Rust ❤️ Emacs

rust emacs

130,435 commits | 6 branches | 0 releases | 540 contributors | GPL-3.0

Branch: master | New pull request | Find file | Clone or download

birkenfeld committed on GitHub Merge pull request #234 from Wilfred/floatfns ... Latest commit f6c81e1 41 seconds ago

admin	Remove more traces of unused gnulib modules.	an hour ago
build-aux	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
doc	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
etc	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
leim	More remacs renaming and packaging path fixups	6 months ago
lib-src	Handle remaining warnings.	3 days ago
lib	Remove more traces of unused gnulib modules.	an hour ago
lisp	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
lwlib	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
m4	Remove more traces of unused gnulib modules.	an hour ago
modules	Integrate module test with normal test suite	3 months ago
nextstep	Merge emacs/master into master	6 months ago
nt	Remove more traces of unused gnulib modules.	an hour ago
oldXMenu	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
rust_src	Merge pull request #234 from Wilfred/floatfns	41 seconds ago
src	Merge pull request #234 from Wilfred/floatfns	41 seconds ago
test	Fixup tramp test to work with a strict POSIX shell.	3 days ago
.dir-locals.el	* .dir-locals.el (c-noise-macro-names): Remove NONVOLATILE.	11 months ago
.dockerignore	Add docker environment	a month ago
.gitattributes	Update .gitattributes to match sources better	2 months ago
.gitignore	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
.gitlab-ci.yml	; Spelling fix	2 months ago
.travis.sh	updated travis to format remacs-macros and updated README	27 days ago
.travis.yml	Explicitly pin a rustfmt version	29 days ago
CONTRIBUTING.md	Mention @birkenfeld	8 days ago
COPYING	Restore files that I seem to have mistakenly deleted.	7 years ago
Dockerfile	Add docker environment	a month ago
GNUmakefile	Merge from origin/emacs-25	7 months ago
INSTALL	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
INSTALL.REPO	; INSTALL.REPO: Mention cases that 'make bootstrap' can't handle (Bug…	2 months ago
Makefile.in	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
README	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
README.md	Highlight bash comments [ci skip]	11 hours ago
REMACS_COMPATIBILITY.md	Mention MS-DOS in compatibility docs [ci skip]	5 months ago
autogen.sh	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
configure.ac	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago
docker-compose.yml	Add docker environment	a month ago
make-dist	Merge https://git.savannah.gnu.org/git/emacs into rollup	3 days ago

README.md

Rust ❤️ Emacs

chat on gitter | build passing

A community-driven port of Emacs to Rust.

GPLv3 license.

Table of Contents

Why Emacs?

Emacs will change how you think about programming.

Emacs is **totally introspectable**. You can always find out 'what code runs when I press this button?'.

Emacs is an **incremental programming environment**. There's no edit-compile-run cycle. There isn't even an edit-run cycle. You can execute snippets of code and gradually turn them into a finished project. There's no distinction between your editor and your interpreter.

Emacs is a **mutable environment**. You can set variables, tweak functions with advice, or redefine entire functions. Nothing is off-limits.

Emacs **provides functionality without applications**. Rather than separate applications, functionality is all integrated into your Emacs instance. Amazingly, this works. Ever wanted to use the same snippet tool for writing C++ classes as well as emails?

Emacs is full of **incredible software concepts that haven't hit the mainstream yet**. For example:

- Many platforms have a single item clipboard. Emacs has an **infinite clipboard**.
- If you undo a change, and then continue editing, you can't redo the original change. Emacs allows **undoing to any historical state**, even allowing tree-based exploration of history.
- Emacs supports a **reverse variable search**: you can find variables with a given value.
- You can perform **structural editing** of code, allowing you to make changes without breaking syntax. This works for lisps (paredit) and non-lisps (smartparens).
- Many applications use a modal GUI: for example, you can't do other edits during a find-and-replace operation. Emacs provides **recursive editing** that allow you to suspend what you're currently doing, perform other edits, then continue the original task.

Emacs has a **documentation culture**. Emacs includes a usage manual, a lisp programming manual, pervasive docstrings and even an interactive tutorial.

Emacs has **a broad ecosystem**. If you want to edit code in a niche language, there's probably an Emacs package for it.

Emacs doesn't have a monopoly on good ideas, and there are other great tools out there. Nonetheless, we believe the Emacs learning curve pays off.

Why Rust?

Rust is a great alternative to C.

Rust has **a fantastic learning curve**. The documentation is superb, and the community is very helpful if you get stuck.

Rust has **excellent tooling**. The compiler makes great suggestions, the unit test framework is good, and `rustfmt` helps ensure formatting is beautiful and consistent.

The Rust **packaging story is excellent**. It's easy to reuse the great libraries available, and just as easy to factor out code for the benefit of others. We can replace entire C files in Emacs with well-maintained Rust libraries.

Code written in Rust **easily interoperates with C**. This means we can **port to Rust incrementally**, and having a working Emacs at each step of the process.

Rust provides **many compile-time checks**, making it much easier to write fast, correct code (even when using multithreading). This also makes it much easier for newcomers to contribute.

Give it a try. We think you'll like it.

Why A Fork?

Emacs is a widely used tool with a long history, broad platform support and strong backward compatibility requirements. The core team is understandably cautious in making far-reaching changes.

Forking is a longstanding tradition in the Emacs community for trying different approaches. Notable Emacs forks include XEmacs, Guile Emacs, and emacs-jit.

There have also been separate elisp implementations, such as Deuce, JEmacs and El Compilador.

By forking, we can **explore new development approaches**. We can use a pull request workflow with integrated CI.

We can **drop legacy platforms and compilers**. Remacs will never run on MS-DOS, and that's OK.

There's a difference between **the idea of Emacs** and the **current implementation of Emacs**. Forking allows us to explore being even more Emacs-y.

Getting Started

Requirements

1. You will need Rust installed. If you're on macOS, you will need Rust nightly.
2. You will need a C compiler and toolchain. On Linux, you can do something like `apt-get install build-essential automake`. On macOS, you'll need Xcode.
3. You will need some C libraries. On Linux, you can install everything you need with:

```
apt-get install texinfo libjpeg-dev libtiff-dev \
  libgif-dev libxpm-dev libgtk-3-dev libgnutls-dev \
  libncurses5-dev libxml2-dev
```

On macOS, you'll need libxml2 (via `xcode-select --install`) and gnutls (via `brew install gnutls`).

Dockerized development environment

If you don't want to bother with the above setup you can use the provided docker environment. Make sure you have docker 1.12+ and docker-compose 1.8+ available.

To spin up the environment run

```
docker-compose up -d
```

First time you run this command docker will build the image. After that any subsequent startups will happen in less than a second.

The working directory with remacs will be mounted under the same path in the container so editing the files on your host machine will automatically be reflected inside the container. To build remacs use the steps from Building Remacs prefixed with `docker-compose exec remacs`, this will ensure the commands are executed inside the container.

Building Remacs

```
$ ./autogen.sh
$ ./configure --enable-rust-debug
$ make
```

For a release build, don't pass `--enable-rust-debug`.

The Makefile obeys cargo's RUSTFLAGS variable and additional options can be passed to cargo with CARGO_FLAGS.

For example:

```
$ make CARGO_FLAGS="-vv" RUSTFLAGS="-Zunstable-options --pretty"
```

Running Remacs

You can now run your shiny new Remacs build!

```
# Using -q to ignore your .emacs.d, so Remacs starts up quickly.
# RUST_BACKTRACE is optional, but useful if your instance crashes.
$ RUST_BACKTRACE=1 src/remacs -q
```

Rustdoc builds

You can use rustdoc to generate API docs:

```
# http://stackoverflow.com/a/39374515/509706
$ cargo rustdoc -- \
    --no-defaults \
    --passes strip-hidden \
    --passes collapse-docs \
    --passes unindent-comments \
    --passes strip-priv-imports
```

You can then open these docs with:

```
$ cargo doc --open
```

Porting Elisp Primitive Functions: Walkthrough

Let's look at porting `numberp` to Rust.

First, make sure you have configured and built Remacs on your system. You'll probably want to generate TAGS too, so you can jump to definitions of C functions.

Emacs C uses a lot of macros, so it's also useful to look at the expanded version of the code.

Define a little file `src/dummy.c` with the C source of `numberp`, along with the `lisp.h` header file:

```
#include "lisp.h"

DEFUN ("numberp", Fnumberp, Snumberp, 1, 1, 0,
       doc: /* Return t if OBJECT is a number (floating point or integer).  */
       attributes: const)
  (Lisp_Object object)
{
  if (NUMBERP (object))
    return Qt;
  else
    return Qnil;
}
```

Then expand it with GCC:

```
$ cd /path/to/remacs
$ gcc -Ilib -E src/dummy.c > dummy_exp.c
```

This gives us a file that ends with:

```
static struct Lisp_Subr
# 3 "src/dummy.c" 3 4
_Alignas
# 3 "src/dummy.c"
(8) Snumberp = { { PVEC_SUBR << PSEUDOVECTOR_AREA_BITS }, { .a1 = Fnumberp }, 1, 1, "numberp", 0, 0}; Li


  (Lisp_Object object)
{
  if (NUMBERP (object))
    return Qt;
  else
    return builtin_lisp_symbol (0);
}
```

We can see we need to define a `Snumberp` and a `Fnumberp`. We define a `numberp` function that does the actual work, then use an attribute (implemented as a procedural macro) named `lisp_fn` that handles these definitions for us:

```
// This is the function that gets called when
// we call numberp in elisp.
//
// `lisp_fn` defines a wrapper function that calls numberp with
// LispObject values. It also declares a struct that we can pass to
// defsubr so the elisp interpreter knows about this function.

/// Return t if OBJECT is a number.
#[lisp_fn]
fn numberp(object: LispObject) -> LispObject {
    LispObject::from_bool(object.is_number())
}
```

The elisp name of the function is derived from the Rust name, with underscores replaced by hyphens. If that is not possible (like for the function `+`), you can give an elisp name as an argument to `lisp\_fn`, like `#[lisp_fn(name = "+")]`.

Optional arguments are also possible: to make the minimum number of arguments from elisp different from the number of Rust arguments, pass a `min = "n"` argument.

The docstring of the function should be the same as the docstring in the C code. (Don't wonder about it being a comment there, Emacs has some magic that extracts it into a separate file.)

Finally, we need to delete the old C definition and call `defsubr` inside `rust_init_syms`:

```
pub extern "C" fn rust_init_syms() {
    unsafe {
        // ...
        defsubr(&*yourmodule::Snumberp);
    }
}
```

You're done! Compile Remacs, try your function with `M-x ielm`, and open a pull request. Fame and glory await!

Porting Widely Used C Functions

If your Rust function replaces a C function that is used elsewhere in the C codebase, you will need to export it. The wrapper function needs to be exported in lib.rs:

```
pub use yourmodulename::Fnumberp;
```

and add a declaration in the C where the function used to be:

```
// This should take the same number of arguments as the Rust function.
Lisp_Object Fnumberp(Lisp_Object);
```

Design Goals

Compatibility: Remacs should not break existing elisp code, and ideally provide the same FFI too.

Similar naming conventions: Code in Remacs should use the same naming conventions for elisp namespaces, to make translation straightforward.

This means that an elisp function `do-stuff` will have a corresponding Rust function `Fdo_stuff`, and a declaration struct `Sdo_stuff`. A lisp variable `do-stuff` will have a Rust variable `Vdo_stuff` and a symbol `'do-stuff` will have a Rust variable `Qdo_stuff`.

Otherwise, we follow Rust naming conventions, with docstrings noting equivalent functions or macros in C. When incrementally porting, we may define Rust functions with the same name as their C predecessors.

Leverage Rust itself: Remacs should make best use of Rust to ensure code is robust and performant.

Leverage the Rust ecosystem: Remacs should use existing Rust crates wherever possible, and create new, separate crates where our code could benefit others.

Great docs: Emacs has excellent documentation, Remacs should be no different.

Non-Design Goals

`etags`: The universal ctags project supports a wider range of languages and we recommend it instead.

Contributing

Pull requests welcome, no copyright assignment required. This project is under the Rust code of conduct.

Help Needed

There's lots to do! We keep a list of low hanging fruit here so you can easily choose one. If you do, please open a new issue to keep track of the task and link to it.

Easy tasks:

- [ ] Find a small function in lisp.h and write an equivalent in lisp.rs.
- [ ] Improve our unit tests. Currently we're passing `Qnil` to test functions, which isn't very useful.
- [ ] Add docstrings to public functions in lisp.rs.
- [ ] Tidy up messy Rust that's been translated directly from C. Run `rustfmt`, add or rename internal variables, run `clippy`, and so on.
- [ ] Add Rust-level unit tests to elisp functions defined in lib.rs.

Medium tasks:

- [ ] Choose an elisp function you like, and port it to rust. Look at `rust-mod` for an example.
- [ ] Teach `describe-function` to find functions defined in Rust.
- [ ] Expand our Travis configuration to run 'make check', so we know remacs passes Emacs' internal test suite.
- [x] Expand our Travis configuration to ensure that Rust code has been formatted with rustfmt
- [ ] Set up bors/homu.
- [ ] Set up a badge tracking pub struct/function coverage using cargo-doc-coverage.
- [ ] Search the Rust source code for `TODO` comments and fix them.
- [ ] Teach Emacs how to jump to definition for Rust functions.

Big tasks:

- [ ] Find equivalent Rust libraries for parts of Emacs, and replace all the relevant C code. Rust has great libraries for regular expressions, GUI, terminal UI, managing processes, amongst others.
- [ ] Change the elisp float representation to use nan boxing rather than allocating floats on the heap.

Rust Porting Tips

C Functions

When writing a Rust version of a C function, give it the same name and same arguments. If this isn't appropriate, docstrings should say the equivalent C function to help future porters.

For example, `make_natnum` mentions that it can be used in place of `XSETFASTINT`.

C Macros

For C macros, we try to define a fairly equivalent Rust function. The docstring should mention the original macro name.

Since the Rust function is not a drop-in replacement, we prefer Rust naming conventions for the new function.

For the checked arithmetic macros (`INT_ADD_WRAPV`, `INT_MULTIPLY_WRAPV` and so on), you can simply use `.checked_add`, `.checked_mul` from the Rust stdlib.

Assertions

`eassert` in Emacs C should be `debug_assert!` in Rust.

`emacs_abort()` in Emacs C should be `panic!("reason for panicking")` in Rust.

Safety

`LispObject` values may represent pointers, so the usual safety concerns of raw pointers apply.

If you can break memory safety by passing a valid value to a function, then it should be marked as `unsafe`. For example:

```
// This function is unsafe because it's dereferencing the car
// of a cons cell. If `object` is not a cons cell, we'll dereference
// an invalid pointer.
unsafe fn XCAR(object: LispObject) -> LispObject {
    (*XCONS(object)).car
}

// This function is safe because it preserves the contract
// of XCAR: it only passes valid cons cells. We just use
// unsafe blocks instead.
fn car(object: LispObject) -> LispObject {
    if CONSP(object) {
        unsafe {
            XCAR(object)
        }
    } else if NILP(object) {
        Qnil
    } else {
        unsafe {
            wrong_type_argument(Qlistp, object)
        }
    }
}
```