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

void g(X * p) {
    // ...
    delete p;
}
Manual resource management

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

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

Leak
```
Manual resource management

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

Leak
```
Manual resource management

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

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

Leak
```
Manual resource management

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

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

Double-free
Manual resource management

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

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

Use-after-free
Manual resource management

```cpp
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

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

```cpp
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


Ownership/affine types
Automatic resource management

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

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

```cpp
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**

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

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

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

```cpp
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

```cpp
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

A NEW CONSTRUCTIVE LOGIC: CLASSICAL LOGIC

Jean-Yves GIRARD

Juin 1991
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A∧B</th>
<th>A∨B</th>
<th>A→B</th>
<th>A⇔B</th>
<th>∀xA</th>
<th>∃xA</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+1</td>
<td>A⊗B</td>
<td>A⊕B</td>
<td>A∞B</td>
<td>A⇒B</td>
<td>∀xB</td>
<td>∀xA</td>
</tr>
<tr>
<td>0</td>
<td>+1</td>
<td>!A⊗B</td>
<td>!A⊕B</td>
<td>!A∞B</td>
<td>!A⇒B</td>
<td>∀xA</td>
<td>∀xA</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>!A⊗B</td>
<td>A?B</td>
<td>A'?B</td>
<td>!A⇒B</td>
<td>∀xA</td>
<td>∀xA</td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
<td>A⊗!B</td>
<td>A⊕!B</td>
<td>A∞!B</td>
<td>A⇒B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A&amp;B</td>
<td>A⊗!B</td>
<td>!A⊗!B</td>
<td>!A⇒B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>A&amp;B</td>
<td>A?B</td>
<td>A'?B</td>
<td>!A⇒B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>A⊗!B</td>
<td>?A&amp;B</td>
<td>A&lt;B</td>
<td>A⇒B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>A&amp;B</td>
<td>?!A&amp;B</td>
<td>?!A&amp;B</td>
<td>!A⇒B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>A&amp;B</td>
<td>A?B</td>
<td>A&gt;B</td>
<td>!A⇒B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tableau 3: classical and intuitionistic connectives

Definition in terms of linear logic
Girard’s polarisation

\[ !A \oplus (!B \oplus !C) = (!A \oplus !B) \oplus !C \]

+1
Girard’s polarisation

Goal: minimise the number of modalities to maximise type isomorphisms, valid $\eta$ 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

\[
\begin{array}{c}
\Gamma \vdash A \\
\hline
A \vdash A \\
\hline
\Gamma \vdash A, \Delta \vdash B \\
\hline
\Gamma, \Delta \vdash A \otimes B \\
\hline
\Gamma \vdash 1 \\
\hline
\Gamma, A \vdash B \\
\hline
\Gamma \vdash A \multimap B \\
\hline
A, \Gamma \vdash B \\
\hline
\Gamma \vdash B \multimap A
\end{array}
\]

\[
\begin{array}{c}
\Delta \vdash A \\
\Gamma, A, \Gamma' \vdash B \\
\hline
\Gamma, \Delta, \Gamma' \vdash B \\
\hline
\Gamma, A, B, \Delta \vdash C \\
\hline
\Gamma, A \otimes B, \Delta \vdash C \\
\hline
\Gamma, \Delta \vdash C \\
\hline
\Gamma, 1, \Delta \vdash C \\
\hline
\Gamma, B, \Gamma' \vdash C, \Delta \vdash A \\
\hline
\Gamma, A \multimap B, \Delta, \Gamma' \vdash C \\
\hline
\Gamma, B, \Gamma' \vdash C, \Delta \vdash A \\
\hline
\Gamma, \Delta, B \multimap A, \Gamma' \vdash C \\
\hline
\Gamma, A, B, \Gamma' \vdash C \\
\hline
\Gamma, B, A, \Gamma' \vdash C \\
\hline
\Gamma \vdash C \\
\hline
\Gamma, \Gamma' \vdash C \\
\hline
\Gamma, A, \Gamma' \vdash C
\end{array}
\]
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

\[(A, \delta^A_{A\to TI}) \otimes (B, \delta^B_{B\to TI}) = (A \otimes B, \lambda(a, b).\left(\delta(a); \delta'(b)\right)^{A\otimes B\to TI})\]

(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: Ownership (with drop and move semantics) and Unrestricted (as usual)
- U <: O for parametric polymorphism
- Assume for now everything is GC-allocated
Ownership polarity

\[
\text{type } u = t \text{ with destructor } f \\
\quad (* \text{ must not raise } *)
\]

\[
\text{type } \text{file}_\text{in} = \text{in}\_\text{channel} \\
\quad \text{with destructor } \text{close}_\text{in\_noerr}
\]
Ownership polarity

```ocaml
let open_file name : file_in =
  new file_in (open_in name)
```
Ownership polarity

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

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

```ocaml
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

```ocaml
let open_list =
    List.map (fun name ->
        (name, open_file name))
(* (string * file_in) list : 0 *)
```
Ownership polarity

let open_list =
    List.map (fun name ->
               (name, open_file name))
(* (string * file_in) list : 0 *)

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

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

Compiling \textbf{U} \textless{} \textbf{O} (abstract type)

Compile twice (monomorphismisation of polarities)

\textbf{U} Compiled as usual

\textbf{O} Compiled according to RAII and move semantics, receives destructor in argument (modular implicit)
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.

We present two pieces of evidence to demonstrate the validity of our design goals. First, we introduce a prototype implementation of Alms and discuss our experience programming in the language. Second, we establish the soundness of the core language. We also use the core model to prove a principal kinding theorem.
Borrow polarity

\[
\text{let } \text{read} \_\text{line} \text{ name } = \\
\text{let } f = \text{open} \_\text{file} \text{ name in} \\
\text{print} \_\text{end}\_\text{line} (\text{input} \_\text{line} \& f); \\
\text{flush stdout}
\]
Borrow polarity

Cf. Real World OCaml

```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

```ocaml
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 : 0
  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

\[ \text{filter} : ('a \to \text{bool}) \to 'a \text{ list} \to 'a \text{ list} \]
\[ \Rightarrow \text{filter} : ('a \& \to \text{bool}) \to ('a \&) \text{ list} \to ('a \&) \text{ list} \]

\[
\text{let } x = \&l \text{ in let } y = \text{filter } f \ x \text{ in } \ldots
\]

\[
(\text{string } \ast \text{ File.t}) \text{ list } \&
\]

vs.

\[
(\text{string } \ast \text{ File.t } \&) \text{ list}
\]
Borrow polarity

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

\[(t * u) & = t & * u &\]
\[(t \text{ list}) & = (t &) \text{ list}\]
\[(t : G) & = t\]
Mild case of iterator invalidation:

\[
(* \ x : (\text{string} \times \text{File.t}) \text{ list } *) \\
\text{let } y = &x \text{ in} \\
(* \ y : (\text{string} \times \text{File.t } \&) \text{ list } *) \\
\text{drop } x; \\
\times\text{print_endline (match hd } y \text{ with } (x,y) \Rightarrow x)\]
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

\[ t \ &@a : B@a \]
\[ G <: B@a \]
\[ t : B@a \land u : B@a \Rightarrow t * u : B@a \]

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

Perry Cheng
perryche@us.ibm.com

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

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, 30].
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

\[ \downarrow_{O}^{U} : U \rightarrow O \]

Register GC root; set destructor to unregister root.
Mixing tracing GC and RAII

\[ \mathbf{B} \circ \mathbf{O} : \mathbf{O} \rightarrow \mathbf{B} \]

Forgetful functor
Uniform representation of values between GC&RAII.
Mixing tracing GC and RAII

$\downarrow^B_U : B \rightarrow U$

Stop propagation of region information in the type

$\left( \downarrow^B_U (t \&) \otimes \downarrow^B_U (u \&) \neq (\downarrow^B_U t \otimes \downarrow^B_U u) \& \right)$
## Mixing tracing GC and RAII

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A * B</th>
<th>A list</th>
<th>A &amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>A \otimes_{GC} B</td>
<td>\mu X^U.(1 \oplus_{GC} (A \otimes_{GC} X))</td>
<td>A</td>
</tr>
<tr>
<td>O</td>
<td>U</td>
<td>A \otimes_{RAII} \downarrow^U_O B</td>
<td>\mu X^O.(1 \oplus_{RAII} (A \otimes_{RAII} X))</td>
<td>\uparrow^B_O A</td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>A \otimes_{G/R} B</td>
<td>\mu X^B.(1 \oplus_{G/R} (A \otimes_{G/R} X))</td>
<td>A</td>
</tr>
<tr>
<td>O</td>
<td>O</td>
<td>A \otimes_{RAII} B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>\downarrow^U_O \downarrow^B_U A \otimes_{RAII} B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A \otimes_{G/R} B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Semantics**
### Mixing tracing GC and RAII

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A * B</th>
<th>A list</th>
<th>A &amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>O</td>
<td>U</td>
<td>O</td>
<td>O</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Types (resulting polarity)
## Mixing tracing GC and RAII

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A * B</th>
<th>A list</th>
<th>A &amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>O</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0/1</td>
</tr>
<tr>
<td>O</td>
<td>O</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ⊆ 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 ⊆ 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
     \((t \times u) \land = (t \land) \times (u \land), etc.\)
   • 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
     \(t \&\) cond

3. A separation logic to verify unsafe code (à la RustBelt)


