetimes)
   - Decomposition of Rust's copiable, read-only borrow as `t & const`
3. A separation logic to verify `unsafe` code (à la RustBelt)

References I

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

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

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

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

Henry G. Baker. Minimum reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Notices*, 29(9):38–43, 1994b. doi: 10.1145/185009.185016.

Henry G. Baker. "use-once" variables and linear objects - storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, 1995. doi: 10.1145/199818.199860.

References II

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

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

Howard E. Hinnant, Peter Dimov, and Dave Abrahams. A proposal to add move semantics support to the c++ language, 2002. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm`.

Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003. doi: 10.1007/978-3-540-45070-2_9.

References III

David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In John M. Vlissides and Douglas C. Schmidt, editors, *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*, pages 50–68. ACM, 2004. doi: 10.1145/1028976.1028982.

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In Peter Sestoft, editor, *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*, volume 3924 of *Lecture Notes in Computer Science*, pages 7–21. Springer, 2006. doi: 10.1007/11693024_2.

References IV

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

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

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

References V

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