Introduction to writing GNU Emacs native modules

Extending Emacs in C or other languages

Aurélien Aptel <aaptel@suse.com>
SUSE Labs / Samba team

GNU Emacs

- Editor of the Emacs family
- Written in 1976 by RMS
- 42 years old and still ~~going strong~~

  mildly popular
- Extensible and self-documented

✝ St. IGNUcius ✝ blessing a computer, date unknown

Hacker culture

- Emacs history is in intertwined with the legendary MIT AI lab, Lisp and hacker culture (HAKMEM, Jargon file, ...)
- *The Founding of the MIT AI lab* (2001)
- *Multics Emacs: The History, Design and Implementation* (1979; 1996)
- *EMACS: The Extensible, Customizable Display Editor* (rms, 1981) paper published on ACM!
- *Emacs Timeline* (jwz, 2007)
- *Emacs implementations and literature* (1990-2017)
- *Evolution of Emacs Lisp* (Stefan Monnier, Michael Sperber, 2018)
- ...Wikipedia to continue down the rabbit hole

Extensible?

- Emacs Lisp
- Fun language albeit not very well designed
- Bytecode compiler & VM
- Still not super fast
- Experimental JIT branch
- Tom Tromey working on Elisp compiler (in Elisp, obviously :)
  - https://github.com/tromey/el-compilador
  - https://github.com/tromey/emacs-ffi

IPC

- Until recently:
  - Files (duh)
  - Processes (sync and async)
  - TCP and UDP sockets (make connections/listen, send/receive)
  - D-BUS
- Not always convenient
  - Always needs some sort of serialization/unserialization to do anything remotely complex
- Most common solutions:
  - Pure Elisp
  - Write separate program and call it
  - Server program for persistence between calls

Native API

- Idea first introduced by Steve Kemp in 2000
  - Allowed C DEFUNs to be written in external libraries
  - Never merged
  - Some version of it by Reini Urban ended up in XEmacs
- Followup patch by Dave Love in 2006
  - Same idea, using libtool this time
  - Never merged
- Why?
  - Allowing Emacs to load native code could lead to proprietary software bundle (free Emacs + closed binary blob)

Artistic rendition of RMS shutting down potential threats to software freedom

Maintaining freedom

- Same licensing issues in GCC wrt plugins
- Solved sometime around 2009 by a adding a requirement in modules
  - Presence of “plugin_is_GPL_compatible” symbol
- How is this enough?

Maintaining freedom

How is this enough?

- I Am Not A Lawyer
- From GNU Coding Standards:
  - *By adding this check to your program you are **not** creating a new legal requirement. The GPL itself requires plug-ins to be free software, licensed compatibly. [...] the GPL and AGPL already require those plug-ins to be released under a compatible license. The symbol definition in the plug-in—or whatever equivalent works best in your program—makes it harder for anyone who might distribute proprietary plug-ins to legally defend themselves. If a case about this got to court, we can point to that symbol as evidence that the plug-in developer understood that the license had this requirement.*

Third time’s the charm?

- In 2014 I start my own attempt for Emacs, using the same requirement
  - libtool-based
  - Links against the Emacs binary itself
  - allows C DEFUNs to be defined in loadable shared libs
  - PoC received positively

Finally

- After some discussions on emacs-devel
  - Turns out people hate when their editor crashes!
  - People would rather have a more stable and robust API
    - i.e no internals knowledge needed
    - No “raw” calls to the Emacs C core
    - some guarantees that a module wont break after new emacs release
  - New API design similar to JNI proposed by Daniel Colascione
  - Implemented and merged with the help of Philipp Stephani
  - Released in Emacs 25!
  - ~2 years of on-and-off work, reviews and bike-shedding

The API

- Single header file emacs-module.h
- No linking against Emacs things needed
- 3 important types:
  - emacs_runtime: struct containing function ptrs for module initialization
  - emacs_env: struct containing function ptrs for vm interaction
  - emacs_value: opaque type to store Lisp objects
- Module must implement a emacs_module_init() function
- Emacs will call that function when the module is loaded.
- A runtime ready for use will be passed as argument

```
extern int emacs_module_init (struct emacs_runtime *ert);
```

emacs_runtime

- Struct providing operations for module initialization
- For now, just has an operation to get a new emacs_env

```
/* Struct passed to a module init function (emacs_module_init).  */
struct emacs_runtime
{
  /* Structure size (for version checking).  */
  ptrdiff_t size;


  /* Private data; users should not touch this.  */
  struct emacs_runtime_private *private_members;


  /* Return an environment pointer.  */
  emacs_env *(*get_environment) (struct emacs_runtime *ert);
};
```

emacs_env

- Struct providing operations for vm interactions
  - Type conversion Lisp <=> C
  - Lisp function calls
  - Basic tests (eq, nil test, ..)
  - Memory management
  - Exception handling
  - Function registration
  - Vector manipulation
  - ...

```
typedef struct emacs_env_25 emacs_env;

struct emacs_env_25
{
  /* Structure size (for version checking).  */
  ptrdiff_t size;


  /* Private data; users should not touch this.  */
  struct emacs_env_private *private_members;


  /* Memory management.  */


  emacs_value (*make_global_ref) (emacs_env *env,
                                  emacs_value ref);


  void (*free_global_ref) (emacs_env *env,
                           emacs_value ref);

     ...
```

Configure

- Modules are available starting with Emacs 25

- Disabled by default, need to pass --with-modules

Example: Adding a new function

All functions exposed to Emacs must
have this prototype

```
static emacs_value
Fmymod_test (emacs_env *env, ptrdiff_t nargs, emacs_value args[], void *data)
{
  return env->make_integer (env, 42);
}
```

Make the function callable from Lisp

```
/* create a lambda (returns an emacs_value) */
emacs_value fun = env->make_function (env,
      0,            /* min. number of arguments */
      0,            /* max. number of arguments */
      Fmymod_test,  /* actual function pointer */
      "doc",        /* docstring */
      NULL          /* user pointer of your choice
                       (data param in Fmymod_test) */
      );
```

We now have a Lisp function value
but we need to bind it to a symbol to be able to call it by name

Binding a symbol

- Elisp is a “Lisp-2”, every symbol has 2 cells:
  - Value cell
  - Function cell
- In other words, there are 2 different namespaces for variable names and function names
- fset sets the function cell of a symbol
- “`(defun foo () (code))`” is equivalent to “`(fset ‘foo (lambda () (code))`”
- Let’s make that call from C

Calling fset

```
void bind_fun (emacs_env *env, const char *fname, emacs_value fun) {
    env->funcall(env,
         env->intern(env, "fset"), /* symbol to call */
         2,                        /* 2 args */
         (emacs_value []) {
             env->intern(env, fname), /* arg 1: symbol to bind */
             fun                      /* arg 2: value */
         });
}
```

Putting it all together in our init function

```
int plugin_is_GPL_compatible;

int emacs_module_init (struct emacs_runtime *ert) {
  emacs_env *env = ert->get_environment (ert);

  /* create a lambda (returns an emacs_value) */
  emacs_value fun = env->make_function (env,
        0,            /* min. number of arguments */
        0,            /* max. number of arguments */
        Fmymod_test,  /* actual function pointer */
        "doc",        /* docstring */
        NULL          /* user pointer of your choice */
        );

  bind_fun (env, "mymod-test", fun);
  /* loaded successfully */
  return 0;
}
```

Compiling

```
# path to the emacs source dir
# (you can provide it here or on the command line)
#ROOT    =
CC      = gcc
LD      = gcc
CFLAGS  = -ggdb3 -Wall
LDFLAGS =
all: mymod.so

# make shared library out of the object file
%.so: %.o
        $(LD) -shared $(LDFLAGS) -o $@ $<

# compile source file to object file
%.o: %.c
        $(CC) $(CFLAGS) -I$(ROOT)/src -fPIC -c $<
```

Memory management

- Any emacs_value created inside a module function will be freed after it returns

```
emacs_value foo (emacs_env *env, ptrdiff_t nargs, emacs_value args[], void *data)
{
    emacs_value a = env->make_integer(env, 42);
    emacs_value b = env->make_integer(env, 43);


    return env->make_integer(env, 44);
    /* a and b automatically freed */
}
```

Memory management

- emacs_value can be marked as global (reference-counted)
  - Use free_global_ref() when you're done

```
emacs_value Qnil, Qt;

int emacs_module_init (emacs_runtime *rt)
{
     emacs_env *env = rt->get_environment(rt);
     Qt = env->make_global_ref(env, env->intern(env, "t"));
     Qnil = env->make_global_ref(env, env->intern(env, "nil"));
     /* ...register foo()... */
     return 0;
}

emacs_value foo (emacs_env *env, ptrdiff_t nargs, emacs_value args[], void *data)
{
     /* OK to use Qnil and Qt here */
}
```

Memory management

- Unless you use global references, no need to worry about memory
- global reference are not freed after a module function returns and are still valid
- But they cannot be used *between* module function calls, always have to be inside a module function
- Globals are reference counted and must be freed manually with free_global_ref()
- Common usecase:
  - Make values outlive function call
  - “cache” commonly used value (like certain symbols) to reduce CPU usage

Error handling

- Elisp has exceptions (“signals”) but C doesn’t
  - C has setjmp()/longjmp() but no built-in mechanism to properly unwind the stack
- Use non_local_exit_signal() to signal from a module function
- All API operations may fail and set a **pending per-env exit state**
  - Similar to errno in concept
- Use non_local_exit_check() to know how the last call ended

```
enum emacs_funcall_exit
{
  emacs_funcall_exit_return = 0,
  emacs_funcall_exit_signal = 1,
  emacs_funcall_exit_throw = 2
};
```

Error handling

- Use non_local_exit_clear() to clear the error state of an env
- Errors need to be checked as most API functions will fail and return immediately if there’s a pending error
- 2 idiomatic patterns for error checking:
  - Check every API call
  - Only check after “important” calls
    - If no decisions depends on the outcome of a call, you can let all calls fail, Emacs will automatically make your function signal if you return with a pending error state.
- Kind of verbose...

User-pointers

- New Lisp object type to wrap arbitrary pointer
- Possibility to set a finalizer to free ressources when garbage-collected
  - make_user_ptr()
  - {get,set}_user_ptr()
  - {get,set}_user_finalizer()

DEMO

praise the demo gods

Using other languages

- Emacs can load any valid module
- Doesn’t matter how the shared object is generated as long as the ABI is kept
- If your language can compile to .so and manipulate pointers, probably useable
  - Rust https://github.com/ubolonton/emacs-module-rs
  - Ocaml https://github.com/janestreet/ecaml
  - Go https://github.com/sigma/go-emacs
  - Nim https://github.com/yuutayamada/nim-emacs-module
  - ...

Documentation

- Philipp Stefani wrote an exhaustive documentation
- Not part of official doc for now
- Covers many corner cases and questions you might have
  - https://phst.github.io/emacs-module
- Chris Wellons has a couple of very interesting blog posts
  - Simple C module https://nullprogram.com/blog/2016/11/05/
  - Async requests in modules using unix signals https://nullprogram.com/blog/2017/02/14/
- Syohei YOSHIDA github
  - Plenty of modules
  - Perf analysis https://speakerdeck.com/syohex/dynamic-module

Existing modules

- Lib bindings
  - sqlite3
  - Capstone (multi platform, multi arch disassembler)
  - Csound
  - Openssl
  - Parson (JSON parser)
  - Zstd (compression lib)
  - yaml
- Embedded interpreters
  - Lua
  - Mruby
  - Perl

Future work: Foreign Function Interface

- Emacs can load shared lib if it implements the module interface
- In other words we need to write a module to use an existing library
  - e.g. libcurl doesn't implement Emacs module ABI, need to write a thin emacs-libcurl module that would translate and pass arguments to/from Lisp/libcurl
- What if we could load any lib *from* Lisp, skipping the module step?
- This is called a Foreign Function Interface
- Tom Tromey has some experimental code for that
  - https://github.com/tromey/emacs-ffi

Questions?