Andrea Leopardi

Using C from Elixir with NIFs

Erlang supports a way to implement functions in C and use them transparently from Erlang. These functions are called NIFs (native implemented functions). There are two scenarios where NIFs can turn out to be the perfect solution: when you need raw computing speed and when you need to interface to existing C bindings from Erlang. In this article, we're going to take a look at both use cases.

Note that if we want to use C bindings (i.e., we want to interface with an existing C program), then NIFs are not our only choice. Erlang has other ways to do foreign-function interface in order to talk to other languages. One example is ports; Sasa Juric wrote an excellent blog post about ports if you want to know more about them.

Here, we'll take a look at all-things-NIFs. First, we'll see how to write simple NIFs that perform calculations; then, we'll see how to use those NIFs from Elixir. Later on, we will see how to interface with existing C bindings from NIFs. Finally, we'll see how to integrate the C compilation step into the compilation of our Elixir code.

Most of the things I'll talk about here can be read in more detail in the Erlang documentation for the `erl_nif` C library.

The things discussed in this article apply both to Erlang as well as Elixir with minimal tweaking. I'll show all my examples in Elixir, but I'll refer to Erlang and Elixir indifferently.

Canonical NIF warning

NIFs are **dangerous**. I bet you've heard about how Erlang (and Elixir) are reliable and fault-tolerant, how processes are isolated and a crash in a process only takes that process down, and other resiliency properties. You can kiss all that good stuff goodbye when you start to play with NIFs. A crash in a NIF (such as a dreaded segmentation fault) will **crash the entire Erlang VM**. No supervisors to the rescue, no fault-tolerance, no isolation. This means you need to be extremely careful when writing NIFs, and you should always make sure that you have a good reason to use them.

Another thing worth noting is that NIFs are not preempted by the Erlang scheduler: a NIF runs as a single unit of computation and cannot be interrupted. This means your

NIFs should strive to be as fast as possible; as the Erlang documentation for NIFs suggest, a good rule of thumb is to keep NIFs under a millisecond of execution time. Check out the Erlang documentation for suggestions on what to do when your NIFs need more time to finish.

The basics

The way NIF works is simple: you write a C file and export a bunch of functions with the help of some Erlang-provided facilities, then compile that file. Then, you define an Erlang/Elixir module and call `:erlang.load_nif/2` in it. This function will define all the NIFs in the C file as functions in the calling module.

It's easier to see this in practice.

Let's start easy: let's write a NIF with no side effects that only takes a value in and returns a value out. For the purpose of this example, we'll write `fast_compare`, a function that takes two integers and compares them, returning `0` if they're equal, `-1` if the first is smaller than the second and `1` otherwise.

Defining a NIF

Let's start with the C side of things: we'll work on `fast_compare.c`. The first thing we have to do is include the `erl_nif.h` header file, which contains all the stuff we need (types, functions, and macros) to work with NIFs.

```
#include "erl_nif.h"
```

The C compiler won't know where `erl_nif.h` is so we'll have to specify that when we compile our program later on.

Now, every C file defining NIFs has a similar structure: there's a list of C functions, then a list of which of these C functions should be exported to Erlang/Elixir (and with what name), and finally a call to the `ERL_NIF_INIT` macro, which performs all the magic needed to actually hook things up.

For our example, the list of C function will only include the `fast_compare` function. The signature of this function looks like this:

```
static ERL_NIF_TERM
fast_compare(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv
```

```
  // cool stuff here
}
```

There are two NIF-specific types here: `ERL_NIF_TERM` and `ErlNifEnv`. `ERL_NIF_TERM` is a "wrapper" type that represents all Erlang types (like binary, list, tuple, and so on) in C. We'll have to use functions provided by `erl_nif.h` in order to convert an `ERL_NIF_TERM` to a C value (or multiple C values) and viceversa. `ErlNifEnv` is just the Erlang environment the NIF is executed in, and we'll mostly just pass this around without actually doing anything with it.

Let's take a look at the arguments for `fast_compare` (which are the same for all NIFs):

- `env`, as mentioned above, is just the Erlang environment the NIF is executed in and we won't care too much about it;
- `argc` is the number of arguments passed to the NIF when called from Erlang. We'll expand on this later;
- `argv` is the array of arguments passed to the NIF.

Reading Erlang/Elixir values into C values

We'll call `fast_compare` from Elixir like this:

```
fast_compare(99, 100)
#=> -1
```

When executing `fast_compare`, `argc` will be `2` and `argv` will be an array with the `99` and `100` values. These arguments however are of type `ERL_NIF_TERM`, so we have to "convert" them to C terms before being able to manipulate them. `erl_nif.h` provides functions to "get" Erlang terms into C terms; in this case, we need `enif_get_int`. The signature for `enif_get_int` is this:

```
int enif_get_int(ErlNifEnv *env, ERL_NIF_TERM term, int *ip);
```

We have to pass in the environment `env`, the Erlang term we want to "get" (which we'll take from `argv`) and the address of an integer pointer that will be filled with the Erlang integer value.

Turning C values to Erlang values

`erl_nif.h` provides several `enif_make_*` functions to convert C values back to Erlang values. They all have a similar signature (which is adapted to the type each function has to convert) and they all return a `ERL_NIF_TERM` value. In our case, we'll need `enif_make_int`, which has this signature:

```
ERL_NIF_TERM enif_make_int(ErlNifEnv *env, int i);
```

Writing the NIF

Now that we know how to go back and forth between Erlang values and C values, writing the NIF is straightforward.

```
static ERL_NIF_TERM
fast_compare(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv
  int a, b;
  // Fill a and b with the values of the first two args
  enif_get_int(env, argv[0], &a);
  enif_get_int(env, argv[1], &b);

  // Usual C unreadable code because this way is more true
  int result = a == b ? 0 : (a > b ? 1 : -1);

  return enif_make_int(env, result);
}
```

Wiring our C up

We now have to export the function we wrote to Erlang. We'll have to use the `ERL_NIF_INIT` macro. It looks like this:

```
ERL_NIF_INIT(erl_module, functions, load, upgrade, unload, rel
```

where:

- `erl_module` is the Erlang module where the NIFs we export will be defined; it shouldn't be surrounded by quotes as it will be stringified by the `ERL_NIF_INIT` macro (e.g., `my_module` instead of `"my_module"`);

- `functions` is an array of `ErlNifFunc` structs that defines which NIFs will be exported, along with the name to use as their Erlang counterpart and the arity;
- `load`, `upgrade`, `unload`, and `reload` are function pointers that point to hook functions that will be called when the NIF module is loaded, unloaded, and so on; we won't pay too much attention to these hooks right now, setting all of them to `NULL`.

We have all the ingredients we need. The complete C file looks like this:

```
#include "erl_nif.h"

static ERL_NIF_TERM
fast_compare(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv
  int a, b;
  enif_get_int(env, argv[0], &a);
  enif_get_int(env, argv[1], &b);

  int result = a == b ? 0 : (a > b ? 1 : -1);
  return enif_make_int(env, result);
}

// Let's define the array of ErlNifFunc beforehand:
static ErlNifFunc nif_funcs[] = {
  // {erl_function_name, erl_function_arity, c_function}
  {"fast_compare", 2, fast_compare}
};

ERL_NIF_INIT(Elixir.FastCompare, nif_funcs, NULL, NULL, NULL,
```

Remember we have to use the full Elixir module name atom in the `ERL_NIF_INIT` macro (`Elixir.FastCompare` instead of just `FastCompare`).

Compiling our C code

NIF files should be compiled to `.so` shared objects. The compilation flags vary between different systems and compilers, but they should look something like this:

```
$ cc -fPIC -I$(ERL_INCLUDE_PATH) \
     -dynamiclib -undefined dynamic_lookup \
     -o fast_compare.so fast_compare.c
```

With this command, we're compiling `fast_compare.c` into `fast_compare.so` (`-o fast_compare.so`), using some flags for dynamic code along the way. Note how we're including `$(ERL_INCLUDE_PATH)` in the include paths: this is the directory that contains the `erl_nif.h` header file. This path is usually in the Erlang's installation directory, under `lib/erts-VERSION/include`.

Loading NIFs in Elixir

The only thing we have left to do is load the NIF we defined in the Elixir `FastCompare` module. As the Erlang documentation for NIFs suggests, the `@on_load` hook is a great place to do this.

Note that for each NIF we want to define, we need to define the corresponding Erlang/Elixir function in the loading module as well. This can be taken advantage of in order to define, e.g., fallback code in case NIFs aren't available.

```
# fast_compare.ex
defmodule FastCompare do
  @on_load :load_nifs

  def load_nifs do
    :erlang.load_nif('./fast_compare', 0)
  end

  def fast_compare(_a, _b) do
    raise "NIF fast_compare/2 not implemented"
  end
end
```

The second term for `:erlang.load_nif/2` can be anything and it will be passed to the `load` hook we mentioned above. You can have a look at the docs for `:erlang.load_nif/2` for more information.

We're done! We can test our module in IEx:

```
iex> c "fast_compare.ex"
iex> FastCompare.fast_compare(99, 100)
-1
```

Examples in the wild

Writing “pure” NIFs (with no side effects, just transformations) is extremely useful. One example of this that I like a lot is the devinus/markdown Elixir library: this library wraps a C markdown parser in a bunch of NIFs. This use case is perfect as turning Markdown into HTML can be an expensive task, and a lot can be gained by delegating that work to C.

Something useful: resources

As I mentioned above, a great use of NIFs is wrapping existing C libraries. Often, however, these libraries provide their own data abstractions and data structures. For example, a C database driver could export a `db_conn_t` type to represent a database connection, defined like this:

```
typedef struct {
  // fields
} db_conn_t;
```

alongside functions to initialize a connection, issue queries, and free a connection, like this:

```
db_conn_t *db_init_conn();
db_type db_query(db_conn_t *conn, const char *query);
void db_free_conn(db_conn_t *conn);
```

It would be useful if we were able to handle `db_conn_t` values in Erlang/Elixir and pass them around between NIF calls. The NIF API has something just like that: **resources**. No better way to quickly explain what resources do than the Erlang documentation:

> *The use of resource objects is a safe way to return pointers to native data structures from a NIF. A resource object is just a block of memory […].*

Resources are blocks of memory, and we can build and return safe pointers to that memory *as Erlang terms*.

Let's explore how we could wrap the simple API sketched above inside NIFs. We're going to start with this skeleton C file:

```
#include "db.h"
#include "erl_nif.h"

typedef struct {
  // fields here
} db_conn_t;

db_conn_t *db_init_conn();
db_type db_query(db_conn_t *conn, const char *query);
void db_free_conn(db_conn_t *conn);
```

Creating resources

To create a resource, we have to allocate some memory with the help of the `enif_alloc_resource` function. This function is similar (in principle) to `malloc`, as you can tell by its signature:

```
void *enif_alloc_resource(ErlNifResourceType *res_type, unsign
```

`enif_alloc_resource` takes a resource type (which is just something we use to distinguish resources of different types) and the size of the memory to allocate, and returns a pointer to the allocated memory.

Resource types

Resource types are created with the `enif_open_resource_type` function. We can declare resource types as global variables in our C files and take advantage of the `load` hook passed to `ERL_NIF_INIT` to create the resource types and assign them to the global variables. It goes something like this:

```
ErlNifResourceType *DB_RES_TYPE;

// This is called everytime a resource is deallocated (which h
```

```
// enif_release_resource is called and Erlang garbage collects
void
db_res_destructor(ErlNifEnv *env, void *res) {
  db_free_conn((db_conn_t *) res);
}

int
load(ErlNifEnv *env, void **priv_data, ERL_NIF_TERM load_info)
  int flags = ERL_NIF_RT_CREATE | ERL_NIF_RT_TAKEOVER;
  DB_RES_TYPE =
    enif_open_resource_type(env, NULL, "db", db_res_destructor
}
```

Creating the resource

We can now wrap `db_init_conn` and create our resource.

```
static ERL_NIF_TERM
db_init_conn_nif(ErlNifEnv *env, int argc, const ERL_NIF_TERM
  // Let's allocate the memory for a db_conn_t * pointer
  db_conn_t **conn_res = enif_alloc_memory(DB_RES_TYPE, sizeof

  // Let's create conn and copy the memory where the pointer i
  db_conn_t *conn = db_init_conn();
  memcpy((void *) conn_res, (void *) &conn, sizeof(db_conn_t *

  // We can now make the Erlang term that holds the resource..
  ERL_NIF_TERM term = enif_make_resource(env, conn_res);
  // ...and release the resource so that it will be freed when
  enif_release_resource(conn_res);

  return term;
}
```

Retrieving the resource

In order to wrap `db_query`, we'll need to retrieve the resource that we returned in `db_init_conn_nif`. To do that, we'll use `enif_get_resource`.

```
static ERL_NIF_TERM
db_query_nif(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv
  db_conn_t **conn_res;
  enif_get_resource(env, argv[0], DB_RES_TYPE, (void *) conn_r

  db_conn_t *conn = *conn_res;

  // We can now run our query
  db_query(conn, ...);

  return argv[0];
}
```

Using resources in Elixir

Let's skip the part where we export the NIFs we created to a `DB` module and jump right into IEx, assuming the C code is compiled and loaded by `DB`. As I mentioned above, resources are completely opaque terms when returned to Erlang/Elixir. They're represented as empty binaries:

```
iex> conn_res = DB.db_conn_init()
""
iex> DB.db_query(conn_res, ...)
...
```

Since resources are opaque, you can't really do anything with them in Erlang/Elixir other than passing them back to other NIFs. They act and look like binaries, and this can even cause problems because they can be mistaken for just binaries. For this reason, my advice is to wrap resources inside structs. This way, we can limit our public API to only handle structs and handle resources internally. We also get the benefit of being able to implement the `Inspect` protocol for structs, which means we can safely inspect resources, hiding the fact that they look like empty binaries.

```
defmodule DBConn do
  defstruct [:resource]

  defimpl Inspect do
    # ...
```

```
  end
end
```

Compiling with Mix

Mix provides a feature called Mix compilers. Each Mix project can specify a list of compilers to run when the project is compiled. A new Mix compiler the perfect place to automate the compilation of our C source code. For the scope of this section, let's say we're building a `:my_nifs` Elixir application that will use NIFs from the `my_nifs.c` C source file.

First, let's create a `Makefile` to compile the C source (as we would probably do anyways).

```
ERL_INCLUDE_PATH=$(...)

all: priv/my_nifs.so

priv/my_nifs.so: my_nifs.c
  cc -fPIC -I$(ERL_INCLUDE_PATH) -dynamiclib -undefined dynami
```

This Makefile assumes `my_nifs.c` is stored in the root of your Mix project. We're going to put the `.so` shared object in the `priv` directory of our application so that it will be available in releases. Now, whenever we change `my_nifs.c` and run `$ make`, then `priv/my_nifs.so` will be recompiled.

We can now hook up a new Mix compiler that just calls `make`. Let's do this at the top of `mix.exs`:

```
defmodule Mix.Tasks.Compile.MyNifs do
  def run(_args) do
    {result, _errcode} = System.cmd("make", [], stdout_to_stde
    IO.binwrite(result)
  end
end
```

We call `IO.binwrite/1` in order to write whatever was the output of `$ make` on the terminal. In a real-world application, we obviously would want to check the result of the

`make` command, as well as make sure that `cc` and `make` are installed on the system and available in the path; here, we're just omitting those parts for simplicity.

We now need to add the `:my_nifs` compiler to the list of compilers for the `:my_nifs` application:

```
# in mix.exs
defmodule MyNifs.Mixfile do
  use Mix.Project

  def project do
    [app: :my_nifs,
     compilers: [:my_nifs] ++ Mix.compilers,
     ...]
  end
end
```

Now, whenever we run `$ mix compiler`, our C code will be recompiled (if necessary) automatically. This will also work when other libraries list `:my_nifs` as a dependency, as now running `make` is part of the compilation process for the `:my_nifs` project.

Conclusion

This was a long post, but I hope I covered most of the NIF universe in it. As you saw, using NIFs in Erlang/Elixir turns out to be fairly straightforward. As mentioned at the beginning of this article, NIFs are to be used carefully and are not always the right tool for the job because of their fragility (remember that NIFs can cause the entire Erlang VM to crash) and speed requirements.

Thanks for making it this far!

Written on December 5, 2015

Manuel Garandal • 10 months ago

Are you planning to update this blog post? Would be great! In any case, do you have better resources/tutorials regarding elixir/erlang interoperability with C/C++?

^ | v • Reply • Share ›

Andrea Mod ➚ Manuel Garandal • 9 months ago

Hey Manuel,

the Erlang [erl_nif](http://erlang.org/doc/man/e... documentation is pretty good. What do you mean by "update this blog post"? In what areas?

3 ^ | v • Reply • Share ›

Brent Shaffer • a year ago

Great article! I had to run the following command on Debian Jesse in order to avoid "undefined reference" errors:

```
cc -fPIC -I/usr/local/lib/erlang/usr/include ¥
    -Wl,-undefined -Wl,dynamic_lookup -shared ¥
    -o fast_compare.so fast_compare.c
```

^ | v • Reply • Share ›

Andrea Rossi • 2 years ago

Hey Andrea, any chance you could put the entirety of the code on a repository? The post is fantastic, but it'd be good to see the entirety of the C code in a file. Thank you!

^ | v • Reply • Share ›

Joshua Kidd • 3 years ago

Thanks for this! I've been trying to wrap my head around NIFs. One step closer to getting libsass working in Elixir :)

^ | v • Reply • Share ›