```
$ cat .profile
GIT_AUTHOR_NAME=Florian Gilcher
GIT_AUTHOR_EMAIL=florian.gilcher@asquera.de
TM_COMPANY=Asquera GmbH
TWITTER_HANDLE=argorak
GITHUB_HANDLE=skade
```

- Backend developer
- Focused on infrastructure and databases

- Elasticsearch Usergroup
- mrgn.in meetup
- Rust Usergroup (co-org)
- organizer alumni eurucamp
- organizer alumni JRubyConf.EU
- Ruby Berlin board member

I generally have a nose for new things:

- learned Ruby before Rails was a thing
- Elasticsearch early adopter
- Now into Rust

# Rust is the Only Thing That's Left

Not necessarily new,
but definitely emerging

# Three goals

- Safe
- Concurrent
- (Predictably) Fast

# Safe

- static type system with local type inference
- null-pointer free (except...)
- unique pointers all the way

# Concurrent

- Mutability as a first-class concept
- No shared mutable state
- Important base types built in

# (Predictably) fast

- No hidden allocations
- Predictable deallocations
- Abstractions must be zero-cost or cheap (computation-wise)
- (except...) optional unsafe sublanguage

# We’re aiming at C here

- Controllable memory-layout
- Can generate C-ABI binaries
- No runtime
- Close to the machine

# What does it look like?

```rust
fn main() {
    println!("Hello strange-group!");
}
```

# Braces, like Richie indented

```rust
pub type Id = u64;

pub enum KeyType {
    Queue,
    Chunk
}

pub struct Key {
    id: Id,
    keytype: KeyType,
}
```

```rust
#[repr(u64)]
pub enum KeyType {
    Queue,
    Chunk
}
```

```
#[repr(u64)]
#[deriving(Show,
           PartialEq,
           Eq,
           PartialOrd,
           Ord,
           Clone)]
pub enum KeyType {
    Queue,
    Chunk
}
```

# Generics

```
enum Result<T, E> {
  Ok(T),
  Err(E)
}

enum Option<T> {
  Some(T),
  None
}
```

# Traits

```
impl Key {
  pub fn default() -> Key {
    Key { keytype: Queue, id: 0 }
  }

  pub fn new(keytype: KeyType, id: Id) -> Key
{
    Key { keytype: keytype, id: id }
  }

  pub fn compare(&self, other: &Key) ->
Ordering {
    //...
  }
}
```

```
pub trait Comparable {
  fn compare(&self, other: &Key) -> Ordering;
}

impl Comparable for Key {
  fn compare(&self, other: &Key) -> Ordering
{
    //...
  }
}
```

# (almost) everything is an expression

```
fn main() {
  let foo = if key.keytype == Queue {
                "this is a queue key"
            } else {
                "this is a chunk key"
            }

  println!(foo);
}
```

```
fn main() {
  let key = Key { keytype: Queue, id: 1 };
  let foo = match key.keytype {
                Queue => "queue key",
                Chunk => "chunk key"
              };

  println!("{}", foo);
}
```

# Semantic semicolons

```
fn main() -> () {
    "Hello!"
}
// error: mismatched types: expected `()`,
found `&'static str` (expected (), found
&-ptr)

fn main() -> () {
    "Hello!";
}
```

# Mutability

```
fn main() {
    let x = 5i;
    x = 4i;
}
//error: re-assignment of immutable variable
`x`
//x = 4i;
//...
```

```
fn main() {
    let mut x = 5i;
    x = 4i;
}
```

# References

```
fn add_one(x: &int) -> int { *x + 1 }

fn main() {
    assert_eq!(6, add_one(&5));
}
```

```
fn main() {
    let x = 5i;
    let y = &mut x;
}
//error: cannot borrow immutable local
variable `x` as mutable

fn main() {
    let mut x = 5i;
    let y = &mut x;
}
```

```
trait Access for Database {
    fn get(&self, key: &[u8]) {
        //...
    }

    fn put(&mut self,
           key: &[u8],
           value: &[u8]) {
        //...
    }
}
```

# Mutability runs deep

# Ownership, borrowing

- Every resource in Rust has a single owner
- controls when the resource is deal-located.

- may lend that resource, immutably, as much as wanted
- may lend that resource, mutably, to a single borrower.
- borrows cannot outlive the owner

Refcounting for people
that can only count to one.

Ownership can move,
but never be copied.

Values can be copied, cloned, etc. and will then be owned by someone else.

Decidable at compile time.

Predictible, safe memory management without garbage collection.

```rust
fn main() {
    let tempdir = try!(
      TempDir::new("strange-group")
    );
    let mut tempfile =
            try!(File::create(
              &tempdir.path().join("tmp")
            ));
    // look, no close necessary!
}
```

Deallocation and destructors run when leaving the scope.

```rust
fn main() {
    let tempdir = try!(
      TempDir::new("strange-group")
    );
    let mut tempfile =
            try!(File::create(
              &tempdir.path().join("tmp")
            ));
    drop(tempfile);
}
```

```rust
#[inline]
#[stable]
pub fn drop<T>(_x: T) { } // takes ownership,
does nothing
```

Mutability tracking, a set of carefully chosen base traits and borrow rules form the basis of making Rust a safe language for concurrent use.

No mutable borrows over boundaries,
special traits for things that
can be sent between tasks.

# FFI

```
#[repr(c)]
struct MyStruct {
    one: int,
    two: int
}
// free movement between C-Space and
Rust-Space
```

```
#[link(name = "tsm")]
extern {
  pub fn tsm_vte_new(
    out: *mut *mut tsm_vte,
    con: *mut tsm_screen,
    write_cb: tsm_write_cb,
    data: *mut c_void,
    log: Option<tsm_log_t>,
    log_data: *mut c_void) -> c_int;
}

extern "C" fn write_cb(_: *const tsm_vte,
                       _: *const u8,
                       _: size_t,
                       _: c_void) {}
```

# unsafe

```
impl Key {
    pub fn from_u8(key: &[u8]) -> &Key {
        use std::mem::transmute;

        assert!(key.len() == 16)

        unsafe { transmute(key.as_ptr()) }
    }
}
```

A sublanguage for pointer fiddling, unsafe memory operations for speed, interfacing with C, etc.

**highfive** commented on 30 Sep

⚠ **Warning** ⚠

- These commits modify **unsafe code**. Please review it carefully!

Aggressive bias towards using the language to gain insight

Rust wants you to declare a lot.

Rust has a complainer.

The amount of lints built into the compiler is mindboggling, strict and very useful.

```rust
use std::io::TempDir;

fn main() {
    TempDir::new("strange-group");
}
// warning: unused result which must be used,
#[warn(unused_must_use)] on by default
```

```rust
use std::io::TempDir;

#[deny(unused_must_use)]
fn main() {
    TempDir::new("strange-group");
}
// error: unused result which must be used
```

Results are everywhere.

```
#![deny(missing_docs)]
#![deny(warnings)]
// Karma boost!
```

This is a first-class language feature, not a compiler feature!

- Unused documentation
- Type casing
- Unused code
- Unused struct fields
- private struct leaks

# Good doc tools

- rustdoc allows testing of embedded code
- most example code in the rust world is runnable for that reason
- play.rust-lang.org allows playing with it

# Dependency management tools

- Cargo, like Bundler for Rust
- The third try
- Sane default code layout rules
- Provides test harness

# Bias towards good documentation

- Guide currently written by a paid author (Steve Klabnik)
- Good documentation of the stable stdlib
- Stability-markers throughout

# Production use:

- Used to build a browser engine passing ACID 2 (Servo)
- Used at skylight.io
- A couple of small tooling tasks

# Extreme focus on community

- Community code of conduct...
- ... that has seen necessity and use
- IRC and /r/rust are very well moderated
- Awesome community code reviews

- A community CI server that trigger travis builds nightly
- Weekly newsletters about changes
- open core meetings
- meetup groups around the world

# Emerging

Release of 1.0 will be in January or early 2015.

backwards-compatible releases afterwards, every 6 weeks

# beta and nightly channels

# Stability

Rust has no runtime.  Bugs with failing compilations are possible (albeit rare), but the resulting binaries are of good quality.

# Usage

Rust is a good language to bind to C libraries and provides tools to easily abstract over C.

It is also a good language
to replace C libraries.

Rust aims to fulfill all your needs when generating native binaries, down to specifics of microprocessors.

Game development
takes huge interest.

# Outstanding issues

Lack of important
libraries, most notably:

- HTTP
- Fast serialization/deserialization (being reworked)
- DB drivers, interfaces, etc.

# Can I use it now?

Yes.

Up until 1.0, use nightly builds.

It has been used for serious software for a while now.

# Type system

Rust uses a linar (affine, to be exact) type system without ever mentioning it or having you bother with it.
Also, no higher-kinded types for 1.0, because there needs to be a 1.0.

# A bit of history

Rust, 3 years ago

```
fn draw_a_square_and_a_circle(gfx:
graphics_context) {
    let s = @square(...);
    let c = @circle(...);
    let objs = [s as draw, c as draw];
    draw_all(gfx, objs);
}

fn draw_many<D:draw>(gfx: graphics_context,
drawables: [D]) {
    for drawables.each {|drawable|
        drawable.draw(gfx)
    }
}
```

- Classes, Interfaces
- Optional Garbage collection
- Task-Runtime using libuv under the hood (later: multiple)
- Star Wars, redefined: &, *, ~, @ as pointer sigils
- A lot more special cases (e.g. owned pointers vs. gc-ed pointers)

If there are two ways, let’s do them both

# Rust, today

- Traits, Structs and Enums only
- No garbage collection
- All traces of runtimes removed
- Star Wars, old-school:  &, * are the only sigils left
- a rather lean language

There are multiple ways, let’s provide the building blocks.

Multiple rewrites of
important parts of the stdlib.

Trimming and throwing away is a strong part of the culture until release.

Clear focus and target.

Rust is the only thing
that’s left after that.

# Thank you!

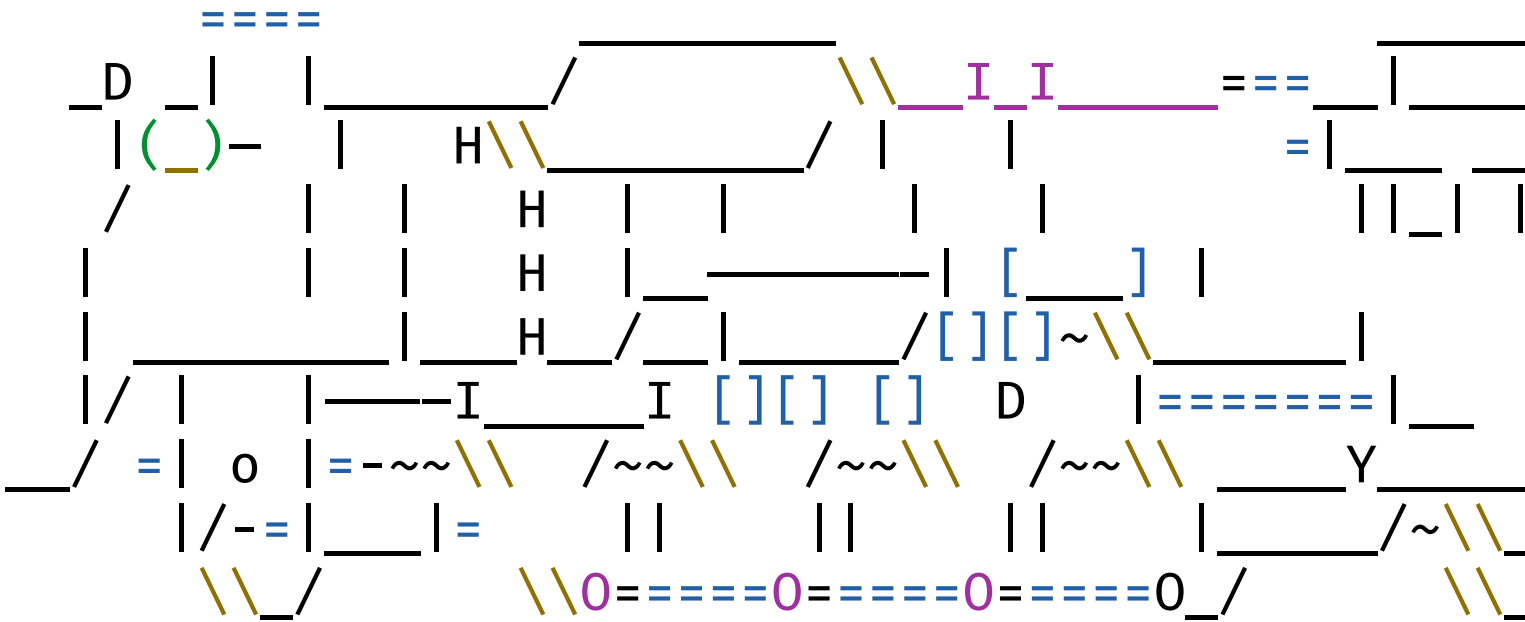
I’ll post references to
the meetup group later.

# Coda

Rust has also been used for silly software for a while now.

There’s an issue I’d
like to speak about.

# SL(1)



- sl(1) is in danger
- appreciation is sinking
- distributions ship it with options allowing to stop the train (debian)
- in outdated versions (debian)

I’m pretty sure someone is rebuilding it as a systemd module.

Rebuild your own sl(1), as close to the original as possible, in your favourite language, system, whatever.

https://github.com/mtoyoda/sl

Tweet to:  @argorak