Extending Emacs in Rust with Dynamic Modules

Tuấn-Anh Nguyễn

November 23, 2021

Emacs Dynamic Modules

- Loadable native libraries
- C ABI (`.so`/`.dylib`/`.dll`)
- C APIs
- Since Emacs 25

Rust vs. C

- Memory-safe and type-safe
- High-level, good tooling
- Strong community

Emacs Dynamic Modules in Rust

- https://github.com/ubolonton/emacs-module-rs
- High-level Rust APIs: `emacs` crate (i.e. library)
- Compiled to C ABI (`.so`/`.dylib`/`.dll`)

Demo

Declaring a Module - C

```c
#include <emacs-module.h>

int plugin_is_GPL_compatible;

static void provide (emacs_env *env, const char *feature)
{
  emacs_value Qfeat = env->intern (env, feature);
  emacs_value Qprovide = env->intern (env, "provide");
  emacs_value args[] = { Qfeat };
  env->funcall (env, Qprovide, 1, args);
}

int emacs_module_init (struct emacs_runtime *ert) {
  emacs_env *env = ert->get_environment (ert);
  // Other boilerplates to bind module functions here ...
  provide (env, "mod-test");
  return 0;
}
```

Declaring a Module - Rust

Decorate an init function with `#[emacs::module]`

```rust
use emacs::{Env, Result};

emacs::plugin_is_GPL_compatible!();

// Automatically calls `provide`, using the function name as the module name.
#[emacs::module(name(fn))]
fn mod_test(env: &Env) -> Result<()> {
    // No code to bind functions. They are automatically exposed.
    // Only very complex modules will need extra initialization here.
    Ok(())
}
```

Exposing Module Functions - C

```c
static intmax_t sum (intmax_t a, intmax_t b)
{
  return a + b;
}

static emacs_value
Fmod_test_sum (emacs_env *env, ptrdiff_t nargs, emacs_value args[], void *data)
{
  intmax_t a = env->extract_integer (env, args[0]);
  intmax_t b = env->extract_integer (env, args[1]);
  return env->make_integer (env, sum(a, b));
}

static void bind_function (emacs_env *env, const char *name, emacs_value Sfun)
{
  emacs_value Qdefalias = env->intern (env, "defalias");
  emacs_value Qsym = env->intern (env, name);
  emacs_value args[] = { Qsym, Sfun };
  env->funcall (env, Qdefalias, 2, args);
}

int emacs_module_init (struct emacs_runtime *ert) {
  emacs_env *env = ert->get_environment (ert);
#define DEFUN(lsym, csym, amin, amax, doc, data) \
  bind_function (env, lsym, \
         env->make_function (env, amin, amax, csym, doc, data))
  DEFUN ("mod-test-sum", Fmod_test_sum, 2, 2, "Return A + B\n\n(fn a b)", NULL);
  return 0;
}
```

Exposing Module Functions - Rust

Simply decorate functions with `#[defun]`

```rust
use emacs::{defun, Result};

// The correct signature (sum A B) will automatically show up in help modes.

/// Parse string B and add it to the integer A.
/// This same docstring will be seen in Emacs's help modes.
#[defun]
fn sum(a: i64, b: String) -> Result<i64> {
    let b: i64 = b.parse().unwrap();
    Ok(a + b)
}
```

Calling Lisp Functions - C

```c
// Can't fit on the slide!
static emacs_value
Fmod_test_non_local_exit_funcall (emacs_env *env, ptrdiff_t nargs,
                  emacs_value args[], void *data)
{
  assert (nargs == 1);
  emacs_value result = env->funcall (env, args[0], 0, NULL);
  emacs_value non_local_exit_symbol, non_local_exit_data;
  enum emacs_funcall_exit code
    = env->non_local_exit_get (env, &non_local_exit_symbol,
                  &non_local_exit_data);
  switch (code)
    {
    case emacs_funcall_exit_return:
      return result;
    case emacs_funcall_exit_signal:
      {
        env->non_local_exit_clear (env);
        emacs_value Flist = env->intern (env, "list");
        emacs_value list_args[] = {env->intern (env, "signal"),
                  non_local_exit_symbol, non_local_exit_data};
        return env->funcall (env, Flist, 3, list_args);
      }
    case emacs_funcall_exit_throw:
      {
        env->non_local_exit_clear (env);
        emacs_value Flist = env->intern (env, "list");
        emacs_value list_args[] = {env->intern (env, "throw"),
                  non_local_exit_symbol, non_local_exit_data};
        return env->funcall (env, Flist, 3, list_args);
      }
    }

  /* Never reached.  */
```

Calling Lisp Functions - Rust

```rust
use emacs::{defun, Result, Value, ErrorKind::{self, *}};

emacs::use_symbols! {
    signal throw
}

#[defun]
fn non_local_exit_funcall(func: Value) -> Result<Value> {
    let env = func.env;
    match func.call([]) {
        Err(error) => unsafe {
            match error.downcast_ref::<ErrorKind>() {
                Some(Signal { symbol, data }) =>
                    env.list((signal, symbol.value(env), data.value(env))),
                Some(Throw { tag, value }) =>
                    env.list((throw, tag.value(env), value.value(env))),
                _ => unreachable!{}
            }
        }
        v => v,
    }
}
```

Calling Lisp Functions - Rust

```rust
use emacs::{defun, Result, Env, Value};

// "Import" the functions.
emacs::use_symbols! {
    text_mode_hook
    variable_pitch_mode
    message
}

// Env exposes convenient functions to interact with the Lisp runtime.
// Lisp won't see it as a parameter in the function's signature.
#[defun]
fn my_settings(env: &Env) -> Result<Value> {
    // Calling through env.
    let args = env.list((1, "str"))?;
    env.call(message, ("Hello %d %s", args))?;
    // Calling the symbol directly.
    let add_hook = env.intern("add-hook")?;
    add_hook.call((text_mode_hook, variable_pitch_mode))
}
```

Type Conversions

- Automatic for inputs/output of `#[defun]`
- Can be done manually

```
let value = "5".into_lisp(env)?;
let i: i64 = value.into_rust()?;

true.into_lisp(env)?; // t
false.into_lisp(env)?; // nil
```

Error Handling

- Lisp: signals, throws
- Rust: `Result<T>`
- Automatic conversion at boundaries

Signaling Errors to Lisp

```
emacs::define_errors! {
    // (define-error 'not-an-integer "String cannot be parsed as integer")
    not_an_integer "String cannot be parsed as integer"
}

#[defun]
fn sum(x: i64, y: String, env: &Env) -> Result<i64> {
    match y.parse::<i64>() {
        Ok(y) => Ok(x + y),
        // (signal 'not-an-integer (list y))
        Err(_) => env.signal(not_an_integer, (y,))
    }
}

#[defun]
fn sum(x: i64, y: String, env: &Env) -> Result<i64> {
    // (signal 'not-an-integer (list string-repr-of-err))
    let y: i64 = y.parse().or_signal(env, not_an_integer)?;
    Ok(x + y)
}
```

Handling Errors from Lisp

```
emacs::use_symbols! {
    insert
    buffer_read_only
}

match env.call(insert, ("some_text",)) {
    Err(error) => {
        // Handle `buffer-read-only` error.
        if let Some(Signal { symbol, .. }) = error.downcast_ref::<ErrorKind>() {
            if symbol.value(env).eq(buffer_read_only) {
                env.message("This buffer is not writable!")?;
                Ok(())
            } else {
                // Propagate other errors.
                Err(error)
            }
        }
        // Propagate (throw TAG VALUE).
        Err(error)
    },
    _ => Ok(()),
}
```

Embedding Rust Data in Lisp

- Opaque to Lisp: `#<user-ptr ptr= ... finalizer= ... >`
- Manipulated by module functions
- Return value: add `user_ptr` option

```
#[defun(user_ptr)]
fn make_parser() -> Result<Parser> {
    Ok(Parser::new())
}
```

- Input parameters: pass references

```
#[defun]
fn timeout_micros(parser: &Parser) -> Result<u64> {
    Ok(parser.timeout_micros())
}

#[defun]
fn set_timeout_micros(parser: &mut Parser, max_duration: u64) -> Result<()> {
    Ok(parser.set_timeout_micros(max_duration))
}
```