Project Servo

Technology from the past
come to save the future
from itself

Mozilla Annual Summit, July 2010

<graydon@mozilla.com>

Hi

- I have been writing a compiled, concurrent, safe, systems programming language for the past four and a half years.
- Spare-time kinda thing. Yeah, I got problems.
- A small group of people in Mozilla got interested in it this past year, once I told them what I was up to.
- We've been trying to finish it for the past few months, to see what we can make of it.

OMGWTFBBQ?!

- Relax.
- There is no master plan, nefarious plot, etc.
- You are not going to be forced to use it.
- We are **not** “rewriting the browser”. That's impossible. Put down the gun.
- We do not know what exactly will come of it.
- It was a coincidence of a maturing side project and a desire for some slightly less-annoying language technology, nothing crazy.

Why Oh Why? (#1)

- C++ is well past expiration date:
  - Wildly unsafe in almost every way
    - Memory unsafe, no ownership policies, no concurrency control at all, can't even keep const values constant.
  - Heavily burdened with legacy issues
    - Absurd compilation model, weak linkage and module system, nigh-impossible to write tools for.
  - Spend more time fighting its weaknesses than seems reasonable.
    - Maybe you've noticed?

Why Oh Why (#2)

- Most “new” languages are unsuitable. One or more of:
  - JVM/CLR or similar tie-in, VM/FFI burden.
  - Complex GC + pointer-heavy = poor memory use.
  - “Different paradigm” (hard to find talent for, comprehension barrier, unpredictable).
  - “Script language” (few types or static checks).
  - Mostly ignore isolation, interference, concurrency.
- Everyone is *dodging* the niche I'm interested in.

Introducing: Rust

- **Rust** is a language that mostly cribs from past languages. **Nothing new**.
- Unapologetic interest in the static, structured, concurrent, large-systems language niche.
  - Not for scripting, prototyping or casual hacking.
  - Not for research or exploring a new type system.
- Concentrate on **known** ways of achieving
  - more safety,
  - more concurrency,
  - less mess.

Nothing new?

- Hardly anything. Maybe a keyword or two.
- Many older languages *better* than newer ones:
  - eg. Mesa (1977), BETA (1975), CLU (1974) ...
    - We keep forgetting already-learned lessons.
- Rust picks from 80s / early 90s languages:
  - **Nil** (1981), **Hermes** (1990),
  - **Erlang** (1987),
  - **Sather** (1990),
  - **Newsqueak** (1988), **Alef** (1995), **Limbo** (1996),
  - **Napier** (1985, 1988).

A quick taste (#1)

- It looks like a C-lineage family:

```
fn main() {
    log “hello, world”;
}
```

- It has most of the usual statements:

```
fn max(int x, int y) -> int {
    if (x > y) {
        ret x;
    } else {
        ret y;
    }
}
```

A quick taste (#2)

- Stack iterators:

```
iter range(int lo, int hi) -> int {
    while (lo < hi) {
        put lo;
        lo += 1;
    }
}

fn main() {
    for each (int i in range(1, 10)) {
        log i;
    }
}
```

A quick taste (#3)

- Lightweight tasks:

```
fn worker(int lo, int hi) {
    while (lo < hi) {
        log lo;
        lo += 1;
    }
}

fn main() {
    let task t0 = spawn worker(1, 100);
    let task t1 = spawn worker(100, 200);
    join t0;
    join t1;
}
```

A quick taste (#4)

- Structural objects and local type inference:

```
obj counter(int i) {
    fn incr() {
        i += 1;
    }
    fn get() -> int {
        ret i;
    }
}

fn main() {
    auto c = counter(10);
    c.incr();
    log c.get();
}
```

A quick taste (#5)

- Type-parametric code and structural types

```
obj swap[T](tup(T,T) pair) -> tup(T,T) {
    ret tup(pair._1, pair._0);
}

fn main() {
    auto str_pair = tup(“hi”, “there”);
    auto int_pair = tup(10, 12);
    str_pair = swap[str](str_pair);
    int_pair = swap[int](int_pair);
}
```

Ok, that could go on all day

- There is a lot I'm not showing there.
- The semantics is the interesting part.
- The syntax is, really, about the last concern.
- That was just a “taste” so you don't get all frustrated wondering what it looks like and/or assume that at the last minute it's going to read like Lisp or Haskell
  - (Hush, I know and love these languages, but there is a time and place).

Details! (#1)

- **Static** safety:
  - Memory safety, *no wild pointers.*
  - Typestate system*, no null pointers.*
  - Mutability control, *immutable by default.*
  - Side-effect control, *pure by default.*

Details! (#2)

- **Dynamic** safety:
  - Bounds-checked indexing, trapped signals, etc.
  - Dynamic *assertions* drive typestates.
  - All errors cause *failure*, unwinding.
    - “Expected errors”? Use a disjoint union return.
  - Failure of a task is *non-recoverable.*
    - “Crash-only” tasks with isolation, trapping.
    - Pervasive logging, annotations for unwinding.
    - Supervision / restart task ownership tree.

Details (#3)

- **Pragmatic** safety:
  - You can break the static rules.
  - You have to authorize *where* and *how*.
  - In a standard way, that's integrated into the language and easy to audit.
  - And globally visible, in a single place per-project.
    - Device for applying (or ignoring) social pressure.
    - Mechanism not policy.
    - Decide for yourself how strong your stomach is.

Details! (#4)

- **Structural** type bestiary:
  - Records, tuples, vectors.
  - Tagged disjoint unions.
  - First class functions (with bindings).
  - *Structural* objects.
    - Lightweight.
    - Immutable by default also.
    - No classes, no class hierarchy.
      - Just object types and objects that conform to them.

Details! (#5)

- **Actor** language bestiary:
  - Lightweight tasks (spawn 100k tasks = ~1s)
  - Async, half-duplex, weak, transmittable channels.
    - “buffered capabilities”.
  - **No shared mutable state.**
  - Can only pass immutable messages.
  - Idempotent task failure, failure-signal linkage.

Details! (#6)

- **Systems** language bestiary:
  - Fast calling of C (~8 insns, switch stacks).
  - Fast and safe stack-iterators (no cursor objects).
  - **No global GC** to fight (only per-task, mutable bits).
  - Real data structures (incl. nested structures).
    - Stack allocation, destructors, RAII.
  - Multi-file compilation / optimization.
    - ELF/MachO/PE + DWARF.
    - works with GDB, valgrind, shark, etc.

Details (#7)

- **Multi-paradigm** (hopefully clear by now).
  - Not “everything is an object”.
    - The object system is “pay as you go”, feature-wise.
  - Equal(-ish) support given to FP, procedural, actor and OO styles.
    - Different abstractions for different problems, trade-offs between control and expression, clarity and brevity.
    - Different strengths and weaknesses in each style.
    - Hopefully they combine tastefully.

Details! (#8)

- Other useful bits (trying to be thorough).
  - Type-parametric code.
  - Bignums.
  - Nested modules with import/export control.
  - UTF8 strings (not UCS2).
  - Marked syntax-extension system.
  - Reflection, dynamic type, type-switch.
- None of this stuff is particularly novel.

Implementation status

- Young, immature, hobby project until lately.
  - Mostly-done design by now, heads down.
  - ~90% language features “working” in rough form.
  - ~70% runtime working.
- 38kloc bootstrap compiler (Ocaml).
  - Built-in x86 backend for Linux, Win32, OSX.
  - LLVM backend in progress.
- Minimal standard library, mostly tests.

Inevitable question: is this like “Go”?

- No.
  - I've been working on Rust for years. Coincidence. There are dozens of actor languages in the pipeline. Go to a PL conference and ask around.
- Go seems to be barking up a different tree?
  - Has coroutines, but *kept shared mutable state.*
  - Has memory safety, but *kept null pointers.*
  - Has unwinding, but *no destructors or RAII.*
  - Has message passing, but *no immutability.*
  - Has some built-in generics, but *not in user code.*

Immediate plans

- Keep hacking on compiler, library, runtime.
  - Eventually transition to self-hosted frontend, LLVM backend.
  - Build out libraries and bindings.
- Need help:
  - Experienced language implementors!
  - Anyone who feels like bug fixing or library-writing.
  - Please: no research or novelty! There's plenty of known-good technology in the literature.
  - Also please: skip syntax or bikeshed arguments.

Released?

- Kinda. Not in any “supported” or stable sense.
- It's not ready for general use, but we felt bad enough keeping this quiet as long as we did.
  - Mostly my request, because I'm shy, and also because it was in flux for a while and needed focused attention and work, not debate.
- Hosting in public now.
- BSD-licensed, Github-hosted, we require committer agreement from you for us to pull.

Fini

github.com/graydon/rust

Demos and
Q and A time!