The Nifty Way to Call Hell from Heaven

Andreas Löscher Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden
andreas.loscher@it.uu.se konstantinos.sagonas@it.uu.se

Abstract

Often Erlang programmers want or need to use existing C libraries. Also, occasionally they have good reasons to implement parts of their applications directly in C. To cater for such situations, the Erlang/OTP system comes with various mechanisms to call C from Erlang. The most modern of them allows to call C functions from Erlang as natively implemented functions (NIFs). Unfortunately, the use of a NIF library currently requires writing by hand a fair amount of code that to a large extent is boilerplate. To ease the lives of Erlang programmers and simplify the task of using existing C code bases from Erlang, we have created Nifty, a tool that automates the process of creating NIF libraries from C header files containing declarations of types and functions that the library supplies. This paper describes the functionality and implementation of Nifty, its current limitations and our experiences with it so far.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed and parallel languages; D.2.12 [*Software Engineering*]: Interoperability

Keywords language interoperability; foreign language interface

1. Introduction

In many applications developed in functional programming languages, it is sometimes desirable or even necessary, to employ code implemented in lower-level languages, such as C. Consequently, implementations of most functional languages come with various interoperability mechanisms for programs written in other, less safe, languages. Erlang is no exception to this. In fact, the Erlang/OTP system offers multiple ways to call C code [1]. The most modern and fastest of them [3] allows to call C functions from Erlang as so called *natively implemented functions (NIFs)*. To use NIFs, the C code needs to be compiled and linked into a dynamically loadable shared library which is then loaded into the runtime system and can be called from Erlang code.

There exist many reasons why using NIFs in a language such as Erlang that claims to take fault tolerance seriously is not a good idea. Still, the 'need for speed' and the wish to take advantage of already existing libraries are sometimes hard to resist. To satisfy these urges, Erlang programmers are currently required to write by hand a fair amount of C and Erlang code that to a large extent is boilerplate. Despite our views on whether it is a good idea to use NIFs in Erlang applications, we felt that there is probably very little reason for their employment to continue to be tedious and painful. We also had an ulterior motive: we wanted to extend PropEr [23], a property-based testing tool, with the ability to test C libraries from Erlang, and needed some automatic way to connect Erlang with C. Rather than coming up with something specific to PropEr that would just satisfy only our own needs, we decided to create a general tool, called Nifty [10]. Starting from a header file with types and declarations of functions that a C library supplies, Nifty automatically creates all infrastructure required to call the functions of the library from Erlang.

Some of the characteristics of a language such as C make the creation of such a tool a non-trivial task. In the main part of this paper we describe the functionality and use of Nifty (Section 3), dangers that come with using NIFs and Nifty's support for dealing with them (Section 4), the most important aspects of Nifty's implementation (Section 5), its current limitations (Section 6), and some of our experiences from using Nifty (Section 7). Brief sections with related work and some concluding remarks follow. Let us however begin with a section describing NIFs and how these are currently supported in Erlang/OTP.

2. Native Implemented Functions

As mentioned, Erlang/OTP comes with a mechanism to enhance the runtime system with native implemented functions [3]. To write such a function one has to create a shared library in C that can be loaded from the Erlang/OTP system at runtime. In this shared library, which we will call a *NIF module* from now on, we can write C functions that we can call from Erlang. The arguments to these functions are Erlang terms that need to be translated to C to use them in NIFs. Erlang/OTP provides a NIF *library* that contains translation functions from Erlang terms to C datatypes for most of Erlang's basic types.

As a first example, let us suppose we want to use the function

```
int magic(int value);
```

from Erlang. Such a function specification is typically given in some C header file. For simplicity, let us assume that this function is implemented by some code like the one shown below.

```
#include "magic.h"

int magic(int value) {
  return value + 42;
}
```

In order to use this function from Erlang, we have to write a NIF module that wraps the function. A NIF can contain arbitrary C code which allows us to call other functions implemented in C. In order to call `magic()` with an integer argument, we have to translate Erlang's representation of integers into C's. In a similar manner

Erlang'16, September 23, 2016, Nara, Japan
ACM. 978-1-4503-4431-9/16/09...$15.00
http://dx.doi.org/10.1145/2975969.2975970

```c
static ERL_NIF_TERM
nif_magic(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
  int err;
  int retval;
  int arg;
  err = enif_get_int(env, argv[0], &arg);
  if (!err) {
    return enif_make_badarg(env);
  }
  retval = magic(arg);
  return enif_make_int(env, retval);
}

static ErlNifFunc nif_functions[] = {
  {"magic", 1, nif_magic}
};

ERL_NIF_INIT(magic, nif_functions, NULL, NULL, NULL, NULL);
```

(a) Hand-written NIF file that wraps `magic()`.

```c
static ERL_NIF_TERM
_nifty_magic(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
  int err = 0;
  int c_retval;
  ERL_NIF_TERM retval;
  int carg_0;
  err = enif_get_int(env, argv[0], &carg_0);
  if (!err) {
    goto error;
  }
  c_retval = (int)magic((int)carg_0);
  retval = enif_make_int(env, c_retval);
  return retval;
error:
  return enif_make_badarg(env);
}

static ErlNifFunc nif_functions[] = {
  {"magic", 1, _nifty_magic}
};

int upgrade(ErlNifEnv* env, void** priv_data,
            void** old_priv_data, ERL_NIF_TERM load_info) {
  return 0;
}

ERL_NIF_INIT(magic, nif_functions, NULL, NULL, upgrade, NULL);
```

(b) NIF file automatically generated by Nifty.

Figure 1. Two NIF files wrapping `magic()`.

```erlang
-module(magic).
-export([magic/1]).
-on_load(init/0).

init() ->
  erlang:load_nif("magic", 0).

magic(_) ->
  erlang:nif_error(nif_library_not_loaded).
```

Figure 2. An Erlang module to use `magic()` as a NIF.

we need to translate the C representation for integers into Erlang's when we want to pass the result of the function call back to Erlang.

In Figure 1(a) we show an implementation of a NIF file that wraps the C function `magic`. It is the code that a programmer would write following the NIF tutorial [3]. In line 6 we translate the Erlang term to a C integer. If this translation fails, we generate an exception in line 8. We call `magic()` in line 10 and translate the result back to an Erlang term in line 11.

The NIF file consists of a part that specifies which C functions are exported to Erlang (lines 14–16). Each specification starts by declaring the Erlang function name for the NIF (`"magic"` in this case), its arity, and the NIF function that it corresponds to.

We also have to declare the name of the Erlang module that these NIFs belong to through the initializer `ERL_NIF_INIT` (line 18). The name of this module is `magic` in this case. Through this initializer we can specify four functions that are called when the module is loaded for the first time, reloaded, upgraded, or unloaded. If the upgrade function is set to `NULL`, as is the case in Figure 1(a), then the upgrade mechanism of Erlang is disabled. An attempt to reload the module will result in an error as shown in the Erlang shell:

```
1> l(magic).
{module,magic}
2> l(magic).
{error,on_load_failure}
=ERROR REPORT==== 15-May-2016::18:19:49 ===
The on_load function for module magic returned
{error, {upgrade, "Upgrade not supported by this NIF library."}}
```

The NIF file can not be used standalone. We rather have to supply an Erlang module with the same name, like the one shown in Figure 2. This module must contain stubs for all functions that are exported from the NIF module. We usually want to use the NIF functions and bind the loading of the NIF module to the `on_load` event of the Erlang module. Before we load the NIF module, the stub functions are called. After we loaded the NIF module, all calls are directed to the NIF module. If we call a NIF we abandon the safety of Erlang. The C function can have side-effects, memory leaks or even crash the Erlang VM.

Let us also present a more involved piece of C code as example. Figure 3 shows a part of the header file used in the C library snappy-c [19], namely definitions for a structure and a function. The translation between C and Erlang types to call the function `magic()` was straightforward. For the function `snappy_compress()` the type translation is more complicated. The function reads `input_length` bytes of the data referenced by `input` and writes the compressed data to `compressed`. The length of the compressed data is written to the integer referenced by `compressed_length`. It also utilizes a `struct snappy_env`. We are faced with three problems when we want to write a wrapper for this function.

The first problem is that some types (e.g. `size_t`) are defined in another header file. If we want to translate them, then we need to look them up to see how they are represented.

The second problem is that the function uses pointers. Pointers in C are an address that usually points to some memory area containing data. This is the case for the argument `input`. The pointer does not store information about the size of the memory area. In C it is thus necessary to use some kind of escape sequence like in

```
#include <stddef.h>

struct snappy_env {
  unsigned short *hash_table;
  void *scratch;
  void *scratch_output;
};

int snappy_compress(struct snappy_env *env,
                    const char *input,
                    size_t input_length,
                    char *compressed,
                    size_t *compressed_length);
```

Figure 3. A part of the header file supplied with snappy-c.

NULL-terminated strings, or to specify the size of the memory area. In our example the size is specified by the argument `input_length`. Pointers are not only used to point to data that is processed by the function. They are also used to return values from functions, as is the case with the buffer referenced by the argument `compressed`. `compressed` points to a pre-allocated memory area in which the compressed data is to be written. This means, that we have to allocate and free memory to use the function.

The third problem we have to handle is the usage of a **`struct`**. There is no obvious translation between Erlang and C when it comes to **`struct`**s, meaning that we have to come up with one.

Writing a NIF module that translates a simple function like `magic()` requires several lines of C code. Wrapping a more complex function like `snappy_compress()` requires even more effort.

The snappy-c library is a small C library which exports only nine functions. Writing a wrapper for only one of the functions is already a time consuming and tedious task, let alone writing a wrapper for all of them. In the next section we will introduce Nifty, a tool that automates the generation of NIF modules.

3. Nifty

Nifty is a tool that interfaces Erlang with conventional C libraries using NIF modules. It generates NIF modules using the definitions from a regular C header file and provides an automated translation from Erlang terms to C datatypes using the type information in the header file. It also comes with a utility library that supports the handling of C data structures built from pointers and **`struct`**s.

3.1 Using Nifty

Nifty can be used to create NIF modules from the examples we introduced in Section 2 using the `nifty:compile` command from the shell:

```
1> Opts = [{port_specs, [{".*", "$NIF", ["magic.c"]}]}],
   nifty:compile("magic.h", magic, Opts).
generating...
==> magic (compile)
Compiled src/magic.erl
Compiled src/magic_remote.erl
Compiling c_src/magic_nif.c
ok
```

As shown, the `nifty:compile/3` function takes three arguments:

1. The header file that is used to generate the NIF module.
2. The name of the generated NIF module.
3. A list of options, which are compatible with rebar options.

Nifty reads the header file and generates a NIF for each defined function. Nifty generates a complete Erlang package that contains the source files for the NIF module (here, these files are `magic_nif.c`, `magic_remote.erl` and `magic.erl`), an Erlang header file, an application resource file, and a rebar config file. After the files are generated, Nifty uses rebar [11] to compile them, and makes them available in Erlang. We can call the generated functions directly after `nifty:compile/3` returned **`ok`**:

```
2> magic:magic(17).
59
```

Figure 1(b) shows the C code that Nifty generates to wrap the C implementation of `magic()`. Compared to the hand-written code, the generated one is a bit more verbose but has more functionality. The error handling is centralized at the end of the function. Nifty generates some module specific utility functions for the support library and passes an `upgrade` function to the initializer, which enables Erlang's upgrade mechanism. A reload of the module will upgrade the Erlang part of the NIF module but not the C part. More importantly, a reload of the NIF module will not result in an error.

In Section 2 we introduced a more complex example, showing a part of the snappy-c library interface. To generate a NIF module for snappy-c, we have to make sure that we build the source code of snappy-c correctly. This means we have to set some defines and some compiler flags. Using Nifty, we can generate a NIF module for snappy-c with this command:

```
3> Opts = [{port_specs,
           [{".*", "$NIF", ["snappy-c/snappy.c"],
             [{env, [{"CFLAGS", "$CFLAGS -Isnappy-c/ -DSG=1"},
                     {"LDFLAGS", "$LDFLAGS -DSG=1"}]}]}]}],
   nifty:compile("snappy-c/snappy.h", snappyc, Opts).
generating...
==> snappyc (compile)
...
ok
```

Figure 4 shows parts of the function generated by Nifty to wrap `snappy_compress()`. After the declaration of the local variables the code to translate the first argument is shown starting in line 7. The first argument is a pointer to a struct. Nifty represents C pointers as two-tuples in Erlang, with the first element being the address of the pointer, and the second element being a string representing the type. Lines 23–26 show the code that Nifty generates to translate the third argument. This argument is of type `size_t` which is defined in the header file `stddef.h`. Nifty resolves this type to **`unsigned long`** and uses the correct NIF library function to translate the term. An exception is raised if the argument cannot be translated to a C **`unsigned long`** during runtime.

Although we have now generated a NIF module that we can use to access the snappy-c functions, we have to be careful in doing so. NIFs come with a big warning tag and can cause serious problems to the rest of the Erlang VM if not used properly. In Section 4 we will describe how those problems can be addressed using Nifty.

3.2 Nifty Types

One of Nifty's main tasks is to translate Erlang terms to C types. For every argument type of the C functions we want to call, we need a rule describing how to translate the type from Erlang to C and vice versa.

Type Names Nifty uses its own representation of type names. Type names are represented as strings and are divided into two parts. The first part is the module name of the module, that defines the type. This is the name which is given to `nifty:compile/3` as second argument. The second part is the C type. Both types are connected with a ".". The module part can be omitted for base types. A valid type name for an **`int`** would therefore be `"nifty.int"` or just `"int"`. The typename for `size_t` from the second example is `"snappyc.size_t"`.

```c
static ERL_NIF_TERM
_nifty_snappy_compress(ErlNifEnv* env, int argc, const
      ERL_NIF_TERM argv[]) {
  int err = 0;
  int c_retval;
  ERL_NIF_TERM retval;
  ...
  if (!enif_compare(argv[0], enif_make_atom(env, "null"))) {
    carg_0 = 0;
  } else {
    err = enif_get_tuple(env, argv[0], &arity0, (const
          ERL_NIF_TERM**)(&tpl0));
    if (err) {
      if (arity0 > 2) {
        err = 0;
      } else {
        err = enif_get_uint64(env, tpl0[0], (uint64_t*)&carg_0);
      }
    }
  }
  if (!err) {
    goto error;
  }
  ...
  err = enif_get_ulong(env, argv[2], &carg_2);
  if (!err) {
    goto error;
  }
  ...
  c_retval = (int)snappy_compress((struct snappy_env *)carg_0,
                                  (const char *)carg_1,
                                  (size_t)carg_2,
                                  (char *)carg_3,
                                  (size_t *)carg_4);
  retval = enif_make_int(env, c_retval);
  return retval;
error:
  return enif_make_badarg(env);
}
```

Figure 4. Nifty generated function wrapping `snappy_compress()`.

Base Types Erlang and C have corresponding data types for most of the basic types. Table 1 shows how Nifty translates the C base types between Erlang and C. For better portability, Nifty also supports fixed length integer types of C99 like `uint64_t` or `int16_t`.

It is important to be aware of the different ranges of the integer types. Erlang uses arbitrary ranged integers, while integers in C and C99 have a fixed range. Using a too large integer in Erlang will result in an integer overflow in C. The value gets casted to the C type of the function argument. If the value exceeds the range of the type `long long` for signed integers or `unsigned long long` for unsigned integers, an exception is raised indicating a bad argument type. In a similar manner, the precision of Erlang `float()`, which corresponds to a `double`, gets reduced to C `float`, when the argument type is `float`.

Pointers and Memory Memory management is a large part of programming in C since it does not come with a garbage collector like Erlang. Memory has to be allocated and freed manually. Pointers are C types that reference a memory area and have a type indicating what the referenced memory area contains.

Nifty manages pointers and memory areas similar to C. Pointers are stored as two-tuples where the first element is the memory address of the referenced memory area and the second element is a Nifty type name. Erlang provides resource types which are opaque handles to memory objects. While this seems like an obvious choice for pointers it prevents pointer arithmetic that usually can be done in C.

```
4> Ptr1 = nifty:pointer("snappyc.struct snappy_env").
{140611836710104,"snappyc.struct snappy_env *"}

5> Rec = nifty:dereference(Ptr1).
#snappy_env{hash_table = {140611836706816,
                          "snappyc.unsigned short *"},
            scratch = {0, "snappyc.void *"},
            scratch_output = {140611791346656, "snappyc.void *"}}

6> M = nifty:as_type(nifty:mem_alloc(1024), "unsigned short *").
{140611836710232,"nifty.unsigned short *"}

7> NRec = Rec#snappy_env{hash_table = M,
                         scratch = null, scratch_output = null},
   Ptr2 = nifty:pointer_of(NRec, "snappyc.struct snappy_env").
{140611836711328,"snappyc.struct snappy_env *"}

8> nifty:free(Ptr1).
ok

9> nifty:as_type(
     nifty:mem_alloc(nifty:size_of("snappyc.struct snappy_env")),
     "snappyc.struct snappy_env *").
{140046394985752,"snappyc.struct snappy_env *"}
```

Figure 5. Using pointers and `struct`s with Nifty's support library.

Figure 5 shows examples of how pointers are used with Nifty. There are multiple ways of acquiring a pointer. We can create a pointer with the `nifty:pointer/1` function, which allocates a memory area of the size of the given type and returns a pointer to it. We can also create a pointer from a value by using the function `nifty:pointer_of/2`. This allocates a memory area, initializes it with the given value and returns a pointer to it. We can use `nifty:dereference/1` to read a value referenced by a pointer. The type information stored with the pointer is used to interpret the referenced memory.

In addition, the `nifty` module provides functions `mem_allocate/1` or `malloc/1`, `mem_read/2`, `mem_write/2`, `mem_copy/3` and `free/1` to allocate, read, write, and free memory areas. The `nifty:size_of/1` function can be used to get the size of a type.

Structs and Unions In C, `struct`s are complex data types that represent a continuous area of memory. The `struct` definition contains a list of fields. Each field has a name and a type that defines the structure of the memory area. The fields of a `struct` are mapped one after another in memory as opposed to the field in a `union` which are mapped to the same memory address.

Nifty represents `struct`s as Erlang records. For every `struct` Nifty creates an Erlang record definition in a header file created together with the NIF module. These Erlang records have the same name as the `struct`s and contain the same name, order, and type of the fields. The type of a field in a `struct` can be another `struct`, which allows for nested `struct` definitions. Nifty translates these nested `struct`s to nested Erlang records.

`union`s, like `struct`s, are represented as Erlang records by Nifty. Such a record contains an entry for each `union` field similar to `struct`s. All fields are set to the correct value when a `union` is translated to a record. If a record is translated into a `union` only one of the fields must be set to the desired value; the rest of the fields must have `undefined` as a value.

C allows to define incomplete `struct` and `union` types by omitting the field definitions. Other C files using this header file can (if they do not implement the full type) only create pointers to this type. Nifty behaves identically.

Table 1. Nifty's translation of basic types between Erlang and C.

C Type	Erlang Type	Nifty Type Name
`signed int` or `int`	`integer()`	`nifty.int`
`unsigned int`	`integer()`	`nifty.unsigned int`
`char`	`integer()`	`nifty.char`
`short`	`integer()`	`nifty.short`
`long`	`integer()`	`nifty.long`
`long long`	`integer()`	`nifty.long long`
`float`	`float()`	`nifty.float`
`double`	`float()`	`nifty.double`
`<type> *`	{ `integer(), "<module>.<type>"` }	`<module>.<type>`

4. Handling the Dangers of NIFs

4.1 Safe Execution

C is an unsafe language. Reading or writing invalid memory areas can easily result in a memory access exception (`SEGFAULT`) which usually crashes the whole Unix process. By default, NIF modules run in the same Unix process as the Erlang code using them. This has the consequence that erroneous C code can crash the Erlang node. Erlang aims to be a fault-tolerant language and using NIF modules breaks this paradigm.

C code that is executed via a NIF module can access the memory of the calling Erlang VM process. This can pose a security threat or stability issues if the C code is not from a trusted source.

Nifty provides a way to run all NIF modules in a separate Erlang node. This mode allows for fault-tolerant execution, meaning that a crashing NIF will not crash the Erlang node. It also provides means of memory protection in that the NIF can only access the memory of the remote Erlang node.

Nifty generates a `<modulename>_remote.erl` with each generated NIF module, which contains the same functions as the NIF module itself, but executes them in a separate Erlang node with a mechanism similar to Remote Procedure Calls [13]. Two additional control functions, `start/0` and `stop/0` are exported, that start up and stop the remote node. If a node crashes during a NIF call, the return value of the call is `{error,node_crashed}`.

```
%% Direct usage of a NIF that causes a segfault will crash the
%% calling Erlang VM.
10> segf:segf().
Segmentation fault (core dumped)

%% Remote execution of the same NIF is safe.
%% The first call below makes sure epmd is running and starts
%% net_kernel, which switches on distribution, and creates
%% another node where the NIFs of the segf module will now be
%% run.  The NIF that crashes now brings down that other node.
11> segf_remote:start().
ok
(p33_p0_master@cola-light)12> segf_remote:segf().
{error,node_crashed}
(p33_p0_master@cola-light)13> segf_remote:segf().
{error,node_down}
```

When we run the NIF in a remote node, we need to run Nifty's support library in the same node. This is achieved by using the module `nifty_remote` instead of `nifty`.

The control functions `start/0` and `stop/0` can also be used to "reset" the NIF module. Calling `stop/0` causes the termination of the process that loaded the module. This frees the allocated memory and other resources used by the system. By calling `start/0` again, a fresh NIF module is loaded, undoing all side effects.

Nifty manages remote nodes per Erlang process. That means that two NIF modules that are used via the remote module are run on the same node and can share resources like pointers and memory. Two different processes that run the same NIF module will not be able to share these resources in the same way since each of the runs an independent instance of the NIF in a separate node.

Running NIF modules in separate Erlang nodes makes it possible to use NIFs without sacrificing robustness of Erlang applications. Crashes in the NIF module do not bring the main application node down and can be handled by checking the return value of the called functions and employing some mechanism provided by Erlang to restart a node that crashed.

4.2 NIF Scheduling

Another downside of NIFs is that they can disturb the normal operation of the VM by spending too much time in the called C functions. The NIF documentation [2] describes the disturbances as a degraded responsiveness and miscellaneous strange behaviors such as (but not limited to) "extreme memory usage, and bad load balancing between schedulers". As guideline for an acceptable time spend in a NIF the documentation states one millisecond. This is not always possible. For example, a compression library like snappy-c will spend much more time doing its work when subjected to a large amount of data. A NIF interface that is generated using Nifty does not alter the performance of the underlying C library. This means that we have no influence over the timing behavior of the functions of that library.

Fortunately, Erlang comes with an experimental feature called *dirty schedulers* that allows us to call long-running C functions in a NIF module. *Dirty schedulers* execute a NIF call on a special scheduler that does not interfere with Erlang's standard schedulers. Steve Vinoski's talk [25] on dirty schedulers gives an excellent overview of the problems that these schedulers solve and how they work. It is important to note that there are two kinds of dirty schedulers. One for IO bound tasks and one for CPU bound tasks. The reason is that there are only as many dirty schedulers for CPU heavy tasks as there are for the Erlang VM which limits the performance impact of long-running NIFs to a certain degree.

Nifty can generate NIF modules that use dirty schedulers. In fact, it is possible to enable dirty scheduler support globally for all functions of the NIF module or on a per-function basis. The global option configures all functions as CPU bound. This decision was made to conservatively preserve the computational power for the rest of the system. The per-function configuration allows for both types of dirty scheduling and overwrites the global configuration. The following instruction generates a NIF module for snappy-c with dirty scheduler support for all functions:

```
3> Opts = [{port_specs,
            [{".*", "$NIF", ["snappy-c/snappy.c"],
              [{env, [{"CFLAGS", "$CFLAGS -Isnappy-c/ -DSG=1"},
                      {"LDFLAGS", "$LDFLAGS -DSG=1"}]}]}]},
           {nifty, [schedule_dirty]}],
   nifty:compile("snappy-c/snappy.h", snappyc, Opts).
generating...
==> snappyc (compile)
...
ok
```

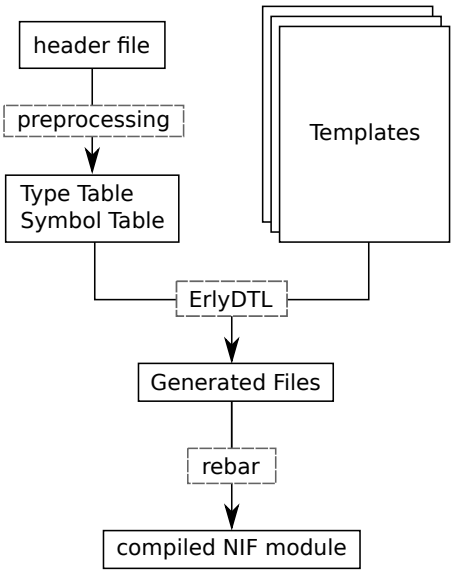


Figure 6. NIF generation process of Nifty.

A second option for using long-running C functions is to run the NIF in a remote node using the `_remote` module. The delay of the C function will still interfere with the remote node but the effects are limited to it. The local node will not be affected by the running C function. This comes however with the tradeoff that potential input and output data have to be sent from the local node to the remote node and back. But using the remote node offers a way to avoid threats for the Erlang VM coming from NIFs.

5. Implementation

Nifty generates NIF modules in three steps:

- In a *preprocessing* step, Nifty reads the given C header file and collects information about the defined types and functions. This information is stored in a type table and a symbol table.
- The second step uses the type and symbol table to *generate* the files for the NIF module.
- In the last step Nifty *compiles* the generated files, and makes them available to Erlang.

Figure 6 shows an overview of Nifty's code generation process.

5.1 Preprocessing

Parsing Nifty uses LibClang [8] to parse the C header files. LibClang is a high-level C interface to the Clang compiler [5]. It invokes a C preprocessor resolving macros and generates an abstract syntax tree (AST). The library provides an interface to iterate over the tree and collect the information about the definitions.

Nifty iterates over the AST and collects relevant information about all defined functions and types. During this process we can decide if we want to descend deeper into the tree or if we want to continue with the next node. By descending we can collect more information about the current node, e.g. in case of a node describing a function we can gather information about the function's arguments.

Nifty interfaces with LibClang using a NIF module and processes the C header into a symbol table, a list of the used types, and a constructor table. The symbol table stores for each function definition the types of the arguments and the return type. The information about the arguments include the position of the arguments. We use the position instead of the name to refer to the argument, since it is possible to omit the argument names in the function declaration. Table 2 shows the symbol table for the snappy-c example header file.

The constructor table stores additional information for the supported user defined types. For `struct`s it is the type information of the fields, for `enum`s the association between names and values, and for `typedef`s the underlying type. Table 3 shows the constructor table for snappy-c.

Type Tables The next step of the preprocessing phase is the construction of a type table that contains detailed information about all used types. Table 4 shows the type table built from parsing `snappy.h` as shown in Figure 3. The type table associates a type name with a type definition. The type definition can be either a base type definition, a user-defined type, a type alias, or a `struct` definition.

A base type definition contains the arithmetic type specifier and the optional specifiers. Arithmetic type specifiers are `char`, `int`, `float`, and `double`. For `char` and `int` types the optional type specifiers `signed`, `unsigned`, `short`, and `long` are used. This information is necessary since it determines how the base types are represented and how we need to translate them from Erlang to C. For `float` and `double` those fields are ignored. For `char` the size specifier is ignored. A pointer to a base type is also considered a base type.

Type aliases store the name of the type they are an alias of. A user-defined type is a type that is defined in the header file and not a base type. Pointers to user-defined types and type aliases are also user-defined types. `struct` types are user-defined types. Their type entries contain a reference to a `struct` definition which is stored in the constructor table.

After the initial generation of the type table, it can happen that the table contains incomplete or invalid types. These can be type aliases that do not end with a basic or user defined type or empty `struct`s. A type checker checks the validity of all types in the type table and removes them. After a type is removed, other depending types like fields of a `struct` may become unresolvable. These dependent types are also removed from the type table until all types can be resolved to valid types.

After the type table has been checked, all functions that rely on invalid types are removed from the symbol table. For each removed function, Nifty prints a warning. In such a case, the return value of `nifty:compile/3` indicates that functions are missing:

```
14> Opts = [{port_specs, [{".*", "$NIF", ["magic.c"]}]}],
     nifty:compile("faulty_header.h", fheader, Opts).
generating...
Warning: Unable to translate function "missing_function"() !!!
==> fheader (compile)
Compiled src/fheader.erl
Compiled src/fheader_remote.erl
Compiling c_src/bla_nif.c
{warning,{not_complete,["missing_function"]}}
```

5.2 Generation

Nifty uses the ErlyDTL [6] template engine to generate the code for the NIF module. ErlyDTL creates BEAM code from templates written in the Django [16] Template Language (DTL). Using a template engine allows us to describe static and dynamic parts of the render output in one domain specific language.

Django Template Language (DTL) is a text based template language. A template is a text file that contains static text, variables, and tags. Variables are written like this:

```
{{variable}}
```

Table 2. Symbol table for `snappy.h`.

Symbol	Entry
`"snappy_compress"`	`[{return,"int"}, {argument,"0","struct snappy_env *"}, {argument,"1","const char *"}... ]`

Table 3. Constructor table for `snappy.h`.

Type	Entry

`{struct,"snappy_env"}`	`[{field,"hash_table","unsigned short *",0},`
	`{field,"scratch","void *",1},`
	`{field,"scratch_output","void *",2}]`

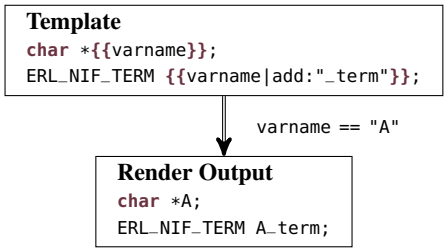
Table 4. Type table for `snappy.h`.

Type	Entry
`"char *"`	`{base,["*","char","signed","none"]}`
`"const char *"`	`{base,["*","char","signed","none"]}`
`"size_t"`	`{typedef,"unsigned long"}`
`"size_t *"`	`{userdef,["*","size_t"]}`
`"struct snappy_env"`	`{userdef,[{struct,"snappy_env"}]}`
`"struct snappy_env *"`	`{userdef,["*",{struct,"snappy_env"}]}`
`"unsigned long"`	`{base,["int","unsigned","long"]}`
`"unsigned short *"`	`{base,["*","int","unsigned","short"]}`
`"void *"`	`{base,["*","void","signed","none"]}`

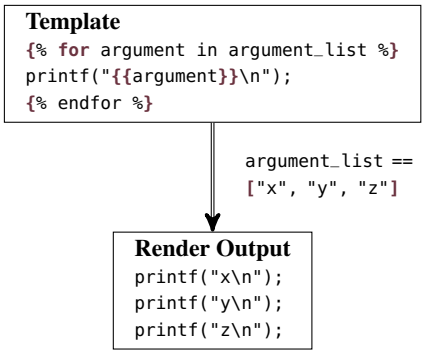
We can modify the value of the variable inside the template engine using filters. Filters can also have one argument:

```
{{variable|filter:argument}}
```

If the template engine encounters a variable during rendering, it is evaluated and exchanged by the value:



Tags are used to perform complex operations in the template like producing text output, loading additional information, or as control structures like loops:



ErlyDTL provides the possibility to extend the tags and filters of the template language by custom ones. Nifty uses custom filters and tags that support basic operations on dictionaries, type tables, and types.

Template Rendering The templates Nifty uses for generating the NIF module are static. ErlyDTL compiles the templates to an Erlang module in BEAM code form. We can load this generated module and call its `render/2` function with the variables we want to render the template. The function returns an `iolist()` that we write into a file.

The generated files form a complete Erlang package with header file, application resource file and rebar build file. This package can be directly used as part of another project, for example as dependency in rebar.

5.3 Compilation

We compile the generated files with rebar [11], an Erlang build tool that supports the building of NIFs. We generate a `rebar.config` file that contains the build information for the generated package. Compiling and linking arbitrary C files or libraries together with the generated code can be a tedious task; however, rebar already provides all the functionality we need. The options that are given to `nifty:compile/3` as a third argument are used as rebar options.

As a side comment, rebar is supposed to be used as a script from the command line instead of as a library. This made it necessary to implement an API that allows us to execute rebar commands based on the original rebar implementation.

5.4 Templates

Besides the generation of the type table, the templates are the core of Nifty.

Basic Type Templates Nifty uses separate templates for each base type. These templates define how a translation between Erlang and C is implemented in the generated NIF module. To translate a data type from Erlang to C it might be necessary to use additional variables or to allocate memory. The translation function from the NIF library has to be called and the retrieved value has to be placed as argument in the call to the wrapped C function. Figure 7 shows the template for translation of `float` and `double` types. The translation happens in multiple phases.

```
/* preparation phase                                    */
{% if phase=="prepare" %}
  {% if argument|is_argument%}
  double {{carg}};
  {% endif %}

  {% if argument|is_return %}
  double c_retval;
  ERL_NIF_TERM retval;
  {% endif %}
...
{% endif %}

/* translation from Erlang to C                         */
{% if phase=="to_c" %}
  err = enif_get_double(env, {{erlarg}}, &{{carg}});
{% endif %}

/* argument emmition phase                              */
{% if phase=="argument" %}
  {% if argument|is_argument %}
({{raw_type}}){{carg}}
  {% else %}
({{type}})
  {% endif %}
{% endif %}

/* translation from C to Erlang                         */
{% if phase=="to_erl"%}
  {{erlarg}} = enif_make_double(env, {{carg}});
{% endif %}

/* clean up phase omitted                               */
{# no cleanup phase #}
```

Figure 7. Template for the base type float.

The first phase is preparation. Here, every local variable which is later used has to be declared. No code should be executed in the preparation phase. Since the wrapped functions use multiple arguments, it is important to create those local variables with a unique name. The templates Nifty uses are designed to pass this name in the variable `{{{carg}}}` to the type template.

The second phase is translation. In this phase arbitrary code can be executed to translate the Erlang term into a C data structure and is achieved by using Erlang/OTP's NIF library. This phase performs basic type checking during runtime. The type template sets the local variable `err` to `0` to indicate an error.

The third phase emits the argument to the wrapped C function. This phase can be used to cast the variable containing the value to the correct type or to make last inline adjustment of the value. The code emitted here is directly used in the function call, hence must be an expression.

The fourth phase translates the return value of the wrapped C function into an Erlang term and stores it into `{{erlarg}}`. The return value in this phase is stored in `{{carg}}`. This phase is implemented in a general way and must not use the local variables `retval` and `c_retval`.

The final phase is used to clean up and free used memory.

Return value and function arguments are handled in equivalent ways in most parts of Nifty. In the base type templates however, they have to be handled differently. Nifty uses custom filters to determine if the current argument is a function argument or the return value.

The base type templates are also used to translate the fields of `struct`s. The `struct` fields have two different preparation phases. One phase is used to translate from Erlang records to C `struct`s, and the other one is used to translate in the other direction.

Function Templates The function template generates a NIF for all functions stored in the symbol table. To translate the types we invoke the five phases of the base type templates. We have to set a number of variables in order for the type template to be able to render correctly. This includes setting the correct phase identifier, generating a unique name, and preparing the type information. These steps have to be done for each phase. Figure 8 shows them for the translation phase. The template code of the other phases looks similar.

The type template can set the local variable `err`. The error handling code gets executed if `err` equals `0`. This will leave the function and raise an exception. The cleanup code will not be executed in this case. This means that the type templates have to handle the cleanup themselves in case of an error.

Some of the variable names that the type templates have to process are fixed. The return value of the NIF function has to be placed in the local variable `retval`. The return value of the wrapped C function is always placed in `c_retval`, which has to be declared by the base type template.

The function does not invoke the type templates directly. It includes a `build_type.tpl` template, which processes the type information and then includes the correct base type template.

Struct Templates Nifty has a `struct` template that works similar to the function template. Instead of translating function arguments and return values, it translates the fields of the `struct`.

The `struct` template defines two translation functions for each `struct`: one that translates a pointer into an Erlang record and one that translates an Erlang record into a pointer. Nifty allocates memory in size of the struct when it translates an Erlang record into a pointer.

Those two functions use the same five phases to translate the fields from Erlang terms to C data types as the functions. The preparation phase is different for each direction. In addition to this, only four of the phases are triggered for each function. If a pointer is translated into an Erlang record, the translation phase from C to Erlang is skipped and vice versa.

The basic type template for structs invokes the two translation functions, which work in the same way as the NIF library functions for other types.

Putting everything together The base template for the NIF module includes the function and `struct` templates and adds them to the export list. It also contains include directives for the NIF library and C standard libraries. Figure 9 shows the relevant parts of the base template.

Nifty uses the templates to generate the rest of the files needed for the Erlang package.

After the templates have been rendered, Nifty creates a package directory structure and saves the render output in the designated files. The resulting package structure can be seen in Figure 10.

6. Current Limitations

6.1 Anonymous Structs and Unions

In C it is possible to define `struct`s and `union`s without a name. This creates a type without name. Those anonymous types are often used together with a `typedef`, where the nameless type is immediately bound to a type alias. Nifty does not support anonymous `struct`s or `union`s at the moment. Another limitation is that Nifty does not support nested `struct` or `union` definitions as shown in Figure 11(a). Most of the time it is however possible to use an equivalent definition like the one in Figure 11(b), that can be parsed by Nifty.

```
{% with fn=symbols|fetch_keys %}                                    /* fetch the function names                          */
  {% for name in fn %}                                              /* iterate over all functions                        */

static ERL_NIF_TERM
_nifty_{{name}}(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
...
  {% with arguments=symbols|fetch:name %}                           /* get the arguments                                 */
    {% for argument in arguments %}                                 /* iterate over arguments                            */
      {% if argument|is_argument %}                                 /* if it is an argument translate it to c            */
        {% with raw_type=argument|getNth:3 phase="to_c" %}          /* get the type of the argument and set the phase    */
          {% with type=raw_type|resolved:types %}                   /* use the resolved type                             */
            {% with N=argument|getNth:2 %}                          /* get the position of the argument                  */
              {% with carg="carg_"|add:N erlarg="argv["|add:N|add:"]" %}  /* build a unique name and extract the term      */
                {% include "lib/builtin_type.tpl" %}                /* use the base type template to perform the phase   */
            {% endwith %}                                           /* closing tags                                      */
          {% endwith %}
        {% endwith %}
  if (!err) {                                                       /* is an error occured, jump to error                */
    goto error;
  }
        {% endwith %}
      {% endif %}
    {% endfor %}
  {% endwith %}
...
  return retval;
error:
  return enif_make_badarg(env);
}

  {% endfor %}
{% endwith %}
```

Figure 8. The template for a function body and the "translation to C" phase.

6.2 Function Pointers

C pointers can be used to reference a function. These function pointers are usually used to implement callback functions. Function pointers do not differ (besides their type) from other pointers in C. Currently, Nifty does not support functions pointers directly. All function pointer types default to `void *`. A function pointer can be obtained as result of a function call and passed to as a function argument. Nifty handles them as regular pointers.

Erlang/OTP's NIF interface does not provide us with a way of calling Erlang functions from C. It is therefore currently not possible to use Erlang functions for function pointers.

6.3 Variable Argument Lists

C functions can be defined with a variable number of arguments. The most prominent example is probably `printf()`. There are two ways of defining a variable number of arguments in C. The first is to write "`...`" after the last regular function argument. The second is to use the type `va_list` as argument type.

Nifty handles these ways of defining a variable number of arguments differently. If the function is defined with three dots (`...`), then Nifty translates the function without any additional arguments. If the function is defined with `va_list`, then Nifty does not render the function. The type `va_list` cannot be resolved, resulting in it being removed from the type table by the type checker. The functions using this type are simply removed from the symbol table and a warning is issued.

7. Some Experiences

Nifty has been around for about two years now. During this time, both us and various users have used the tool to generate interfaces for a variety of C libraries in order to test its capabilities and understand its limitations. Our experiences so far are positive. Even though there exist C constructs that currently cannot be handled, Nifty can automatically generate interfaces for many non-trivial code bases.

For example, we have created NIF interfaces for ZeroMQ [20] and Mongoose [15]. ZeroMQ is a messaging library written in C and C++ consisting of a total of 35 330 lines of code. Its header file, `zmq.h`, consists of 416 lines and contains definitions for 45 functions. Nifty can process this file out of the box and generates a NIF file of 3 888 lines. The functions in the ZeroMQ library can then directly be used as NIFs and all the examples in the ZeroMQ tutorial work as expected. It is important to note that ZeroMQ offers blocking communication for sending as well as receiving operations. Blocking means that the corresponding functions (like `zmq_send()` and `zmq_recv()`) will not return until the operation has successfully completed. We highly advise to use dirty schedulers for such kinds of blocking operations.

Mongoose is an easy to use web server of 4 854 lines of code. Its header file consists of 146 lines of code defining 30 functions. Nifty can process this file automatically and generates a NIF file of 3 696 lines. However, the library part of Mongoose relies on an event handling C callback function that the user needs to provide to the library. Nifty does not have a way of using an Erlang function as C callback function. However, provided one supplies a callback implemented in C, the library functions of Mongoose can be employed as NIFs without any issues.

We used a modified version of Nifty to generate a remote procedure call interface on sensor networks [22]. To achieve this we exchanged the templates to target the Contiki [17] operating system instead of Erlang's NIF interface. Nifty's parsing of the C header

```
#include <erl_nif.h>
#include <string.h>
#include <stdint.h>
...
/* Structs                                                    */
{% include "structures.tpl" %}

/* Functions                                                  */
{% include "functions.tpl" %}

/* Function definitions for ErLang                            */
static ErlNifFunc nif_functions[] = {
  {% with fn=functions|fetch_keys %}
    {% for name in fn %}
  {"{{name}}",                                  /* Erlang name */
   {{ functions|fetch:name|getNth:2|length }}, /* arity       */
   _nifty_{{name}}},                            /* C name      */
    {% endfor %}
  {% endwith %}
  /* functions used by nifty support library                   */
  {"erlptr_to_record", 1, erlptr_to_record},
  {"record_to_erlptr", 1, record_to_erlptr},
  {"new", 1, new_type_object},
  {"size_of", 1, size_of}
};

int upgrade(ErlNifEnv* env, void** priv_data,
            void** old_priv_data, ERL_NIF_TERM load_info) {
  return 0;
}

ERL_NIF_INIT({{module}}, nif_functions, NULL, NULL, upgrade, NULL);
```

Figure 9. Base template for the NIF module.

```
<module>/
├── c_src/
│   └── <module>_nif.c
├── ebin/
│   └── <module>.app
├── include/
│   └── <module>.hrl
├── src/
│   ├── <module>.erl
│   └── <module>_remote.erl
└── rebar.config
```

Figure 10. Directory tree of the generated Erlang package.

```
typedef struct {
  struct nested {
    int a;
    int b;
  } f1;
  char* f2;
} my_struct;
```

(a) Anonymous and nested `struct` definitions cannot be parsed by Nifty.

```
struct nested {
  int a;
  int b;
};

typedef struct my_struct {
  struct nested f1;
  char* f2;
} my_struct;
```

(b) A definition that can be parsed.

Figure 11. Two equivalent definitions of the type `my_struct`.

files and its handling of types remained the same even though the targeted architecture is 16-bit micro-controllers.

Even though Nifty does most of the job in automatically creating the NIF interface for a C library, the functions that the NIF module exports are the one-to-one equivalent of the libraries C functions. Erlang libraries usually offer a much higher level of abstraction than C libraries and do not require things like memory allocation or similar.

However, quite often, one can build higher-level and easier to use abstractions with the exported functions from the NIF module. As a case in point we will again use the code of snappy-c. Its header file contains a total of nine functions. The two main ones are the `snappy_compress` function, which is also shown in Figure 3, and `snappy_uncompress`. The remaining functions are auxiliary; three of them are `snappy_init_env`, `snappy_max_compressed_length` and `snappy_uncompressed_length`.

Figure 12 shows an Erlang module that utilizes the generated NIF module to provide an interface to snappy-c. The interface provides error handling and performs the allocation and cleanup of the needed data structures. The effort to implement this higher-level API to the compress and uncompress functions of Snappy is much lower than to implement the whole interface by hand.

This particular Erlang module contains 42 lines of code that need to be written by hand, while everything else is automatically generated by Nifty. For comparison, snappy-erlang-nif [4] is a package that provides a NIF interface to the C++ implementation of Snappy, that has the same functionality as the module shown in Figure 12. Its hand-written code consists of 462 lines of C and Erlang.

8. Related Work

For nearly all high-level languages there is a way similar to Erlang's NIF library that allows to integrate low-level code. For most of these languages there is also an automated way to generate an interface module.

The Simplified Wrapper and Interface Generator (SWIG) [12] is a tool that generates interfaces for a variety of programming languages like Python and Java. Unlike Nifty, which uses normal C header files, SWIG uses an interface file as input. This interface file is similar to a C header file, but needs to contain additional information like the module name. SWIG is a standalone tool and must be used from the command line unlike Nifty, which can be used directly from Erlang. Currently SWIG does not support Erlang NIFs as output format.

Foreign Function Interfaces are libraries that aim to allow access to the functions in a shared library from another programming language. The libraries do not generate any interface code, but call the function over a standardized interface like libffi [9]. However, the shared libraries contain no information about the argument and return types of the functions. The user has to specify those types when using the functions. For Erlang there exists an Erlang Enhancement Proposal (EEP 7) which proposes an FFI interface. FFIs are available for most high-level languages including Python (CTypes [21]), Haskell [7], ML [14], and others.

Long ago, the Erlang/OTP system contained an Interface Generator (IG) [18], an application that automatically generated code for port or socket communication between an Erlang program and a C program, given a C header file with certain keywords. This application however was abandoned in R6 and no running version of IG is available anymore.

QuviQ's commercial version of Erlang QuickCheck (eqc) contains a seamless interface to call C functions from Erlang [24]. In contrast to Nifty eqc is proprietary software which prevents us from comparing the two tools on a detailed level.

```
-module(snappy).
-export([compress/1, uncompress/1]).

compress(Data) ->
  Env = nifty:pointer("snappyc.struct snappy_env"),
  0 = snappyc:snappy_init_env(Env),
  Ibuf = nifty:mem_write(Data),
  Isz = sz(Data),
  Msz = snappyc:snappy_max_compressed_length(Isz),
  Obuf = nifty:mem_alloc(Msz),
  Osz = nifty:pointer("unsigned long"),
  Ret = case snappyc:snappy_compress(Env, Ibuf, Isz, Obuf, Osz) of
          0 -> nifty:mem_read(Obuf, nifty:dereference(Osz));
          F -> {fail, F}
        end,
  lists:foreach(fun nifty:free/1, [Env, Ibuf, Obuf, Osz]),
  Ret.

uncompress(Data) ->
  Ibuf = nifty:mem_write(Data),
  Isz = sz(Data),
  Osz = nifty:pointer("unsigned long"),
  case snappyc:snappy_uncompressed_length(Ibuf, Isz, Osz) of
    0 ->
      nifty:free(Ibuf),
      nifty:free(Osz),
      {error, snappy_unknown};
    1 ->
      OL = nifty:dereference(Osz),
      nifty:free(OSz),
      Obuf = nifty:mem_alloc(OL),
      Ret = case snappyc:snappy_uncompress(Ibuf, Isz, Osz) of
              0 -> nifty:mem_read(Obuf, OL);
              F -> {fail, F, OL}
            end,
      nifty:free(Ibuf),
      nifty:free(Obuf),
      Ret
    end.

sz(Data) when is_list(Data) -> length(Data);
sz(Data) when is_binary(Data) -> byte_size(Data).
```

Figure 12. An Erlang module utilizing the functions of Nifty and snappy-c to provide a better API to the Snappy library.

9. Concluding Remarks

With Nifty we introduced a tool that brings C and Erlang closer together by giving developers the opportunity to use C libraries in their Erlang programs. Nifty generates NIF modules automatically from header files making the C interface available in Erlang. This allows developers to focus on the more functional and interesting parts of their applications instead of writing wrapper code.

References

[1] Interoperability tutorial user's guide. http://www.erlang.org/doc/tutorial/users_guide.html, Apr. 2016.

[2] API functions for an Erlang NIF library. http://erlang.org/doc/man/erl_nif.html, Apr. 2016.

[3] Erlang Tutorial - NIFs. http://www.erlang.org/doc/tutorial/nif.html, Apr. 2016.

[4] An Erlang NIF wrapper for Google's snappy compressor/decompressor. https://github.com/fdmanana/snappy-erlang-nif, May 2016.

[5] Clang documentation - choosing the right interface for your application. http://clang.llvm.org/doxygen/group__CINDEX.html, Apr. 2016.

[6] ErlyDTL homepage. https://github.com/erlydtl/erlydtl/wiki, Apr. 2016.

[7] Haskell: Foreign Function Interface. https://wiki.haskell.org/Foreign_Function_Interface, May 2016.

[8] libclang. http://clang.llvm.org/doxygen/group__CINDEX.html, Apr. 2016.

[9] libffi: A portable foreign function interface library. http://sourceware.org/libffi/, May 2016.

[10] Nifty - NIF Interface Generator. http://parapluu.github.io/nifty/, May 2016.

[11] rebar homepage. https://github.com/rebar/rebar/wiki, May 2016.

[12] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, Berkeley, CA, USA, July 1996. USENIX.

[13] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/2080.357392.

[14] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1): 36–52, 2001. ISSN 1571-0661. . URL http://www.sciencedirect.com/science/article/pii/S1571066105804529. BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability (Satellite Event of {PLI} 2001).

[15] Cesanta. Mongoose embedded webserver library. https://www.cesanta.com/products/mongoose, May 2016.

[16] Django Software Foundation. Django template language. https://docs.djangoproject.com/en/1.9/topics/templates/, Apr. 2016.

[17] A. Dunkels, B. Grönvall, and T. Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE, 2004.

[18] Ericsson Utvecklings AB. The interface generator (IG). http://www1.erlang.org/documentation/doc-4.8.2/lib/ig-1.8/doc/index.html, May 2016.

[19] Google. snappy - a fast compressor/decompressor. http://google.github.io/snappy/, Apr. 2016.

[20] iMatix Corporation. 0mq code connected. http://zeromq.org/, May 2016.

[21] G. K. Kloss. Automatic C library wrapping Ctypes from the trenches. *The Python Papers*, 3(3):5, 2008.

[22] A. Löscher, K. Sagonas, and T. Voigt. Property-based testing of sensor networks. In *Sensing, Communication, and Networking (SECON), 2015 12th Annual IEEE International Conference on*, pages 100–108. IEEE, June 2015. . URL http://dx.doi.org/10.1109/SAHCN.2015.7338296.

[23] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5. . URL http://doi.acm.org/10.1145/2034654.2034663.

[24] QuviQ AB. Demo of Erlang QuickCheck testing C code. https://vimeo.com/104007760, May 2016.

[25] S. Vinoski. Tackling Dirty Jobs with Erlang's Schedulers. Code Mesh 2014 London, Nov. 2014.