Rust: system programming with guarantees

CRI Monthly Seminar

Arnaud Spiwack Pierre Guillou

MINES ParisTech, PSL Research University

Fontainebleau, July 6th, 2015

High level programming languages

> *A programming language is low level when its programs require attention to the irrelevant.*
>
> – Alan Perlis

> *A language that doesn't affect the way you think about programming, is not worth knowing.*
>
> – also Alan Perlis

High level on the rise

Mozilla

- C++ $\Longrightarrow$ Rust

Apple

- Objective-C $\Longrightarrow$ Swift

Microsoft

- C♯ $\Longrightarrow$ F♯

From Academia

- Scala (Twitter, . . .)
- Ocaml (Facebook, . . .)
- Haskell (Facebook, . . .)

Rust in a nutshell

Rust is

- Strongly typed
- Memory safe
- Garbage-collector free
- Higher order
- Precise wrt. memory

Rust isn't

- Always easy
- For everything
- Finished

Ownership, linearity

```
let x = &mut 42;
let y = x;
println!("Value: {}.", x);
```

error: use of moved value: ‘x‘

Ownership, linearity

```
let x = &mut 42;
let y = x;
println!("Value: {}.", y); // was `x`
```

```
Value: 42.
```

Ownership, linearity

```
let x = &mut 42;
let y = x.clone(); // was ‘x’
println!("Value: {}.", x); // was ‘y’
```

```
Value: 42.
```

Datatypes

```
enum List<A> {
  Nil,
  Cons(A,List<A>)
}
```

error: illegal recursive enum type; wrap the inner value in a box to make it representable

Datatypes

```
enum List<A> {
  Nil,
  Cons(A,List<A>) // unboxed!
}
```

error: illegal recursive enum type; **wrap the inner value in a box** to make it representable

Datatypes

```
enum List<A> {
  Nil,
  Cons(A,Box<List<A>>)
}
```

Works fine but cloning is linear time.

Reference counting

```
enum List<A> {
  Nil,
  Cons(A,Rc<List<A>>)
}


fn append<A:Clone>(l1:Rc<List<A>>,l2:Rc<List<A>>) -> Rc<List<A>> {
  match *l1 {
    List::Cons(ref a,ref t) =>
      Rc::new( List::Cons( a.clone() , append(t.clone() , l2) )),
    List::Nil =>
      l2
  }
}
```

Reference counting

```
enum List<A> {
  Nil,
  Cons(A,Rc<List<A>>)
}

fn append<A:Clone>(l1:Rc<List<A>>,l2:Rc<List<A>>) -> Rc<List<A>> {
  // definition omitted
}


let l1 = Rc::new((1..5).collect()); // build list using an iterator
let l2 = Rc::new((5..10).collect());
println!("append: {}", append(l1,l2))
```

```
append:  [1,2,3,4,5,6,7,8,9]
```

Speaking about iterators

```
let l = (1..10).collect();
println!("Iterator: {}", l)
```

```
Iterator: [1,2,3,4,5,6,7,8,9]
```

Some type annotations have been omitted

Speaking about iterators

```
let nat = 0..; // the natural numbers
let ten = nat.take(10); // take the 10 first
let l = ten.collect();
println!("Iterator: {}", l)
```

```
Iterator: [0,1,2,3,4,5,6,7,8,9]
```

Some type annotations have been omitted

Speaking about iterators

```
let nat = 0..; // the natural numbers
let odd = nat.filter(|n|  n%2==1); // only the odd ones
let ten = odd.take(10); // take the 10 first
let l = ten.collect();
println!("Iterator: {}", l)
```

```
Iterator:  [1,3,5,7,9,11,13,15,17,19]
```

Some type annotations have been omitted

Speaking about iterators

```
let nat = 0..; // the natural numbers
let odd = nat.filter(|n| n%2==1); // only the odd ones
let pairs = odd.flat_map(|n| // cartesian product
   vec!(true,false).into_iter().map(move |p| (n,p)));
let ten = pairs.take(5); // take the 5 first
let l = ten.collect();
println!("Iterator: {}", l)
```

```
Iterator: [(1,true),(1,false),(3,true),(3,false),(5,true)]
```

Some type annotations have been omitted

Back to lists

```
enum List<A> {
  Nil,
  Cons(A,Box<List<A>>)
}
```

Power of ownership

Modify value $\cong$ Make new value

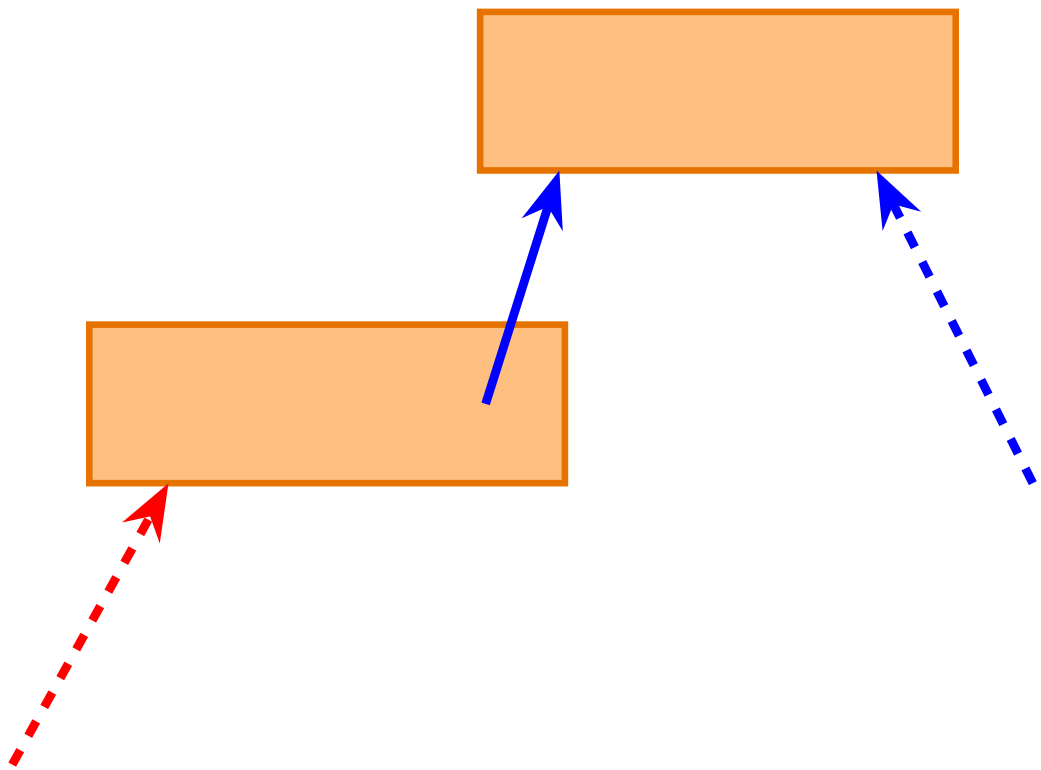
```
fn append<A>(l1:&mut List<A>,l2:List<A>) {
  match *l1 {
    List::Cons(_,ref mut t) => append(&mut *t,l2),
    List::Nil => *l1 = l2
  }
}
```

Remark: non-aliasing

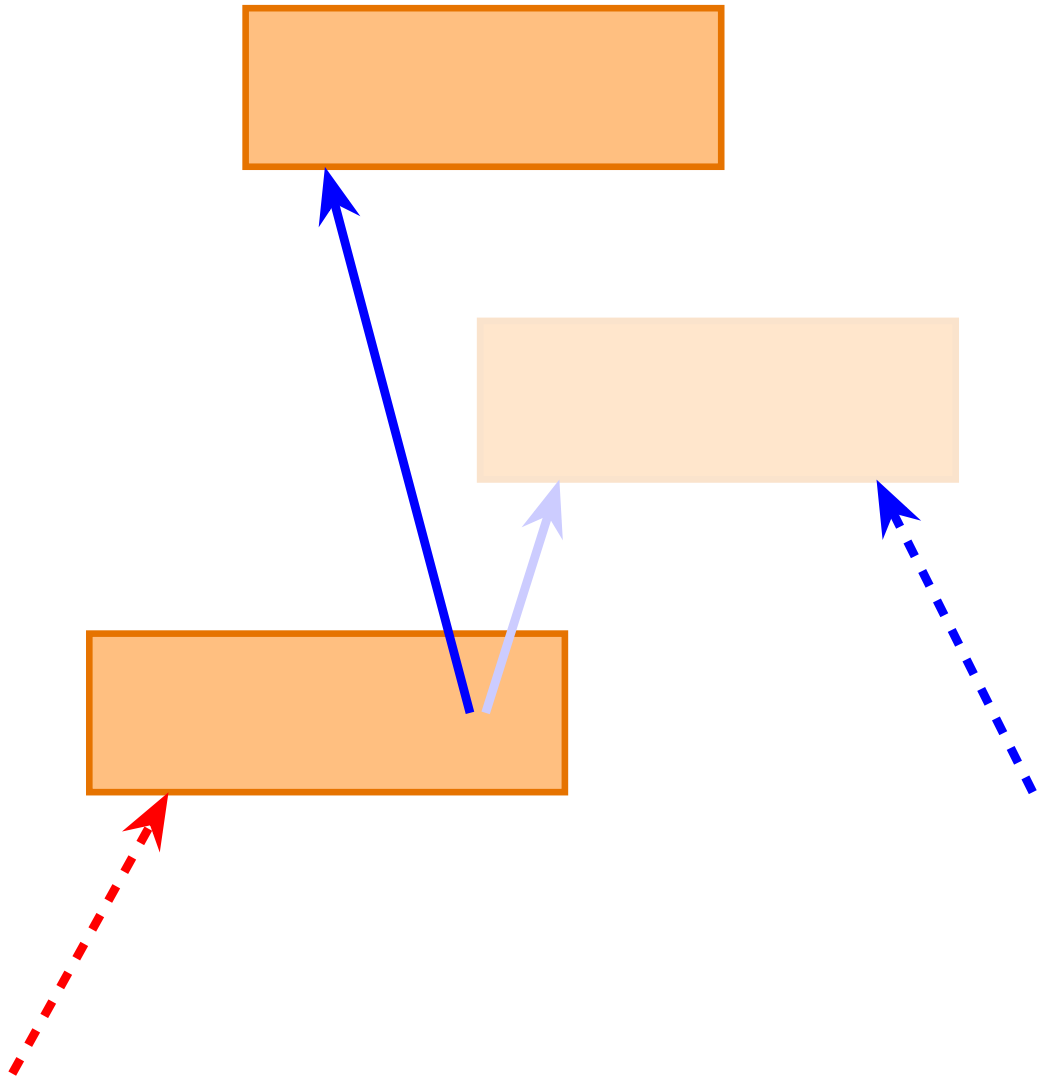
```
let mut x = List::Nil;
let y = &mut x;
let z = & x;
```

error: cannot borrow ‘x‘ as immutable **because it is also borrowed as mutable**

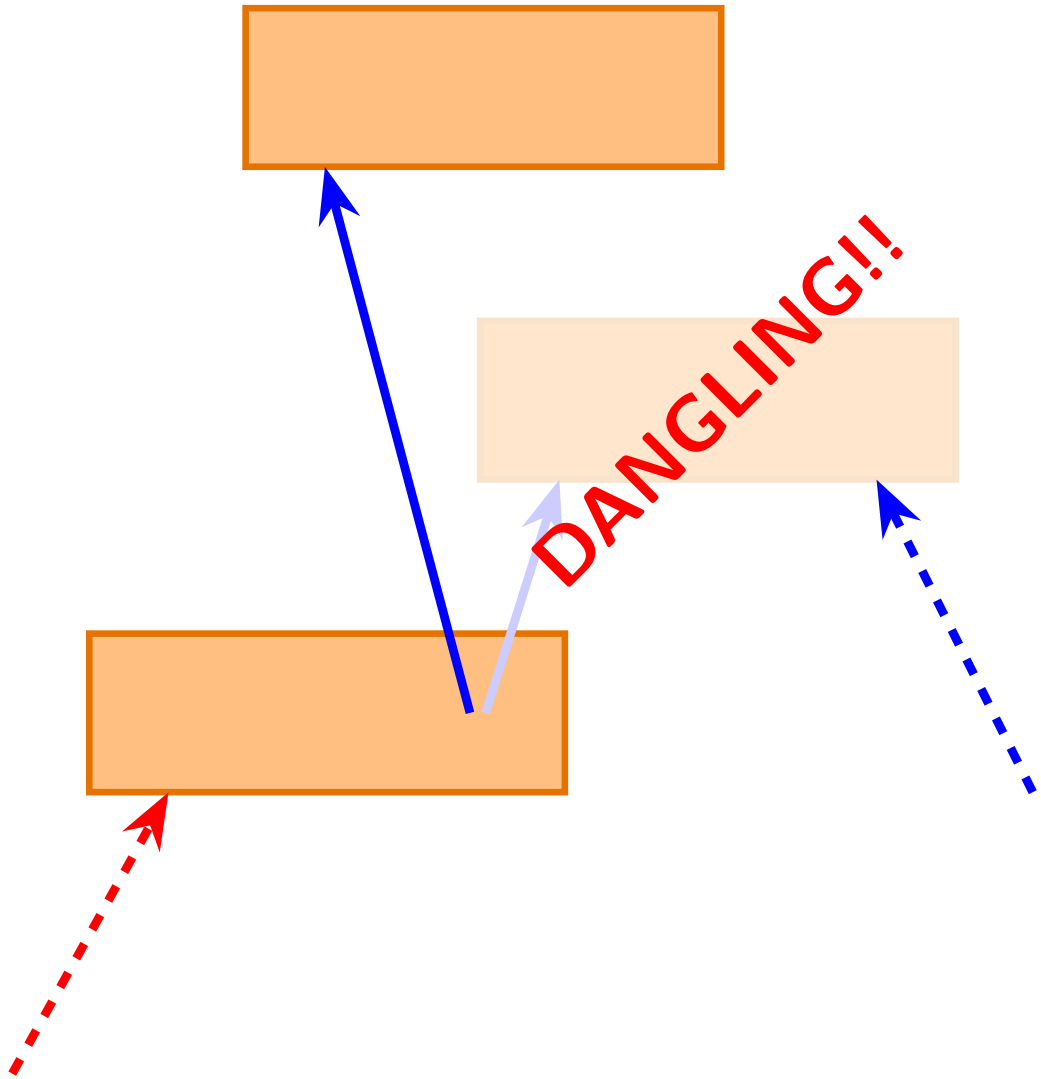
Non-aliasing continued



Non-aliasing continued



Non-aliasing continued



Lifetimes

```
let mut x = List::Nil;
let y = &mut x;
let z = & x;
```

error: cannot borrow ‘x‘ as immutable **because it is also borrowed as mutable**

Some type annotations have been omitted

Lifetimes

```
let mut x = List::Nil;
{ let y = &mut x;}
let z = & x;
```

ok

Some type annotations have been omitted

Lifetimes

```
let mut x = List::Nil;
let z = &mut &x;
{ let y = List::Nil;
  let w = & y;
  *z = w; }
```

error: ‘y‘ **does not live long enough**

Some type annotations have been omitted

Ownership passing

```
enum List<A> {
  Nil,
  Cons(A,Box<List<A>>)
}


fn append<A>(l1:&mut List<A>,l2:List<A>) {
  // definition omitted
}


// This type works on the functional version as well
fn append2<A>(l1:List<A>,l2:List<A>) -> List<A> {
  append(&mut l1,l2);
  l1
}
```

Abstraction: type scope

```
impl<A> List<A> {
  /// invoked as `List::foo(...)`
  fn foo (...) -> ... {
    // definition
  }

  /// invoked as `l.foo(...)`
  fn bar(&self,...) -> ... {
    // definition
  }
}
```

Abstraction: traits

```
/// Allows lists to be used as a target of the `collect` method
impl<A> FromIterator<A> for List<A> {

  // required methods go here

}

/// Allows lists to be used with format macros (such as `println!`)
impl<A:Display> Display for List<A> {

  // required methods go here

}
```

Traits: remember

```
enum List<A> {
  Nil,
  Cons(A,Rc<List<A>>)
}


// `Clone` is a trait
fn append<A:Clone>(l1:Rc<List<A>>,l2:Rc<List<A>>) -> Rc<List<A>> {
  match *l1 {
    List::Cons(ref a,ref t) =>
      Rc::new( List::Cons( a.clone() , append(t.clone() , l2) )),
    List::Nil =>
      l2
  }
}
```

Recap

Rust has

- ownership, borrowing, lifetimes
- unboxed datatypes, pattern-matching
- traits
- iterators

And so much more

- array slices, unboxed functional values
- statically sized data in stack *vs* dynamically sized data in heap
- lifetime-parametric data & functions
- trait objects
- typed exceptions (sort of)
- macros, iterator-based for loop
- safer concurrency

And, also, a vibrant ecosystem

- compilation, standard library
- documentation, test
- some projects written in Rust

Modules and Crates

Crate compilation unit

- library or binary
- link: **extern** crate other_crate;

Module named scope

- a crate = several modules
- visibility: **pub** keyword
- a tree structure from the top-level source directory
- **use** keyword to import into the local scope and defining public interfaces
- no need to maintain separate .h/.mli files:

```
// declare external interface
// independantly of internal code organization
pub mod my_interface {
  // import all my_internal_module functions
  // into current module
  pub use my_internal_module;
}
```

Compiling Rust programs

The Rust compiler

- *rustc*
- hardware targets: (x86, ARM) $\times$ (Linux, OSX, Windows, *BSD)
- built on top of LLVM
- written in Rust (previously in Ocaml)
- compiling rather slow, optimization on their way

Conditional compilation

- code annotations *#(cfg(feature))*, macros cfg!(feature)
- compiler flags: `--cfg feature`

Linking to other libraries

Linking to a Rust crate

- **extern** crate declaration to import public items
- `$ rustc -L` *`path`* `-l` *`lib`* to add library

Linking to a C library

```
#(link(name="m"))
extern { // signatures
    fn pow(a: f64, b: f64) -> f64;
}

pub fn rust_pow(a: f64, b: f64) -> f64 {
    unsafe {
        pow(a, b)
    }
}
```

Calling Rust from C (or Python, C++, . . .)

- *#(no_mangle)* and **extern fn** for functions
- *#(repr(C))* for structs

Rust standard libraries

The `std` crate

- 55 kLoC (glibc: 1.8 MLoC)
- automatically linked at compile time
- except with *#(no_std)* $\longrightarrow$ bare metal
- basic data structures: strings, vectors, hashmaps, error handling
- I/O (files, fs, path), concurrency (threads, channels, processes, simd)

The `core` crate

- 25 kLoC
- minimal and portable replacement for the `std` crate
- no heap allocation, no concurrency, no I/O

The `libc` crate

- wrapper around LibC functions (glibc on Linux)

Documentation, test

Documentation

- can be written in source code using /// delimiters
- Markdown syntax
- `$ rustdoc` to generate HTML
- code examples in documentation can be compiled and executed as tests

Test

- *#(test)* annotation conditionnaly compiled with `$ rustc --test`
- assert! macros for testing conditions
- anywhere in source code, even in documentation
- also support performance tests with *#(bench)*

```
#(test)
fn test_rust_pow() {
    assert_eq!(rust_pow(2.0, 3.0), 8.0);
}
```

Cargo, a project manager for Rust

- `$ cargo new` *`project_name`* `[--bin]`
  - create a folder hierarchy: *`project_name`*`/{src, target}`
  - initialize git/hg version control repository
- `$ cargo run`: compile and run in one command
- `$ cargo [test|bench]`: compile and run tests/benchmarks
- `$ cargo doc`: generate documentation
- `Cargo.toml`: semantic versionning, metadata, dependencies...
- `http://crates.io`: package repository

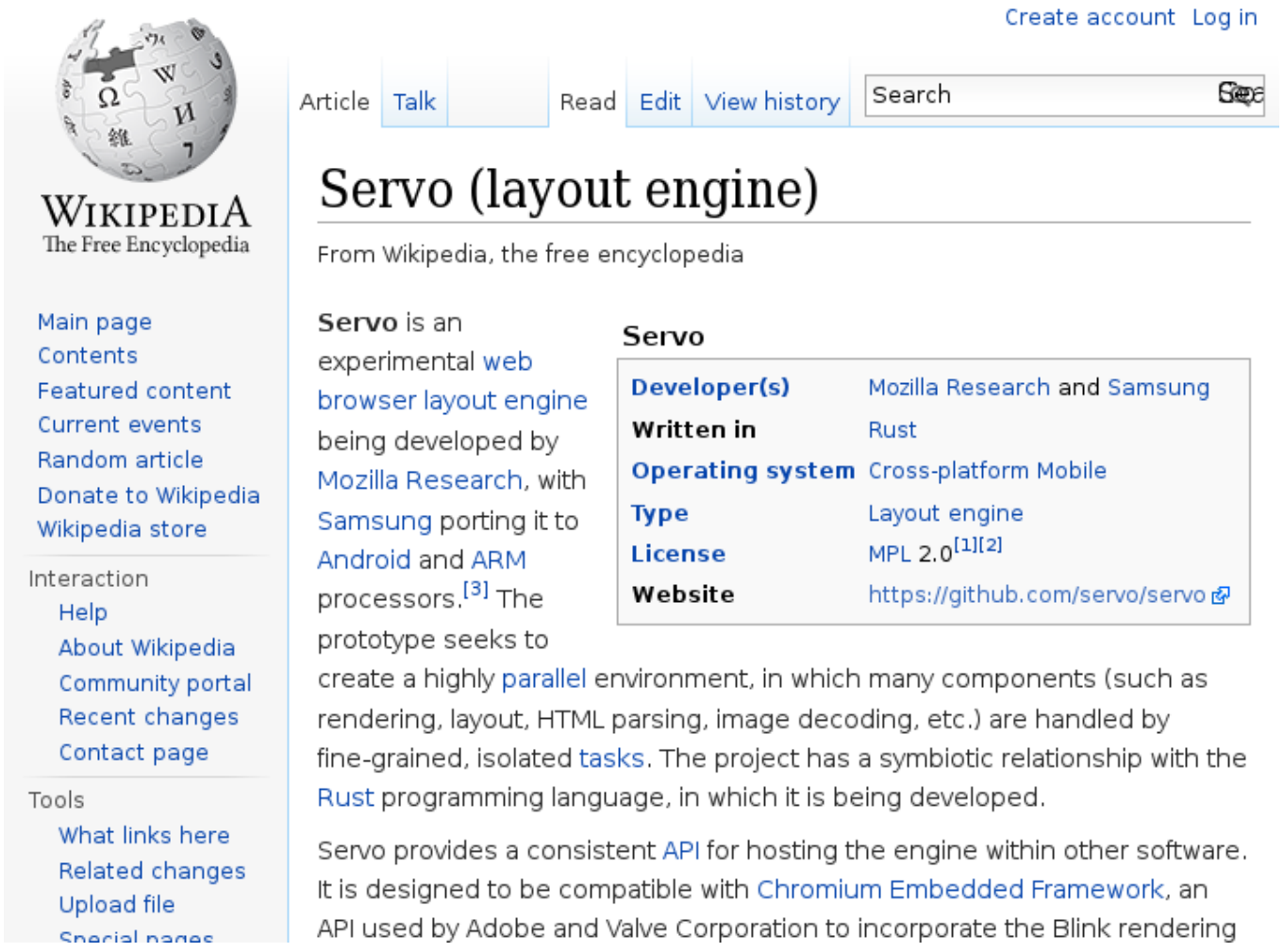
Projects written in Rust

- the Rust compiler and the standard library
  - currently 85 kLoC in Rust
  - 6-weeks iteration process (like Firefox, Chrome)
  - on GitHub

- *cargo*, the Rust project manager
  - 15 kLoC in Rust
  - 2500 available crates on `http://crates.io`

- *Servo*, a parallel web browser layout engine
  - research project
  - aims at replacing the *Gecko* engine
  - already passes the Acid2 rendering test
  - currently 150 kLoC in Rust

Servoception




```
servo -o wikiservo.png --resolution 720x540 https://en.wikipedia.org/wiki/Servo_\(layout\_engine\)
```

On-line resources

- main page: `http://rust-lang.org`
- guides: Rust Book, Rust by Example
- Reddit, Stack Overflow, GitHub, IRC, Twitter, ...
- linkable online interpreter: Rust playground

Conclusion

Rust

- innovative programming language
- high-level features
- fine-grained memory model
- hardware performance
- efficient tools for developers
- interesting projects
- thriving community

Rust: system programming with guarantees

CRI Monthly Seminar

Arnaud Spiwack Pierre Guillou

MINES ParisTech, PSL Research University

Fontainebleau, July 6th, 2015