Rust for functional programmers

Raphael Poss

July 2014

modified: 2020-01-25
category: Programming
tags: rust, haskell, ocaml, functional programming, language comparison, analysis, programming languages

This post follows up on OCaml for Haskellers from Edward Z. Yang (2010) and my own Haskell for OCaml programmers from earlier this year.

Contents

Prologue 2

Why you should learn Rust 2

Straightforward equivalences 4

Traits: Rust's type classes 7

Ad-hoc objects and methods 9

Safe references 10

Lifetime and storage, and managed objects 12

Shared objects: Rc and Arc 14

Macros and meta-programming 15

Literals 16

Acknowledgements 18

References 18

> **Note**
> The latest version of this document can be found online at https://dr-knz.net/rust-for-functional-programmers.html. Alternate formats: Source, PDF.

Prologue

Rust for C programmers != Rust for functional programmers.

For experienced C programmers with little experience in anything else, Rust presents many new features and strange constructs that look and feel relatively arcane. Even to C++, Java or C# programmers, Rust looks foreign: its has objects but not classes, it has "structured enums", a strange "match" statement with no equivalent in the C family, it prevents from assigning twice to the same variable, and all manners of strange rules that were unheard of in the C family.

Why is that?

Although no current Rust manual dares putting it so bluntly, **Rust is a functional language**, inspired from recent advances in programming language design.

The problem faced by Rust advocates is that "functional programming," both as a term and as a discipline, has developed many stigmas over the last 30 years: functional programs are "unreadable", they are relatively slow compared to C, they require heavyweight machinery during execution (at least a garbage collector, often many functional wrappers around system I/O), they are difficult to learn, they use a strange syntax, etc.

Trying to describe Rust as a functional language to C programmers, its primary audience, would be a tough sell indeed. This is why the official Rust manuals and tutorials duly and properly explain how to use Rust for low-level tasks, and care to explain step-by-step the Rust equivalents to many C programming patterns, without refering to other more modern languages.

This is all and well, but what if you already know about functional programming?

For experienced users of Haskell, OCaml, etc., yet another detailed manual that presents the basics of programming with a functional flavor are a bore to read. Tailored to this audience, the text below constitutes **a fast-track introduction to Rust**, from the functional perspective.

Why you should learn Rust

For better or for worse, most hardware processors will continue to be based on program counters, registers and addressable memory for the foreseeable future. This is where we were in the 1970's, when C was designed, and this is where we are still today, and this is precisely why C is still in prevalent use. Specifically, the C *abstract machine model* is the snuggest fit for most hardware platforms, and it is therefore a good level of abstraction to build low-level system software like interrupt service routines, garbage collectors and virtual memory managers.

However, the "user interface" of the C language, in particular *its preprocessor and its type system*, have aged tremendously. For anyone who learned anything newer, *they frankly suck*.

This is where Rust has been designed: **Rust keeps the C abstract machine model but innovates on the language interface**. Rust is expressive, its type system makes system code safer, and its powerful meta-programming facilities enable new ways to generate code automatically.

Note however that Rust is not yet fully stable at the time of this writing: it is still subject to change with little notice, and the official documentation is not fully synchronized with the implementation.

Straightforward equivalences

Syntax	Type	Ocaml	Haskell
`()`	()	`()`	`()`
`true`	bool	`true`	`True`
`false`	bool	`false`	`False`
`123`	(integer)		`123 {- :: Num a => a }`
`0x123`	(integer)		`0x123 {- :: Num a => a }`
`12.3`	(float)		`12.3 {- :: Fractional a => a }`
`'a'`	char		`'a'`
`"abc"`	str	`"abc"`	`"abc"`
`b'a'`	u8	`'a'`	`toEnum$fromEnum$'a'::Word8`
`123i`	int	`123`	`123 :: Int`
`123i32`	i32	`123l`	`123 :: Int32`
`123i64`	i64	`123L`	`123 :: Int64`
`123u`	uint		`123 :: Word`

Like in Haskell, Rust number literals without a suffix do not have a predefined type. Their actual type is inferred from the context. Rust character literals can represent any Unicode scalar value, in contrast to OCaml's character which can only encode Latin-1 characters. Rust's other literal forms are presented in Literals below.

Primitive types:

Rust	Haskell	OCaml	Description
()	()	unit	Unit type
bool	Bool	bool	Boolean type
int	Int	int	Signed integer, machine-dependent width
uint	Word		Unsigned integer, machine-dependent width
i8	Int8		8 bit wide signed integer, two's complement
i16	Int16		16 bit wide signed integer, two's complement
i32	Int32	int32	32 bit wide signed integer, two's complement
i64	Int64	int64	64 bit wide signed integer, two's complement
u8	Word8	char	8 bit wide unsigned integer
u16	Word16		16 bit wide unsigned integer
u32	Word32		32 bit wide unsigned integer
u64	Word64		64 bit wide unsigned integer
f32	Float		32-bit IEEE 754 binary floating-point
f64	Double	float	64-bit IEEE 754 binary floating-point
char	Char		Unicode scalar value (non-surrogate code points)

...continued on next page

Rust	Haskell	OCaml	Description
str			UTF-8 encoded character string.

The type `str` in Rust is special: it is *primitive*, so that the compiler can optimize certain string operations; but it is not *first class*, so it is not possible to define variables of type `str` or pass `str` values directly to functions. To use Rust strings in programs, one should use string references, as described later below.

Operators equivalences:

```
==  !=  <  >  <=  >=  && ||   // Rust
=   <>  <  >  <=  >=  && ||   (* OCaml *)
==  /=  <  >  <=  >=  && ||   {- Haskell -}

+  +  +  +  -  -  *  *  /     /  %     !    // Rust
+  +. @  ^  -  -. *  *. /     /. mod   not  (* OCaml *)
+  +  ++ ++ -  -  *  *  `div` /  `mod` not  {- Haskell -}

&    |   ^    <<     >>     !          // Rust
land lor lxor [la]sl [la]sr lnot       (* OCaml *)
.&.  .|. xor  shiftL shiftR complement {- Haskell -}
```

Note that Rust uses `!` both for boolean negation and for the unary bitwise NOT operator on integers. The unary `-` has a different meaning in Rust than in C, detailed later below.

Compound expressions:

Rust	OCaml	Haskell	Name
`R{a:10, b:20}`	`{a=10; b=20}`	`R{a=10, b=20}`	Record expression
`R{a:30, ..z}`	`{z with a=30}`	`z{a=30}`	Record with functional update
`(x,y,z)`	`(x,y,z)`	`(x,y,z)`	Tuple expression
`x.f`	`x.f`	`f x`	Field expression
`[x,y,z]`	`[|x;y;z|]`		Array expression, fixed size
`[x, ..10]`			Array expression, fixed repeats of first value
`x[10]`	`x.(10)`	`x!10`	Index expression (vectors/arrays)
`x[10]`	`x.[10]`	`x!!10`	Index expression (strings)
`{x;y;z}`	`begin x;y;z end`		Block expression
`{x;y;}`	`begin x;y;() end`		Block expression (ends with unit)

Note that the value of a block expression is the value of the last expression in the block, except when the block ends with a semicolon, in which case its value is `()`.

Functions types and definitions:

```
// Rust
// f : |int,int| -> int
fn f (x:int, y:int) -> int { x + y };

// fact : |int| -> int
fn fact (n:int) -> int {
   if n == 1 { 1 }
   else { n * fact(n-1) }
}
```

```
(* OCaml *)
(* val f : int * int -> int *)
let f (x, y) = x + y

(* val fact : int -> int *)
let rec fact n =
    if n = 1 then 1
    else n * fact (n-1)
```

```
{- Haskell -}
f :: (Int, Int) -> Int
f (x, y) = x + y

fact :: Int -> Int
fact n =
    if n = 1 then 1
    else n * fact(n-1)
```

Pattern match and guards:

```rust
// Rust
match e {
  0           => 1,
  t @ 2       => t + 1,
  n if n < 10 => 3,
  _           => 4
}
```

```
(* OCaml *)
match e with
 | 0             -> 1
 | 2 as t        -> t + 1
 | n when n < 10 -> 3
 | _             -> 4
```

```
{- Haskell -}
case e of
  0          -> 1
  t @ 2      -> t + 1
  n | n < 10 -> 3
  _          -> 4
```

Recursion with side effects:

```rust
// Rust
fn collatz(n:uint) {
  let v = match n % 2 {
     0 => n / 2,
     _ => 3 * n + 1
  }
  println!("{}", v);
  if v != 1 { collatz(v) }
}

fn main() { collatz(25) }
```

```
(* OCaml *)
let rec collatz n =
    let v = match n % 2 with
       | 0 -> n / 2
       | _ -> 3 * n + 1
    in
    Printf.printf "%d\n" v;
    if v <> 1 then collatz v

let _ = collatz 25
```

```
{- Haskell -}
collatz n = do
    let v = case n `mod` 2 of
            0 -> n `div` 2
            _ -> 3 * n + 1

    putStrLn $ show v
    when (v /= 1) $ collatz v

main = collatz 25
```

Obviously, Rust uses strict (eager) evaluation and functions can contain side effecting expressions, just like in OCaml.

Note that Rust does not (yet) guarantee tail call elimination, although the underlying LLVM code generator is smart enough that it should work for the function above. When in doubt, the following is equivalent:

```rust
let mut n = 25u;
while n != 1 {
   n = if n % 2 == 0 { n / 2 }
       else { 3 * n + 1 }
   println("{}", n);
}
```

Record types, expressions and field access:

```rust
// Rust
struct Point {
  x : int,
  y : int
}

let v = Point {x:1, y:2};

let s = v.x + v.y
```

```
(* OCaml *)
type Point = {
  x : int;
  y : int
}

let v = { x = 1; y = 2 };

let s = v.x + v.y
```

```
{- Haskell -}
data Point = Point {
   x :: Int,
   y :: Int
}

v = Point {x = 1, y = 2}

s = (x v) + (y v)
```

Free type parameters (generic data and function types):

```rust
// Rust
type Pair<a,b> = (a,b)

// id<t> : |t| -> t

fn id<t>(x : t) -> t { x }
```

```
(* OCaml *)
type ('a, 'b) pair = 'a * 'b

(* val id : 't -> 't *)

let id x = x
```

```
{- Haskell -}
type Pair a b = (a, b)

id :: t -> t

id x = x
```

Algebraic data types:

```rust
// Rust
enum Option<T> {
   None,
   Some(T)
}
// x : Option<t>
match x {
  None    => false,
  Some(_) => true
}
```

```ocaml
(* OCaml *)
type 't option =
    None
  | Some of 't

(* x : t option *)
match x with
  | None -> false
  | Some _ -> true
```

```haskell
{- Haskell -}
data Maybe a =
    Nothing
  | Just a

{- x : Maybe t -}
case x of
   Nothing -> False
   Just _ -> True
```

Lambda expressions and higher-order functions:

```rust
// Rust
// ||int,int| -> int, int| -> int
fn ff(f:|int,int|->int, x:int) -> int
{ f (x, x) }

// m2 : |int| -> int
fn m2(n : int) -> int
{ ff ((|x,y| { x + y }), n) }
```

```ocaml
(* OCaml *)
(* (int*int->b)*int -> int *)
let ff (f, x) =
    f (x, x)

(* m2 : int -> int *)
let m2 n =
    ff ((fun(x,y) -> x + y), n)
```

```haskell
{- Haskell -}
ff :: ((int,int)->int, int) -> int
ff (f, x) =
   f (x, x)

m2 :: Int -> Int
m2 n =
   ff ((\(x,y) -> x + y), n)
```

Traits: Rust's type classes

Rust's "traits" are analogous to Haskell's type classes.

The main difference with Haskell is that traits only intervene for expressions with dot notation, ie. of the form `a.foo(b)`.

For C++/Java/C#/OCaml programmers, however, traits should not be confused with traditional object classes. They are really type classes: it is possible to add traits to arbitrary data types, including the primitive types!

An example:

```rust
// Rust
trait Testable {
    fn test(&self) -> bool
}

impl Testable for int {
    fn test(&self) -> bool {
       if *self == 0 { false }
       else { true }
    }
}

fn hello(x:int) -> bool {
   x.test()
}
```

```haskell
{- Haskell -}
class Testable a where
    test :: a -> Bool


instance Testable Int where
    test n =
       if n == 0 then False
       else True


hello :: Int -> Bool
hello x =
    test x
```

In a trait method declaration, the identifier "`self`" denotes the actual object on which the method is applied.

Like in Haskell, Rust traits can be used for operator overloading. For example, if one defines a new sum type for Peano integers:

```rust
// Rust
enum Peano {
   Zero,
   Succ(Box<Peano>)
}
```

```haskell
{- Haskell -}
data Peano =
    Zero
  | Succ Peano
```

Then one can overload the comparison operator `==` between Peano integers by instantiating the `PartialEq` class:

```rust
// Rust
impl PartialEq for Peano {
    fn eq(&self, other:&Peano) -> bool {
      match (self, other) {
        (&Zero, &Zero)               => true,
        (&Succ(ref a), &Succ(ref b)) => (a == b),
        (_, _)                       => false
      }
    }
}
```

```haskell
{- Haskell -}
instance Eq Peano where
    (==) self other =
        case (self, other) of
          | (Zero, Zero)     -> True
          | (Succ a, Succ b) -> (a == b)
          | (_, _)           -> False
```

Also, like in Haskell, a trait can provide a default implementation for a method, to be used when instances omit the specialization:

```rust
// Rust
trait PartialEq {
   fn eq(&self, other:&Self) -> bool;
   fn ne(&self, other:&Self) -> bool
   { !self.eq(other) }
 }
```

```haskell
{- Haskell -}
class Eq a where
   (==) : a -> a -> Bool
   (!=) : a -> a -> Bool
   (!=) x y = not (x == y)
```

In the method declarations inside a trait declaration, the identifier "`Self`" refers to the actual type on which the trait applies.

Each overloadable operator in Rust has a corresponding trait in the standard library:

Expression	Expands to	Trait	Equivalent Haskell class/method
`a == b`	`a.eq(b)`	std::cmp::PartialEq	`class PartialEq a where (==) : a -> a -> bool`
`a != b`	`a.ne(b)`	std::cmp::PartialEq	`class PartialEq a where (/=) : a -> a -> bool`
`a <  b`	`a.lt(b)`	std::cmp::PartialOrd	`class PartialOrd a where (<) : a -> a -> bool`
`a >  b`	`a.gt(b)`	std::cmp::PartialOrd	`class PartialOrd a where (>) : a -> a -> bool`
`a <= b`	`a.le(b)`	std::cmp::PartialOrd	`class PartialOrd a : Eq a where (<=) : a -> a -> bool`
`a >= b`	`a.ge(b)`	std::cmp::PartialOrd	`class PartialOrd a : Eq a where (>=) : a -> a -> bool`
`a + b`	`a.add(b)`	std::ops::Add<b,c>	`class Add a b c where (+) : a -> b -> c`
`a - b`	`a.sub(b)`	std::ops::Sub<b,c>	`class Sub a b c where (-) : a -> b -> c`
`a * b`	`a.mul(b)`	std::ops::Mul<b,c>	`class Mul a b c where (*) : a -> b -> c`
`a / b`	`a.div(b)`	std::ops::Div<b,c>	`class Div a b c where (/) : a -> b -> c`
`a % b`	`a.rem(b)`	std::ops::Rem<b,c>	`class Rem a b c where (%) : a -> b -> c`
`-a`	`a.neg()`	std::ops::Neg<c>	`class Neg a c where (-) : a -> c`

...continued on next page

Expression	Expands to	Trait	Equivalent Haskell class/method
`!a`	`a.not()`	std::ops::Not<c>	`class Not a c where (!) : a -> c`
`*a`	`a.deref()`	std::ops::Deref<c>	`class Deref a c where (*) : a -> c`
`a & b`	`a.bitand(b)`	std::ops::BitAnd<b,c>	`class BitAnd a b c where (&) : a -> b -> c`
`a | b`	`a.bitor(b)`	std::ops::BitOr<b,c>	`class BitOr a b c where (|) : a -> b -> c`
`a ^ b`	`a.bitxor(b)`	std::ops::BitXor<b,c>	`class BitXor a b c where (^) : a -> b -> c`
`a << b`	`a.shl(b)`	std::ops::Shl<b,c>	`class Shl a b c where (<<) : a -> b -> c`
`a >> b`	`a.shr(b)`	std::ops::Shr<b,c>	`class Shr a b c where (>>) : a -> b -> c`

The `for` loop uses the special trait `std::iter::Iterator`, as follows:

```
// Rust

// the following expression:
[<label>] for <pat> in <iterator expression> {
   body...;
}

// ... expands to (internally):
match &mut <iterator expression> {
  _v => [label] loop {
    match _v.next() {
       None => break,
       Some(<pat>) => { body... }
    }
  }
}
```

The method `next` is implemented by `Iterator`. The return type of `next` is `Option`, which can have values `None`, to mean "nothing left to iterate", or `Some(x)`, to mean the next iteration value is `x`.

Ad-hoc objects and methods

In addition to the mechanism offered by traits, any `struct` or `enum` can be *decorated* with one or more method interface(s) using "`impl`", separately from its definition and/or in different modules:

```
// Rust
struct R {x:int};

// in some module:
impl R {
 fn hello(&self) {
  println!("hello {}", self.x);



 }
}
// possibly somewhere else:
impl R {
 fn world(&self) {
  println!("world");
 }
}
// Example use:
fn main() {
  let v = R {x:10};
  v.hello();
  v.world();

  (R{x:20}).hello();
}
```

```
// Rust
enum E {A, B};

// in some module:
impl E {
 fn hello(&self) {
   match self {
    &A => println!("hello A"),
    &B => println!("hello B")
   }
 }
}
// possibly somewhere else:
impl E {
 fn world(&self) {
  println!("world");
 }
}
// Example use:
fn main() {
  let v = A;
  v.hello();
  v.world();

  B.hello();
}
```

Safe references

Like in C, by default function parameters in Rust are passed by value. For large data types, copying the data may be expensive so one may want to use *references* instead.

For any object `v` of type `T`, it is possible to create a *reference* to that object using the expression "`&v`". The reference itself then has type `&T`.

```
#[derive(Clone,Copy)]
struct Pt { x:f32, y:f32 }

// This works, but may be expensive:
fn dist1(p1 : Pt, p2: Pt) -> f32 {
   let xd = p1.x - p2.x;
   let yd = p1.y - p2.y;
   (xd * xd + yd * yd).sqrt()
}

// Same, using references:
fn dist2(p1 : &Pt, p2: &Pt) -> f32 {
   let xd = p1.x - p2.x;
   let yd = p1.y - p2.y;
   (xd * xd + yd * yd).sqrt()
}

// Usage:
fn main() {
   let a = Pt { x:1.0, y:2.0 };
   let b = Pt { x:0.0, y:3.0 };
   println!("{}", dist1(a, b));
   println!("{}", dist2(&a, &b));
}
```

As illustrated in this example, Rust provides some syntactic sugar for `struct` references: it is possible to

write `p1.x` if `p1` is of type `&Pt` and `Pt` has a field named `x`. This sugar is also available for method calls (`x.foo()`).

However, in many other cases, either the referenced value must be "retrieved" using the unary `*` operator, or patterns must be matched using `&`:

```
fn add3(x : &int) -> int {
   println!("x : {}", *x); // OK
   3 + x; // invalid! (+ cannot apply to &int)
   3 + *x; // OK
}

fn test<t>(x : &Option<t>) -> bool {
   match *x {
     None     => false,
     Some(_)  => true
   }
   // Also valid, equivalent:
   match x {
     &None    => false,
     &Some(_) => true
   }
}
```

Simple references not not allow modifying the underlying object; if mutability by reference is desired, use "`&mut`" as follows:

```
fn incx(x : &int) {
   x = x + 1;   // invalid! (mismatched types in "int& = int")
   *x = *x + 1; // invalid! (*x is immutable)
}

fn inc(x : &mut int) {
   x = x + 1;   // invalid! (x is immutable)
   *x = *x + 1; // OK
}

fn main() {
   inc(&3); // invalid! (3 is immutable)
   inc(&mut 3); // OK, temp var forgotten after call
   let v = 3;
   inc(&v); // invalid! (v is immutable)
   inc(&mut v); // OK, temp var forgotten after call
   let mut w = 3;
   inc(&w); // OK
}
```

Rust's type system *forbids mutable aliases* via references: it is not possible to modify the same object using different names via references, unlike in C. This is done via the concept of **borrowing**: while ownership of an object is borrowed by a reference, the original variable cannot be used any more. For example:

```
fn main() {
   let mut a = 1;
   a = 2; // OK
   println!("{}", a); // OK, prints 2
   // {...} introduces a new scope:
   {
     let ra = &mut a;
     a = 3;   // invalid! (a is borrowed by ra)
     *ra = 3; // OK
     println!("{}", a); // invalid! (a is borrowed)
```

```
        println!("{}", *ra); // OK, prints 3
    };
    // end of scope, rb is dropped.
    println!("{}", a); // OK, a not borrowed any more (prints 3)
}
```

Reference types, together with "all values are immutable by default" and the ownership/borrowing rules, are the core features of Rust's type system that make it fundamentally safer than C's.

Lifetime and storage, and managed objects

> **Note**
> The concepts and syntax presented in this section are new in Rust version 0.11. Rust 0.10 and previous versions used different concepts, terminology and syntax.
> Also, at the time of this writing (July 2014) the official Rust manual and tutorials are not yet updated to the latest language implementation. This section follows the implementation, not the manuals.

Introduction

Rust is defined over the same *abstract machine model* as C. The abstract machine has segments of memory, and the language's run-time machinery can allocate and deallocate segments of memory over time during program execution.

In the abstract machine, the two following concepts are defined in Rust just like in C:

- the *storage* of an object determines the type of memory where the object is stored;
- the *lifetime* or *storage duration* of an object is the segment of time from the point an object is allocated to the point where it is deallocated.

All memory-related problems in C come from the fact that C programs can manipulate references to objects *outside of their lifetime* (ie. before they are allocated or after they are deallocated), or *outside of their storage* (ie. at lower or higher addresses in memory).

Rust goes to great lengths to prevent all these problems altogether, by **ensuring that objects cannot be used outside of their lifetime or their storage**.

Storages in Rust vs. C

There are four kinds of storage in the C/Rust abstract machine: static, thread, automatic and allocated.

Storage	Type of memory used	Example in C
static	process-wide special segment	`static int i;`
automatic	stack	`int i;` (in function scope)
allocated	global heap	`int *i = malloc(sizeof(int));`
thread	per-thread heap	`_Thread_local int i;`

In C, the lifetime of an object is solely determined by its storage:

Storage	Lifetime in C
static	From program start to termination
automatic	From start of scope to end of scope, one per activation frame
allocated	From `malloc` to `free` (or `mmap` to `munmap`, etc)
thread	From object allocation to thread termination

In Rust, the lifetime of static and automatic objects is the same as in C; however:

- Rust introduces a new "box" type with dedicated syntax for heap allocated objects, which are called *managed objects*.

  Rust supports multiple management strategies for boxes, *associated to different typing rules*.

- The default management strategy for boxes ensures that **boxes are uniquely owned**, so that the compiler knows precisely when the lifetime of a box ends and where it can be safely deallocated without the need for extra machinery like reference counting or garbage collection.

- Another strategy is GC, that uses deferred garbage collection: the storage for GC boxes is reclaimed and made available for reuse at some point when the Rust run-time system can determine that they are not needed any more. (This may delay reclamation for an unpredictable amount of time.) References to GC boxes need not be unique, so GC boxes are an appropriate type to build complex data structures with common nodes, like arbitrary graphs.

- Unlike C, **Rust forbids sharing of managed objects across threads**: like in Erlang, objects have a single owner for allocation, and most objects are entirely private to each thread. This eliminates most data races. Also the single thread ownership implies that garbage collection for GC objects does not require inter-thread synchronization, so it is easier to implement and can run faster.

- As with references to normal objects, **Rust forbids mutable aliases** with references to managed objects.

- **All vector accesses are bound checked**, **and references do not support pointer arithmetic**.

- Rust strongly discourages all uses of unmanaged pointers, by tainting them with the `unsafe` attribute which systematically elicits compilers warnings.

Creating and using boxes

The Rust handling of managed objects is relatively simple: the `box` keyword in expressions "puts objects into boxes", where their lifetime is managed dynamically. The immutability and ownership of boxes is handled like with references described earlier:

```
fn main() {

  let b = box 3i; // b has type Box<int>
  b = 4i; // invalid! can't assign int to box<int>
  *b = 4i; // invalid! b is immutable, so is the box
  let mut c = box 3i; // c has type mut Box<int>
  *c = 4i; // OK

  let v = box vec!(1i,2,3); // r1 has type Box<Vec<int>>
```

```
    *v.get_mut(0) = 42; // invalid! v is immutable, so is the box
    let mut w = box vec!(1i,2,3);
    *w.get_mut(0) = 42; // OK, rust 0.11.1+ also permits w[0] = 42

    let z = w; // z has type Box<Vec<int>>, captures box
    println!("{}", w); // invalid! box has moved to z

    w = box vec!(2i,4,5); // overwrites reference, not box contents
    println!("{} {}", w, z); // OK, prints [2,4,5] [42,2,3]
}
```

The `box` keyword in expressions is actually a shorthand form for `box(HEAP)`. The general form "`box(A) E`" places the result of the evaluation of `E` in a memory object allocated by `A`, which is a trait. As of Rust 0.11, the only other allocator in the standard library is `GC`, for garbage collected objects.

Expression	Type	Meaning
`box v`	Box<T>	Unique reference to a copy of `v`, shorthand for `box(HEAP) v`
`box(GC) v`	Gc<T>	Garbage-collected smart pointer to a copy of `v`
`box(A) v`	??<T>	Some kind of smart pointer; `A` must implement a special trait.

Recursive data structures

Managed objects are the "missing link" to implement proper recursive algebraic data types:

```
// Rust
enum Lst<t> {
   Nil,
   Cons(t, Box<Lst<t>>)
}
fn len<t>(l : &Lst<t>) -> uint {
   match *l {
     Nil => 0,
     Cons(_, ref x) => 1 + len(*x)
   }
}

fn main() {
   let l = box Cons('a',
            box Cons('b',
             box Nil));
   println!("{}", len(l));
}
```

```
(* OCaml *)
type 't lst =
    Nil
  | Cons of 't * 't lst

let len l =
  match l with
  | Nil -> 0
  | Cons(_, x) -> 1 + len x

let l = Cons('a',
          Cons('b',
              Nil));

print_int (len l)
```

```
{- Haskell -}
data Lst t =
    Nil
  | Cons t (Lst t)

let len l =
  case l of
    Nil -> 0
    Cons _ x -> 1 + len x

main = do
         let l = Cons 'a'
                      (Cons 'b'
                            Nil)

         putStrLn $ show l
```

Shared objects: Rc and Arc

In complement to the facilities offered by `box`, the Rust standard library also implements two implementations of **reference counted wrappers to immutable objects** that can be referred to from multiple owners:

```
use std::rc::Rc;
fn main() {
    let b = Rc::new(3i);
    let c = &b;
    println("{} {}", b, c); // OK
}
```

The use of reference counts ensures that an object gets deallocated exactly when its last reference is dropped. The trade-off with boxes is that the reference counter must be updated every time a new reference is created or a reference is dropped.

Two implementations are provided: `std::rc::Rc` and `std::arc::Arc`. Both offer the same interface. The reason for this duplication is to offer a controllable trade-off over performance to programmers: `Rc` does not use memory atomics, so it is more lightweight and thus faster, however it cannot be shared across threads. `Arc` does use atomics, is thus slightly less efficient than `Rc`, but can be used to share data across threads.

Macros and meta-programming

The basic syntax of a macro definition is as follows:

```
macro_rules! MACRONAME (
    (PATTERN...) => (EXPANSION...);
    (PATTERN...) => (EXPANSION...);
    ...
)
```

For example, the following macro defines a Pascal-like `for` loop:

```
macro_rules! pfor (
   ($x:ident = $s:expr to $e:expr $body:expr)
       => (match $e { e => {
              let mut $x = $s;
              loop {
                   $body;
                   $x += 1;
                   if $x > e { break; }
              }
         }});
);

// Example use:
fn main() {
     pfor!(i = 0 to 10 {
         println!("{}", i);
     });
}
```

Note how this macro uses the `match` statement to assign a local name to an expression, so that it does not get evaluated more than once.

Like in Scheme, macros can be recursive. For example, the following macro uses recursion to implement `pfor` both with and without `step`:

```
macro_rules! pfor (
   ($x:ident = $s:expr to $e:expr step $st:expr $body:expr)
       => (match $e,$st { e, st => {
              let mut $x = $s;
              loop {
                   $body;
                   $x += st;
                   if $x > e { break; }
              }
         }});
   ($x:ident = $s:expr to $e:expr $body:expr)
      => (pfor!($x = $s to $e step 1 $body));
```

```
);

// Example use:
fn main() {
     pfor!(i = 0 to 10 step 2 {
         println!("{}", i);
     });
}
```

Macros can also be variadic, in that arbitrary repetitions of a syntax form can be captured by one macro argument. For example, the following macro invokes `println!` on each of its arguments, which can be of arbitrary type:

```
macro_rules! printall (
     ( $( $arg:expr ),*  ) => (
          $( println!("{}", $arg) );*
     );
);

// example use:
fn main() {
    printall!("hello", 42, 3.14);
}
```

The syntax works as follows: on the left hand side (pattern) the form `$( PAT )DELIM*` matches zero or more occurrences of PAT delimited by `DELIM`; on the right hand side (expansion), the form `$( TEXT )DELIM*` expands to one or more repetitions of `TEXT` separated by `DELIM`. The number of repetitions of the expansion is determined by the number of matches of the enclosed macro argument(s). In the example, each argument (separated by commas) is substituted by a corresponding invocation of `println!`, separated by semicolons.

Literals

Rust provides various lexical forms for number literals:

Rust syntax	Same as	Type	Haskell equivalent	OCaml equivalent
`123i`		int	`123 :: Int`	`123`
`123u`		uint	`123 :: Word`	
`123i8`		i8	`123 :: Int8`	
`123u8`		u8	`123 :: Word8`	`Char.chr 123`
`123i16`		i16	`123 :: Int16`	
`123u16`		u16	`123 :: Word16`	
`123i32`		i32	`123 :: Int32`	`123l`
`123u32`		u32	`123 :: Word32`	
`123i64`		i64	`123 :: Int64`	`123L`
`123u64`		u64	`123 :: Word64`	
`1_2_3_4`	`1234`	(integer)		
`1234_i`	`1234i`	int		
`0x1234`	`4660`	(integer)	`0x1234`	`0x1234`

...continued on next page

Rust syntax	Same as	Type	Haskell equivalent	OCaml equivalent
`0x1234u16`	`4660u16`	u16	`0x1234 :: Word16`	
`0b1010`	`10`	(integer)		`0b1010`
`0o1234`	`668`	(integer)	`0o1234`	`0o1234`
`b'a'`	`97u8`	u8		`'a'`
`b"a"`	`[97u8]`	[u8]		
`12.34`		(float)	`12.34`	
`12.34f32`		f32	`12.34 :: Float`	
`12.34f64`		f64	`12.34 :: Double`	`12.34`
`12e34`	`1.2e35`	(float)	`12e34`	
`12E34`	`1.2e35`	(float)	`12E34`	
`12E+34`	`1.2e35`	(float)	`12E+34`	
`12E-34`	`1.2e-33`	(float)	`12E-34`	
`1_2e34`	`12e34`	(float)		
`1_2e3_4`	`12e34`	(float)		

Escapes in character, byte and string literals:

Syntax	Same as	Syntax	Same as	Syntax	Same as
`'\x61'`	`'a'`	`b'\x61'`	97u8	`"\x61"`	`"a"`
`'\\'`	`'\x5c'`	`b'\\'`	92u8	`"\\"`	`"\x5c"`
`'\''`	`'\x27'`	`b'\''`	39u8	`"\""`	`"\x22"`
`'\0'`	`'\x00'`	`b'\0'`	0u8	`"\0"`	`"\x00"`
`'\t'`	`'\x09'`	`b'\t'`	9u8	`"\t"`	`"\x09"`
`'\n'`	`'\x0a'`	`b'\n'`	10u8	`"\n"`	`"\x0a"`
`'\r'`	`'\x0d'`	`b'\r'`	13u8	`"\r"`	`"\x0d"`
`'\u0123'`				`"\u0123"`	
`'\U00012345'`				`"\U00012345"`	

Note that the other common escapes in the C family (`\a`, `\f`, etc.) are not valid in Rust, nor are octal escapes (eg. `\0123`).

Finally, Rust like Python supports raw strings and multiple string delimiters, to avoid quoting occurrences of the delimiters within the string:

Syntax	String value	Syntax	Value
`"foo"`	`foo`	`b"foo"`	`[102u8,111,111]`
`"fo\"o"`	`fo"o`	`b"fo\"o"`	`[102u8,111,34,111]`
`r"fo\n"`	`fo\n`	`rb"fo\n"`	`[102u8,111,92,110]`
`r#"fo\"o"#`	`fo\"o`	`rb#"fo\"o"#`	`[102u8,111,92,34,111]`
`"foo#\"#bar"`	`foo#"#bar`	`b"foo#\"#bar"`	`[102u8,111,111,35,34,35,98,97,114]`

...continued on next page

Syntax	String value	Syntax	Value
`r##"foo#"#bar"##`	`foo#"#bar`	`rb##"foo#"#bar"##`	`[102u8,111,111,35,34,35,98,97,114]`

Acknowledgements

Many thanks to the numerous commenters on Reddit and Hacker news who provided high-quality comments and substantively contributed to improving upon the first version of this article.

References

- The Rust Language Tutorial, version 0.10, April 2014.
- The Rust Language Tutorial, version 0.11.0, July 2014.
- The Rust Guide, version 0.11.0, July 2014.
- Aaron Turon, Rust Guidelines, 2014.
- Will Yager. Why Go is not good, 2014. Explains how Go lacks many features found in Rust and thereby fails to be a modern functional language.
- Edward Z. Yang, OCaml for Haskellers, October 2010.
- Raphael Poss, Haskell for OCaml programmers, March 2014.
- Xavier Leroy et al., The OCaml system release 4.01, September 2013.

Copyright and licensing

Copyright © 2014-2026, Raphael Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.