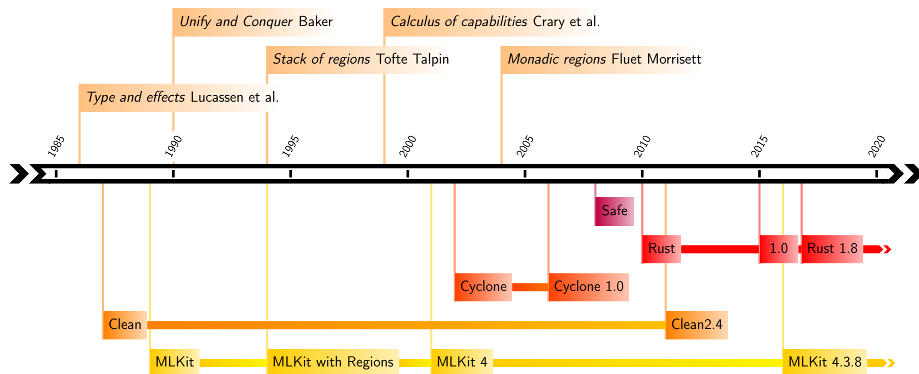
Put some trust in your memory

D. Renault

Haskell small school
LaBRI

1er juillet 2016, v. 1.2.0

A brief history of Rust



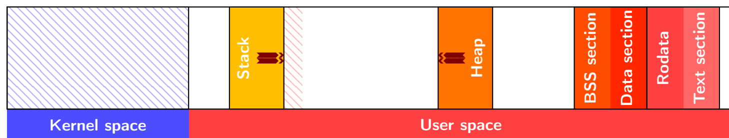
Rust in a nutshell

- Rust is a programming language developed by Mozilla Research, directed by Graydon Hoare.
- Rust is statically-typed, with **functional programming** features, generics and a **traits system** resembling the interfaces of Go.
- Manual memory management along with an advanced type system claiming **memory safety**, without the need of a garbage collector.
- Lacks a formal specification and still evolving, but claims to have reached stability.
- Current highlighted projects :
  - Servo, a « browser engine » developed by Mozilla,
  - Dropbox recently oriented part of its development to Rust

Memory management

Classical strategies for memory management in programming languages :

- **Manual** : alloc./dealloc. of memory in the heap via `malloc` and `free`



Pros : • Complete freedom of management

Cons : • No safety

- **Automatic** : alloc. on demand and dealloc. at end of scope (stack, limited) or depending on a runtime (garbage collection)

Pros : • **Memory safety**

Cons : • Overhead on performance and memory
• Non-deterministic peaks when collecting

Dangers of manual memory management

- Unallocated or **dangling pointer** / Use after free

```
int *pi = (int*) malloc(sizeof(int)); // allocation
*pi = 3;                              // use
free(pi);                             // deallocation
printf("*pi: %d\n", *pi);             // use after free
```

- **Data races** : two threads accessing the same binding read / write

```
void transfer(int amount,
      Account src, Account dst) {
  // Omit test of solvability
  dst.balance += amount;
  src.balance -= amount; }
```

```
Thread t1 = new Thread(() →
  { transfer(10, a1, a2); });
Thread t2 = new Thread(() →
  { transfer(10, a2, a1); });
t1.start(); t2.start();
```

```
a1.balance += 10;
a2.balance -= 10;
```

data race

```
a2.balance += 10;
a1.balance -= 10;
```

Concurrent access to a resource : what is the expected behavior ?

Is it possible to prevent such kinds of problems ?

On the upside, some motivations

- **Move semantics** in C++ : refers to the ability for a binding to take ownership of a value instead of copying it.

```
string(const string& s) {
  size_t size = strlen(s.data)+1;
  data = new char[size];
  memcpy(data, s.data, size);
}
string a(existing_string);
```

```
string(string&& s) {
  data = s.data;
  s.data = nullptr;
}

string b(to_string(pi));
```

Here, `string&&` is an rvalue reference, indicating a **temporary** object.

- **Reuse of memory** in functional programming : combine functions with no side-effects with in-place transformations.

```
(<|)   :: a → Seq a → Seq a        -- add at head
(|>)   :: Seq a → a → Seq a        -- add at tail
update :: Int → a → Seq a → Seq a  -- modify an element
```

Memory can be reused directly, without creating a new element.
Application example : in-place array updates

Scope

Given a binding, its scope is the part of the program where the binding is valid, i.e refers to the value it binds.

In the simple case, this notion is **syntactic** :

```
int sum(int a, int b) {
  int res = 0;
  for (i=a; i<b; i++)
    res += i;
  return res;
}
```


Usually, such a value is automatically allocated on the stack and deallocated at the end of its scope.

For an address, it is possible to extend the notion of scope to the part of the program where the value it points to is correctly allocated.

However, the notion becomes more cunning :

- it is hard to compute, since it can escape the block where it is defined :

```
void sum(int a, int b, int* res) {
  *res = 0;
  for (i=a; i<b; i++)
    *res += i;
}
void main() {
  int s;
  sum(1,10,&s);
}
```

Scope of &s

Allocation

Deallocation

- it can be copied, stored into data structures, accessed in different ways, and also possibly lost.

Aliasing

When two or more bindings corresponding to the same memory address.

Problems :

- what happens if two threads update the same object without noticing each other ?
  $\Rightarrow$ data races, race conditions, deadlocks
- what happens if a reference is never deallocated or worse, if not every aliased binding is noticed of the deallocation ?
  $\Rightarrow$ memory leaks, dangling pointers, `NullPointerException`

Dynamic solutions incur an overhead $\Rightarrow$ is a static analysis possible ?

Undecidable problem in general, even without talking of pointer arithmetic.

Smart pointers and the C++ world

C++11 introduced a series of types for smart pointers as tools for memory management, but the notion is much older and existed in Boost since at least 1994. In the norm C++11 were introduced :

- shared pointers (containing a **reference count**)
- weak pointers (possibly dangling)
- unique pointers (**exclusive ownership**)

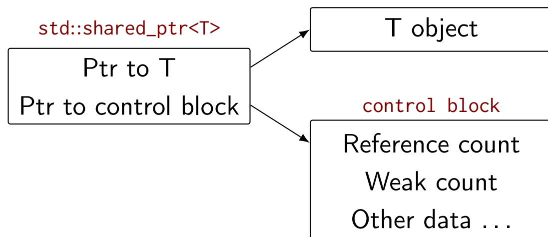
Goal : improve on manual memory management

- restrain the use of manual allocation and destruction operators

  $\neq$ `malloc`, `new`, `free`, `delete`

- let the scope of pointers determine the lifetime of an object.

Shared pointers



- Can be copied, each new copy increasing the reference counter ;
- Each deletion decreases the reference counter, and if reaching zero, deletes the pointed object.

Problems :

- incurs size overhead wrt basic pointer,
- requires atomic reference count manipulations,
- cycles in structures are not detected.

Beware of aliasing

A particular example in this very case :

```
auto pw = new Widget;            // pw is raw ptr
std::shared_ptr<Widget> spw1(pw); // 1st smart ptr
std::shared_ptr<Widget> spw2(pw); // 2nd smart ptr
```

taken from *Effective modern C++*

Same object, different smart pointers with different reference counters.
Solution : never used a naked pointer → `make_shared`.

Unique pointers and uniqueness types

Another solution is to restrict the number of aliases on a given instance.

Unique pointer

A unique pointer is a pointer that cannot be aliased.

- such a pointer cannot be copied, only read through ;
- its ownership can be transferred (e.g. parameter in a function call) ;
- it is deallocated automatically if it loses its single owner.

Pros : memory efficient (no control block),
move semantics for no copy.

Cons : requires some discipline when writing code, and is
sometimes inappropriate when sharing is necessary.

Nevertheless, it is a completely static solution ("zero-cost abstraction")
Cf. also uniqueness types in Clean, Cyclone and Rust.

Aside : with-open-file

In Lisp, the macro `(with-open-file file block)` :

- allocates a reference to `file` at the beginning of the `block` ;
- when control leaves the `block`, either normally or abnormally (such as by use of `throw`), `file` is automatically closed.

```
(with-open-file (stream "/some/file.txt")
  (do-things)
  (format stream "Some text.")
  (do-other-things)
)
```

Scope of `stream`

Deallocation

Also called `call-with-output-file` in Scheme, `withFile` in Haskell, `with` in Python or `try-"with-resources"` in Java.

Regionalization

Idea : localize bindings inside zones of code with automatic lifetime control.

Region

Zone of code and of memory determined statically, allowing allocation of bindings within this zone and handling deallocation at exit.

- pointers are allocated inside a particular region ;
- access to an object is possible only inside the region and always valid ;
- when the region ends, all objects still inside are destroyed.

Regions in Cyclone

In Cyclone the regions are handled in an **explicit** manner :

```
{ region <‘r1> r1;                                             ‘r1
  list_t<int,‘r1> x = NULL;
  { region <‘r2> r2;                                          ‘r2
    list_t<int,‘r2> y = rcopy(r2, x);
    x = rnew(r1) List(4,x);
    y = rnew(r2) List(5,y);
    // x = rnew(r2) List(4,x); // Error : incompatible regions
    // x = rnew(r1) List(4,y); // Error : List in unique region
  }
}
```

- Regions are determined at compile time.
- The set of all active regions behaves at runtime like a LIFO stack.

A special region `‘H` is given corresponding to the heap (that can be garbage-collected) and another corresponding to the stack.

Region polymorphism

In order to handle arbitrary deep stacks of regions, the function calls must be **region-polymorphic**.

Example of a recursive computation creating separate regions.

```
list_t<int,‘r> fibo_rec(region_t<‘r> r, int a, int b, int n) {
  if (n == 0)
    return rnew(r) List(b, NULL);
  else {
    region <‘s> s;      // Recursive computation in different region
    list_t<int, ‘s> z = fibo_rec(s, b, a+b, n−1);
    z = rnew(s) List(b, z);
    return rcopy(r, z); // Copy in the final region
}}

list_t<int,‘r> fibo(region_t<‘r> r, int n) {
  return fibo_rec(r,1,1,n−1); }
```

- Regions are reified into first-class values that can be transmitted.
- Functions accept polymorphic region parameters for genericity.

Effects

Effect

An effect is a set of regions annotated with access rights (read or write) used throughout a function call.

```
struct Counter<`r> { int* @effect(`r) cpt; };

void inc_counter(struct Counter<`r>* c) { // Explicit region
  int *x = c->cpt;
  *x = *x + 1;
}
```

The `inc_counter` function has type :

$$\forall \rho, \texttt{struct Counter}[\rho] \xrightarrow{\{get(\rho), put(\rho)\}} \texttt{void}$$

... meaning that the function accesses ρ by reading and writing into it.

Case of closures : hidden effects

Sometimes, a data structure can retain a pointer inside a region, but the region is not immediately apparent in the type of the structure.

```
struct Counter { <`r>       // Counter with hidden enclosed region
  struct Region<`r>* reg;
  int* @region(`r) cpt;
};
void inc_counter(struct Counter* c) { // No apparent region
  let &Counter{.reg=r,.cpt=x} = c;
  { region r = open(r->key);  // Open the region to access the value
    *x = *x + 1;
  }}
```

The `inc_counter` function now has type :

$$\exists \rho, \texttt{struct Counter}[\rho] \xrightarrow{\{get(\rho), put(\rho)\}} \texttt{void}$$

Note : this is typically the case of closures.

Type and effect system

A **type and effect** system is an enhancement of a type system that infers a conservative approximation of :

(a) the set of regions into which a structure or function can point,

(b) the set of regions that are still live at each program point.

In Safe or MLKit, the regions are handled automatically by the runtime :

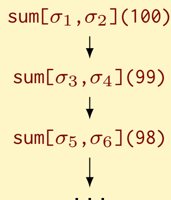
- Regions must be inferred with a unification algorithm in the Hindley-Milner style ;
- Effects being sets of regions, the unification is not first-order, and therefore harder to implement than for types.

Example for a recursive summing function :

```
letrec sum = fun x → if x == 0 then 1 else x + sum (x−1)
in sum(100)
```

Its annotated version :

```
letregion ρ1 in
letrec sum[ρ2,ρ3] at ρ1 = (* Region polymorphism *)
   (fun x:(int,ρ2) →
      if letregion ρ4,ρ5 in (x==(0 at ρ5)) at ρ4
      then 1 at ρ3
      else letregion ρ6 in
           (x + letregion ρ7,ρ8 in
                sum[ρ8,ρ6] at ρ7
                   letregion ρ9 in
                   (x−(1 at ρ9)) at ρ8
           ) at ρ3
   ) at ρ1 in
letregion ρ10,ρ11 in
   sum[ρ11,ρ0] at ρ10 (100 at ρ11)
```



- sum has type $\forall \rho_2 \rho_3, (\texttt{int}, \rho_2) \rightarrow (\texttt{int}, \rho_3)$ and is placed in region ρ_1 ;
- ρ_2 is the region for storing the parameter x, ρ_3 for storing the result.

The same example in a tail-recursive manner :

```
letrec sum = fun res x → if x == 0 then res else sum (x + res) (x−1)
in sum 0 100
```

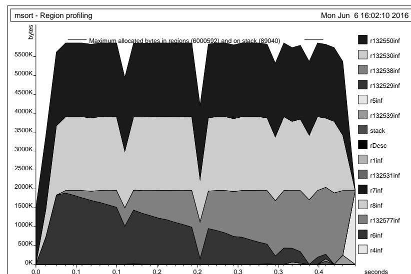
Its annotated version :

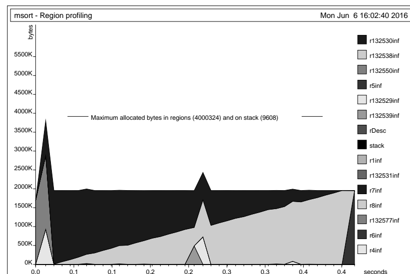
```
letregion ρ1 in
letrec sum [ρ2,ρ3] at ρ1 =
   (fun res:(int,ρ2) x:(int,ρ3) →
      if letregion ρ4,ρ5 in (x == (0 at ρ5)) at ρ4
      then res
      else letregion ρ6 in
           sum[ρ2,ρ6] (* Reuse of the region ρ2 *)
              ((res + x) at ρ2) ((x − 1) at ρ6)
   ) at ρ1 in
letregion ρ7,ρ8
     in sum[ρ8,ρ7] (0 at ρ7) (100 at ρ8)
```


- Still a memory leak for the parameter `x`, which uses a stack of regions.
- In this case, one can use `resetRegions` to force the compiler not to create new regions ⇒ **memory tail-recursivity**.

Benchmarks

- **Memory consumption** : profiling a merge sort algorithm in MLKit for a list of 250000 elements





Without management — With management

Lower triangle : unsorted elements / Upper triangle : sorted elements

- **Performance** : results equivalent with or without management

What about Rust, a more recent language ? It introduces 3 notions :

- **ownership** : for a given resource (cf. unique types), there's exactly one principal binding responsible for it.
- **borrowing** : each resource can have any number of immutable references (aka reference counting with a principal owner) ;
- **lifetimes** : Rust equivalent to scopes or regions.

*"It's just **affine** lambda calculus, plus borrowing to make living with **linearity** easier, plus ML-style effects, polymorphism and typeclasses. Nothing too fancy is going on here."*

Comment on Lambda The Ultimate

Linearity and affinity

A substructural type system is a type system whose underlying logic forbids the use of some structural rules of intuitionistic logic :

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, A, B \vdash B}{\Gamma, B, A \vdash B}\ \text{Exchange}$$

Linearity and affinity

A substructural type system is a type system whose underlying logic forbids the use of some structural rules of intuitionistic logic :

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, A, B \vdash B}{\Gamma, B, A \vdash B}\ \text{Exchange}$$

- Linear logic : every binding is used exactly once.

Linearity and affinity

A substructural type system is a type system whose underlying logic forbids the use of some structural rules of intuitionistic logic :

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, A, B \vdash B}{\Gamma, B, A \vdash B}\ \text{Exchange}$$

- Linear logic : every binding is used exactly once.
- Affine logic : every binding is used at most once.

Linearity and affinity

A substructural type system is a type system whose underlying logic forbids the use of some structural rules of intuitionistic logic :

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, A, B \vdash B}{\Gamma, B, A \vdash B}\ \text{Exchange}$$

- Linear logic : every binding is used exactly once.
- Affine logic : every binding is used at most once.

Some remarks :

- Considered in a strict sense, these logics are not very expressive and their lambda-calculi cannot represent a function such as $x \to x^2$.
- Hence they embed the intuitionistic logic as a subset via exponentials.

Borrowing

Borrowing (Rust)

A (unique) binding `x` may be aliased any number of times (noted `&x`) in an immutable manner.

```
fn foo(v1:Vec<i32>, v2:Vec<i32>)
   → (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2
    // hand them back in the end
    (v1, v2, 42) }

let v1 = vec![1, 2, 3];
let v2 = vec![4, 6, 8];

let (v1, v2, ans) = foo(v1, v2);
```

Without borrowing

```
fn foo(v1:&Vec<i32>, v2:&Vec<i32>)
   -> i32 {
    // do stuff with v1 and v2
    // return the result
    42 }

let v1 = vec![1, 2, 3];
let v2 = vec![4, 6, 8];

let ans = foo(&v1, &v2);
```

With borrowing

- Borrowing alleviates the constraints of working with linearity.

Lifetimes in Rust

- A counter with a reference in Rust :

```
struct Counter<'a> {
    cpt : &'a mut i32 }

fn disp_cpt<'a>(c : &'a Counter) {
    println!("cpt␣=␣{}", c.cpt); }
fn incr_cpt<'a>(c : &'a mut Counter){
    *c.cpt = *c.cpt + 1; }
```

```
let mut c = Counter {
    cpt: &mut 5i32 };

disp_cpt(&c);
incr_cpt(&mut c);
```

- The concatenation function on `Vec<T>` :

```
fn concat<'a,'b,T:Clone>(x : &'a Vec<T>, y : &'b Vec<T>)→Vec<T> {
    let mut z = Vec::new();
    for xx in x { z.push(xx.clone()); }
    for yy in y { z.push(yy.clone()); }
    return z;
}
```

Most of the time, the region variables are inferred by the compiler.

Rust closures

The management of ownership with closures may be subtle :

- A closure with a modifiable internal state ($\approx$ existential type) :

```
fn mk_closure() → Box<FnMut() → i32> {
    let mut num = 0;  // must be Sized
    return Box::new(move || {
        num += 1;
        return num;
    });}
```

```
let mut f = mk_closure();

println!("{}", f()); // 1
println!("{}", f()); // 2
println!("{}", f()); // 3
```

  The `move` keyword forces the closure to take ownership of `num`.

- This can also be implemented with **trait objects**.

```
struct Counter<'a> { cpt : &'a mut i32 }

trait Cpt { fn inc(&mut self) → i32; }
impl<'a> Cpt for Counter<'a> {
    fn inc(&mut self) → i32 {
        *(self.cpt) += 1;
        return *(self.cpt);}}

fn inc_all(c : &mut Cpt) { c.inc(); }
```

```
let mut c = Counter {
    cpt: &mut 5i32 };
{ // Trait object
  let d = &mut c
          as &mut Cpt;
  inc_all(d);
}
disp_cpt(&c);
```

Some pointers

- The Cyclone language : `https://cyclone.thelanguage.org/`
- The MLKit language : `https://www.elsman.com/mlkit`
- The Rust language : `https://www.rust-lang.org/`
  and documentation : `https://doc.rust-lang.org/stable/book/`
- "*Effective Modern C++*", S. Meyers, O'Reilly Media.

Some (educated) reading

- *"Implementation of the typed Call-by-value λ-calculus using a stack or regions"*, M. Tofte and J.-P. Talpin, *POPL '94*.
- *"Region-based memory management in Cyclone"*, Grossman et. al., *PLDI '02*.
- *"Typed memory management in a calculus of capabilities"*, Crary et. al., *POPL '99*.
- *"Monadic regions"*, M. Fluet and G. Morrisett, *ICFP '04*.