Rust

efficient, safe and concurrent

Michael Neumann / mneumann@ntecs.de

Why Rust?

and not *D*, *Java* or *Go*?

Because they **cannot** replace C++!

Currently C++ is the only option when it comes to *performance sensitive, reactive* applications

Application domains of C++

Web browsers

Firefox (C++)

Chrome (C++)

Internet Explorer (C++)

Opera (C++)

Safari (C++)

...

Games

Is there a single, graphical-intensive, commercial game *not* written in C++?

System software

JVM Runtime (C/C++)

LLVM/clang (C++)

gcc (C++)

Perl, Ruby, Python, PHP... (C/C++)

Operating Systems (C/C++)

*As **ugly** as it is, C++ is still what you use when you want a fast application and full control over the code.*

C++ is unsafe

NULL pointer exceptions (segfaults)

Buffer-overruns (out of bounds)

dangling-pointers, double-frees, memory leaks

Implicit type casts

Non-initialized variables (especially within classes)

C++ is error-prone

`if (a = NULL)`

`if (a & 0xFF == 0)`

`switch` "fallthrough"

missing `virtual`

The problem

D, go, ...

Non-deterministic behaviour of memory allocations

Almost unusable *without* the garbage collector

Almost every object is subject of garbage collection

Garbage collector does *not* scale

The solution

Static lifetimes...

```
// C++
{
  string s = "Hello World";
  vector<int> v{1,2,3};
  {
    cout << s << endl;
  }
} // deallocation of s and v
```

```
// Rust
{
  let s = ~"Hello World";
  let v = ~[1, 2, 3];
  {
    io::println(s);
  }
} // deallocation of s and v
```

Objects are allocated on the heap. No GC, because lifetime is known at compile-time.

... and references

```cpp
// C++
void f(const string &s) {
  cout << s << endl;
}

void main() {
  string s = "Hello World";
  f(s);
}
```

```rust
// Rust
fn f(s: &str) {
  io::println(s);
}

fn main() {
  let s = ~"Hello World";
  f(s);
}
```

Similar to pointers, but: References are guaranteed to be non-null, variables are non-assignable and with limited life-time.

Unique pointers

Pointer type with a *single* alias.

```
{
  let i = ~999u; // declares and allocates unique integer pointer
  let j = i;     // moves pointer into variable "j"
  io::println((*j).to_str()); // 999
  io::println((*i).to_str()); // compile error: "use of moved value i"
} // deallocation
```

```
{
  std::unique_ptr<int> i(new int(999));
  auto j = std::move(i);
  cout << *j << endl; // 999
  cout << *i << endl; // Segmentation Fault
} // deallocation
```

Manual deallocation

```
{
  let i = ~999u; // declares and allocates unique integer pointer
  {
    let j = i;    // moves pointer into variable "j"
  } // deallocation
  io::println((*i).to_str()); // compile error: "use of moved value i"
}
```

Unique pointers in *Rust*

Aka. owned-pointer

Heap allocated

Checked at compile-time (no segfaults as in C++)

Deallocated at the end of it's life-time (scope)

Besides references, most common pointer type in Rust

Can be sent (moved) to other tasks efficiently

Managed pointers in *Rust*

```
// Rust
let i = @[ @[1], @[2,3,4], @[5u] ];
let j = i;
io::println(i.to_str()); // [[1],[2,3,4],[5]]
io::println(j.to_str()); // [[1],[2,3,4],[5]]
io::println(i[1].to_str()); // [2,3,4]
```

Reference counted or garbage collected

Allocates on *task-local* heap (!!!)

Has to be copied when sent to other tasks

Rarely used in Rust

Raw pointers in *Rust*

```
// Rust
let mut i = 0;
unsafe {
  let r: *mut int = &mut i;
  *r = 999;
}
io::println(i.to_str()); // 999
```

Identical to regular pointers in C/C++

Used for interaction with C/C++ or runtime implementation

References in *Rust*

Aka. borrowed pointer

Valid only as long as referenced object is valid

Scope known at compile-time

Slicing `&str` and `&[]`

```
let s = ~"Hello World";
let _world: &str = str::slice(s, 5, s.len()); // " World"
let world = str::trim_left(s);
io::println(world); // "World"
```

References to *strings* and *arrays* include besides the raw pointer to the data also the length ("fat-pointer").

Freezing

```
let mut s = ~"Hello";
s.push_str(" ");
{
  let ell = s.slice(1, 4);
  io::println(ell); // "ell"
  s.push_str("World"); // COMPILE ERROR!
}
s.push_str("World"); // this works!!!

io::println(s); // "Hello World"
```

Referencing a mutable object *freezes* it within the scope of that reference.

Important because `push_str()` might have to allocate a *new* string in case it reaches it's capacity.

Highly superior over the non-deterministic behaviour of for example slices in D.

Lifetimes of references

```
fn split2<'a>(s: &'a str, delim: char) -> (&'a str, Option<&'a str>) {
  match str::find_char(s, delim) {
    None => (s, None),
    Some(p) => (str::slice(s, 0, p), Some(str::slice(s, p+1, str::len(s)
  }
}

#[test]
fn test_split2() {
  assert_eq!(split2("abc.def", '.'), ("abc", Some("def")) );
  assert_eq!(split2("abc", '.'),     ("abc", None) );
}
```

Sometimes you have to annotate life-times of references via `'name` notation.

Further features in *Rust*

Built-in unit testing

```
// a.rs
#[test]
fn test_trim_left() {
  assert_eq!(str::trim("   trim   ", "trim   ");
}

#[test]
fn test_trim() {
  assert_eq!(str::trim("   trim   "), "trim");
}
```

```
# rust test a.rs
running 2 tests
test test_trim_left ... ok
test test_trim ... ok

result: ok. 2 passed; 0 failed; 0 ignored
```

Generic data types

```
struct<T> Container {
  arr: ~[T]
}

impl<T> Container<T> {
  fn size(&self) -> uint { self.arr.len() }
}

fn main() {
  let a = Container{ arr: ~[ ~"a", ~"b" ] };
  let sz = a.size();
}
```

Traits

```
trait ToStr {
  fn to_str(&self) -> ~str;
}

struct Foo {bar: uint}

impl ToStr for Foo {
  fn to_str(&self) -> ~str {
    ~"Foo{bar: " + self.bar.to_str() + ~"}"
  }
}

fn main() {
  let foo = Foo{bar: 42};
  io::println(foo.to_str()); // "Foo{bar: 42}"
}
```

Similar to *interfaces* or Haskell's *type classes*

Generic traits

```
trait Seq<T> {
  fn len(&self) -> uint;
  fn iter(&self, b: &fn(v: &T));
}

impl<T> Seq<T> for (T, T, T) {
  fn len(&self) -> uint { 3 }
  fn iter(&self, b: &fn(v: &T)) {
    match *self {
      (ref x, ref y, ref z) => {
        b(x); b(y); b(z)
      }
    }
  }
}

fn main() {
  do (4u, 5, 6).iter |i| {
```

Serialization

```
extern mod std; // link with library `std'
use std::serialize::*;

struct Foo {bar: uint}

impl<S: Encoder> Encodable<S> for Foo {
  fn encode(&self, s: &S) {
    do s.emit_struct("Foo", 1) {
      do s.emit_field("bar", 0) {
        s.emit_uint(self.bar)
      }
    }
  }
}

fn main() {
  let foos = [Foo{bar: 42}, Foo{bar: 24}];
  let bytes = do io::with_bytes_writer |wr| {
```

std::serialize

Abstracts from concrete data format: **JSON**, **MsgPack**, ...

Automatic code generation

```
extern mod std; // link with library `std'
use std::serialize::*;

#[auto_encode]
struct Foo {bar: uint}

fn main() {
  let foos = [Foo{bar: 42}, Foo{bar: 24}];
  let bytes = do io::with_bytes_writer |wr| {
    let encoder = std::json::Encoder(wr);
    foos.encode(&encoder);
  };
  io::println(str::from_bytes(bytes)); // [{"bar":5},{"bar":9}]
}
```

Deriving

```
#[deriving(Eq, Ord)]
struct Foo {bar: uint}

//
// Instead of:
//
impl Eq for Foo {
  fn eq(&self, o: &Self) -> bool { self.bar == o.bar }
}
impl Ord for Foo {
  fn lt(&self, o: &Self) -> bool { self.bar < o.bar }
}
```

Tasks

`do task::spawn { ... }` starts task

```
for int::range(0, 10) |i| {
  do task::spawn {
    io::println(fmt!("Task %d", i));
  }
}
```

Tasks

Light-weight

Cooperative

Dynamic *split stack*

Separate adress space

Task-local GC

Exception handling with tasks

```
let res = do task::try {
  ...
  fail!();
  ...
};
```

`fail!(...)` destroys the task.

Can be caught via `task::try`.

Example

```
fn do_something() {
  fail!(~"exception")
}

fn main() {
  let res = do task::try {
    do_something()
  };

  if res.is_ok() {
    io::println("oK")
  }
  else {
    io::println("failed")
  }
}
```

Task Linking Modes

`spawn_unlinked`

`spawn_linked`

`spawn_supervised`

hierarchical

Unidirectional pipes

```
fn main() {
  let (port, chan) = comm::stream();

  do task::spawn {
    chan.send(~"Hello World")
  }

  io::println(port.recv());
}
```

Very efficient due to *unique pointers*!

Bidirectional pipes

```
extern mod std; use std::comm::DuplexStream;
fn main() {
  let (p1, p2) = DuplexStream();

  do task::spawn {
    loop {
      let num: int = p1.recv();
      p1.send(num + 1);
      if num == 0 { break }
    }
  }

  p2.send(10);
  io::println(p2.recv().to_str()); // 11
  p2.send(0);
  let _ = p2.recv();
}
```

RPC - oneshot pipes

```
fn main() {
  let (port, chan) = comm::stream();

  do task::spawn {
    loop {
      let (num, reply_chan): (int, comm::ChanOne<int>) = port.recv();
      reply_chan.send(num + 1);
      if num == 0 { break }
    }
  }

  let (reply_port, reply_chan) = comm::oneshot();
  chan.send((1, reply_chan));
  io::println(reply_port.recv().to_str());

  // ...
}
```

Pipes benchmark

Multiple sender, one receiver

Language	Messages per second	Comparison
Rust port_set	881,578	232.8%
Scala	378,740	100.0%
Rust port/chan (updated)	227,020	59.9%
Rust shared_chan	173,436	45.8%
Erlang (Bare)	78,670	20.8%
Erlang (OTP)	76,405	20.2%

Link

Protocol *enforcement*

```
proto! pingpong (
    ping: send {
        ping -> pong
    }
    pong: recv {
        pong -> ping
    }
)
```

Module pipes

Tasks and I/O

Internally *non-blocking* using libuv

Exposed as *synchronous* API to the programmer

Work in progress

Macros

```
macro_rules! def(
  ($var:ident => $val:expr) => (
    let $var = $val;
  )
)

fn main() {
  def!(a => 10u);
  io::println(a.to_str());
}
```

And much more!

www.rust-lang.org