Rust

sicher, performant und nebenläufig

Michael Neumann / mneumann@ntecs.de

Anforderungen

Ähnlich performant wie C++

Deterministische Performanz

Niedriger Speicherverbrauch

Asynchrones I/O

Warum Rust?

und nicht *D*, *Java* oder *Go*?

Können C++ **nicht** ersetzen!

C++ bisher einzigste Wahl bei *Performance-sensitiven, reaktiven* Anwendungen

Webbrowser

Firefox (C++)

Chrome (C++)

Internet Explorer (C++)

Opera (C++)

Safari (C++)

...

Spiele

Existiert ein grafisch-intensives kommerzielles Spiel dass *nicht* in C++ geschrieben ist?

System Software

JVM Runtime (C/C++)

LLVM/clang (C++)

gcc (C++)

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

Betriebssysteme (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++ ist unsicher

NULL pointer exceptions (segfaults)

Buffer-overruns (out of bounds)

dangling-pointers, double-frees, memory leaks

Implicit casts

Nicht initialisierte Variablen (besonders bei Klassen)

C++ ist fehleranfällig

`if (a = NULL)`

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

`switch` "fallthrough"

fehlendes `virtual`

Problem

D, go, ...

Nicht deterministisches Verhalten bei der Speicherallokierung

Praktisch *nicht* ohne Garbage Collector verwendbar

Fast alles muss GCed werden

Garbage Collector skaliert nicht

Lösung

Statische Lebenszeiten...

```cpp
// C++
{
  string s = "Hello World";
  vector<int> v{1,2,3};
  {
    cout << s << endl;
  }
} // Deallokation von s und v
```

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

Objekte werden auf dem Heap allokiert. Kein GC, da Lebensdauer zur Compilezeit bekannt.

... und Referenzen

```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);
}
```

Wie Pointer. Jedoch: Garantiert nicht Null. Nicht zuweisbar. Begrenzte Lebenszeit.

Unique Pointer

Pointer-Typ mit nur *einem* Alias.

```
{
  let i = ~999u; // Deklariert und allokiert unique integer Pointer
  let j = i;    // Verschiebt den Pointer in Variable "j"
  io::println((*j).to_str()); // 999
  io::println((*i).to_str()); // compile error: "use of moved value i"
} // Deallokation
```

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

Manuelle Deallokation

```
{
  let i = ~999u; // Deklariert und allokiert unique integer Pointer
  {
    let j = i;    // Verschiebt den Pointer in Variable "j"
  } // Deallokation
  io::println((*i).to_str()); // compile error: "use of moved value i"
}
```

Unique Pointer in *Rust*

aka. Owned-Pointer

Heap-allokiert

Compilezeit geprüft (kein Segfault wie in C++)

Deallokation sobald Gültigkeitsbereich verlassen wird

Neben Referenzen, der häufigste Zeigertyp

Kann an andere Tasks gesendet (verschoben) werden

Managed Pointer 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 oder Garbage Collected

Allokiert auf *Task-lokalem* Heap

Muss beim Senden an andere Tasks kopiert werden

Sehr selten in Rust

Raw Pointer in *Rust*

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

Wie Pointer in C/C++

Zur Interaktion mit C/C++

Referenzen in *Rust*

aka. Borrowed Pointer

Nur solange gültig wie referenziertes Objekt

Gültigkeitsbereich zur Compilezeit bekannt

Slicing `&str` und `&[]`

```
let s = ~"Hallo Welt";
let _welt: &str = str::slice(s, 5, s.len()); // " Welt"
let welt = str::trim_left(s);
io::println(welt); // "Welt"
```

Referenzen auf *Strings* und *Arrays* beinhalten neben dem Pointer auch noch eine Länge ("Fat-Pointer").

Freezing

```
let mut s = ~"Hallo";
s.push_str(" ");
{
  let all = s.slice(1, 4);
  io::println(all); // "all"
  s.push_str("Welt"); // COMPILE ERROR!
}
s.push_str("Welt");

io::println(s); // "Hallo Welt"
```

Referenziert man ein mutierbares Objekt so wird dieses für die Lebenszeit der Referenz *eingefroren*.

Wichtig da `push_str()` evtl. einen *neuen* String allokiert

Lebenszeiten von Referenzen

```
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) );
}
```

Lebenszeiten von Referenzen werden mit `'name` angegeben

Eingebaute Unit-Tests

```
// 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
```

Generische Datentypen

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

impl<T> Container<T> {
  fn size(&self) -> { 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}"
}
```

Interfaces bzw. Haskell's *Type Classes*

Generische 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() {
```

Serialisierung

```
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}];
```

std::serialize

Abstrahiert vom Datenformat: **JSON**, **MsgPack**, ...

Automatische Generierung

```
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}

//
// Anstelle von:
//
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 { ... }` startet Task

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

Tasks

Leichtgewichtig

Kooperativ

Dynamischer *split stack*

Getrennter Addressraum

per Task GC

Task als Exception Handler

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

`fail!(...)` zerstört den Task.

Kann mittels `task::try` aufgefangen werden

Beispiel

```
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`

Hierarchisch

Unidirektionale Pipes

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

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

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

Dank *Unique Pointer* sehr effizient!

Bidirektionale 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

Mehrere Sender, ein Empfänger

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

Protokol *Enforcement*

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

Module pipes

Tasks und I/O

Intern *nicht-blockierend* dank libuv

Für den Programmierer jedoch *synchron*

Work in Progress

Makros

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

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