SWI-Prolog dynamic C function calling

Jan Wielemaker
VU University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

February 18, 2018

Abstract

This document describes library(ffi), a library to wrap calls to C functions in shared objects (DLLs) dynamically. The library uses an external C preprocessor and the C header files to create the calling template, register types such as structures and enums and make the values for `#define` constants available from Prolog. Finally, the library provides predicates to allocate, write and read native C data structures.

Contents

1 Introduction

One of the oldest foreign interfaces for Prolog was provided by Quintus Prolog. In this interface the C functions that are to be made available from Prolog are, together with their parameter types, declared in Prolog. A subsequent call to `load_foreign_files/1-3` attaches the C library, making the functions available as predicates. This interface is also available from SICStus Prolog, Ciao and SWI-Prolog by means of library(qpforeign). There are several ways to realise the interface. The SWI-Prolog library generates C wrapper code, compiles this to a shared object and loads this. Other implementations may use a library for dynamically calling C functions similar to what is underlying the library that is the subject of this document. The Quintus interface only provides support for a limited set of C types: `long`, `double`, `single`, `char*` and the Prolog specific types for atom handles and term references (`atom_t` and `term_t` in SWI-Prolog).

Most native C library APIs rely on a much richer type system than what is supported by the Quintus interface. Realising a Prolog wrapper for such an API typically consists of two parts: a C part that wraps the original interface into one which simple types and operations and a Prolog part that recreates a high-level interface that handles complex data structures using multiple calls on the C layer.

With SWI-Prolog we took a different approach: a foreign predicate is a C function that takes an array of `term_t` Prolog terms and can control Prolog success, failure, exceptions and non-determinism. There is a rich set of C functions to examine `term_t` handles and make results available using *unification*. This approach has several advantages and disadvantages:

Flexible The interface allows for writing high-level Prolog predicates entirely in C.

Portable Making the functions available as Prolog predicates is controlled from C using `PL_register_foreign()`. As a result the dynamic loader only has to load the shared object can call a `void` function that takes no argumens. This functionality is easily provided on any modern OS.

Fast As the entire translation from complex Prolog data to C data and back is done in C based on efficient C low-level primitives, the result performs optimal.

Verbose Although not particularly hard, analysing and building Prolog terms from C is a rather verbose activity and requires C programming skills that are not widespread amoung Prolog programmers.

Error prone Error handling and disposing partly created C data structures after an error is encountered often results in complicated control flow. To some extend this can be avoided by using the C++ wrapper.

We promote SWI-Prolog as a *glue language*. Other languages in this area, such as Lua or Python, provide a dynamic calling interface using the same spirit as the above described Quintus interface, but with a wider coverage of supported types. This is in part facilitated because these languages have a closer resemblance to C, making the mapping more straight forward.

One of the issues is access to user-defined C types, *struct*, *union* and *union* and *enum*, as well as access to C *preprocessor* symbols (`#define`). The Python `ctypes` package fixes some of these issues by extracting information from the library debug information. Its package pyswip provides an interface to SWI-Prolog based on the `ctypes` package. Studying this package provided inspiration for the library described here.

Modern C APIs typically abstract from concrete C data types such as `int` or `long`. By defining their own type system and binding that centrally to the base types using `typedef` declarations the system is easily ported to different OSes and C compilers. For similar reasons they use a lot of preprocessor macros. From `pyswip/core.py` we learn that these definitions from the SWI-Prolog header `SWI-Prolog.h` are repeated in Python syntax and, as `pyswip` is a bit outdated, several of the definitions are now incorrect or do not deal with e.g., portability to 64-bit machines correctly. We need some way to extract up-to-date information from the library automatically.

All required information is available from e.g., the `gcc` debug information when compiled using `-gdwarf-2 -g3`. However, most libraries are not compiled this way and, if another C compiler is used, cannot be compiled this way. Often debug information is stripped from release binaries.

This library opts for a fairly portable route, but requires access to a C preprocessor and the header files that come with the library. This results in code as below to get access to the Linux statfs() function, providing information about the file system on which a file resides:

```
cpp_const('ST_MANDLOCK').
cpp_const('ST_NOATIME').
...

:- c_import("#include <sys/vfs.h>",
            [ libc ],
            [ statfs(string, -struct(statfs), [int])
            ]).
```

The `c_import/3` directive is compiled into Prolog statements that represent the involved types, requested macros (see section 2.4) and a *lazy* binding definition. The first call to `statfs/3` actually loads the library and creates the wrapper predicate. The type information is obtained by processing the first (string) argument using the C preprocessor and parsing the result into a Prolog AST. For this reason the library contains a full parser of the C99 standard including GCC extensions. Given the AST and the above declaration we can

- Find the prototype for statfs()
- Find all involved types by expanding the parameter types and return type until we reach to core C types.
- Find the constants represented by `ST_MANDLOCK`, etc. by adding variable declarations to the provided header and examining the AST that represents these variable declarations.
- The Prolog parameter declaration is verified to be consistent with the prototype and is used to guide the mapping from Prolog data to C and back.

As a result, we can verify the file system supports access time recording. To do this we first define `statfs/2`, dealing with the POSIX success/error conventions. The *FsStat* variable is bound to a *pointer wrapper* that maintains the type (`struct statfs`). The utility `c_load/2` fetches the `[f_flags]` field from the structure pointer. The `'ST_NOATIME'` is replaced by its numerical value based on *term expansion* in Prolog.

```
statfs(File, FsStat) :-
    statfs(File, FsStat, Status),
    posix_status(Status, statfs, file, File).

maintains_access_time(File) :-
    statfs(File, FsStat),
    c_load(FsStat[f_flags], Flags),
    Flags /\ 'ST_NOATIME' =:= 0.
```

1.1 Disadvantages

Above we created a binding that accesses a C library call and extracts information from the filled structure without writing any C. We did not need to worry about the structure layout, nor about the type of the `f_flags` field (`__fsword_t`). This seems to good to be true. What are the disadvantages? We give them below, split into new complexities and limitations. For each we hint at the (im)possibility for remedying.

1.1.1 New complexities and disadvantages

Finding a compatible C preprocessor and headers Our library uses library(process) to talk to the C preprocessor and assumes this is correctly configured to find the header files needed for the target libraries. This currently is configured for Linux and `gcc`. This should cover other popular options and a mechanism for the user to provide rules that match the target.

On Windows, MinGW can provide a compatible toolchain. On MacOS Xcode needs to be installed. Note that applications may provide precompiled `.qlf` files (see `qcompile/1`) which allows a user to access the foreign code without access to a C preprocessor and the header files.

Finding libraries The interface requires the real file that represents the shared object to be loaded. C toolchains come with a complicated OS and compiler dependent search strategy to find the concrete library file for the correct architecture and of the correct version. All we can probably do is to replicate some of this process for popular platforms, allow users to extend the rules and ultimately the user can specify the location as an absolute path.

Portability The portability is notably limited by the low-level library doing the dynamic calling of C functions. This library is in part written in assembler and requires details on the target C calling conventions, i.e., which parameters are placed where and where can the return value be found. The Prolog wrapper around that is only a new pages and thus easily replaced to use another low-level library.

Performance Implemented on top of the existing C-interface, this interface is by definition slower. In addition, Prolog wrappers may be generated to allocate memory for output arguments and the allocated memory is subject to malloc and the Prolog atom garbage collection while in several cases an automatic variable (allocated on the stack) suffices to hold the output. The automatically generated interface often does not provide a natural interface to the target resource, in which case a Prolog library

is required to provide the desired interface. Although this is typically far less work than doing this in C, the result generally performs less.

1.1.2 Limitations

Struct pointers, but no struct The underlying libffi library cannot create dynamic calls to functions that have a structure as argument, nor deal with functions that return a struct. For example we cannot wrap mallinfo() with the synopsis as below.

```
#include <malloc.h>

struct mallinfo mallinfo(void);
```

Accessing such a function is possible by defining a C source using the code below. This file can be compiled to a shared object and the source can be used by this library to access our wrapper library. The disadvantage of this approach is that it requires a fully operational C development toolchain, knowledge on how to operate it and the need to create and deploy the wrapper shared object. Unlike the native SWI-Prolog approach though our library is completely independent from SWI-Prolog and thus `SWI-Prolog.h` nor the library `libswipl.so` is needed.

```
#include <malloc.h>
void
pl_mallinfo(struct mallinfo *info)
{ *info = pl_mallinfo();
}
```

We assume that it is possible to extend the low-level assembly code used to build the dynamic calls to support structures if it is told how large the structures are.

Bitfields The current version does not support bitfields (`unsigned name : bits`) fields in structs.

This is not a fundamental restriction see also *Struct layout* below.

Struct layout The Prolog library rewrites a struct type into a sequence of fields where each field is either a basic scalar type, a struct, union or enum or an array of any of these. It first computes the layout, size and alignment of sub structs and unions. The layout of the struct is computed by placing the fields linearly in memory while adding padding fields required to satisfy the alignment restrictions (see `c_alignof/2`).

This is the default algorithm used by C compilers, but many compilers support additional attributes to control the layout. Such attributes are currently ignored. Accessing a struct with such attributes will lead to incorrect values. Additional rules can be added to resolve this.

Inline functions and macros Part of the C API of a library may look like functions but are in fact implemented as inline functions or macros. For example, the `glibc` (glibc is the standard C runtime library on Linux) function stat() to obtain information about an entry in the filesystem has the synopsis below. Binding this as stat() though results in an existence error.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
```

What is wrong? Going through the headers we find stat() as an inline function that calls __xstat() with an extra parameter that defines the version that should be `'_STAT_VER'`. We now have two options. We can use the same strategy as for mallinfo() above and define our own wrapper library that provides a stat() as a proper function or we can use the code below.

```
cpp_const('_STAT_VER').

:- c_import("#include <sys/types.h>
             #include <sys/stat.h>
             #include <unistd.h>",
            [ libc ],
            [ '__xstat'(int,string,-struct(stat),[int])
            ]).

stat(File, Stat) :-
    '__xstat'('_STAT_VER', File, Stat, Status),
    posix_status(Status, stat, file, File).
```

The advantage of the above is that no wrapper library is needed, but the disadvantage is that this only works for glibc. Other POSIX runtime libraries may provide stat() as a function or do something similar to, but incompatible with, glibc.

2 Making C functions available as predicates

The binding is realised using `c_import/3`

c_import(+*Header,* +*Flags,* +*Functions***)**

Import *Functions* from libraries provided by *Flags*. The *Header* is a string providing C source text that makes the types of the imported functions available. This text is handed to the configured C preprocessor. Normally, this string contains a series of `#include` ⟨*header*⟩ statements. For simple projects it is also possible to supply the actual C source file, e.g., `"#include \"my_c.c\""` (note the escaped double quotes).

Flags is a list of flags that both provide the optional C flags such as `-I` and the libraries where the functions can be found. Libraries may be written as e.g., `-lc`, `libc` or as a compound term for `absolute_file_name/3`. The library specification is resolved to the actual library file by `c_lib_path/2` from library(clocations). Finally, a term `pkg_config`(*Pkg,* ...) may be specified. This runs the `pkg-config` program using the provided options to obtain the required flags. The example below attaches the `uchardet` library.[1]

[1] A complete implementation is in the file `uchardet.pl` of the `examples` directory.

```
:- c_import("#include <uchardet.h>",
            [ pkg_config(uchardet, '--cflags', '--libs') ],
            ...).
```

Functions is a list of functions that must be made available as Prolog predicates. Each specification is a compound term whose name is the name of the C function to import. The arguments of the compound provide additional hints for this library for mapping the native C types to Prolog types. The compound has the same number of arguments as the C function or one more if the return value of the function is used. This last argument must be embedded in a list. The construct `[void]` may be used to forcefully ignore the return value of a function. Note that it is bad practice to ignore the return value if this may indicate an error condition.

Each argument is optionally wrapped in `+type` or `-type` to indicate an input or output argument. An output argument is realised by passing a pointer to an object of the required type and, upon completion of the function, reading the value from this pointer. Next, the argument may be wrapped in `*(type)` to indicate it is a pointer. The supported types are given below:

int
: The argument is to be mapped to a Prolog integer. This is compatible with C integers of all supported sizes. If the argument is an input argument a domain check is performed to validate that the (unbounded) Prolog integer can be represented by the bounded C integer. If the argument is an output argument the C type must be a pointer to a C integer. A buffer of sufficient size is allocated and after completion of the function the result is extracted from the pointer.

  The `int` value is also compatible with an `enum` typed C argument, returning the enum value as an integer. See also `enum` below.

float
: Supports C floats and doubles.

string
string(*Encoding*)
: As input argument, transform the Prolog data into a C string. *Encoding* defines the mapping from Prolog Unicode to the C representation. The default is `text`, using the default encoding of the OS. Other values are `iso_latin_1`, `utf8` and `wchar_t`. This Prolog type is compatible with the C type `char*`, except when *Encoding* is `wchar_t`. In this case the C type must be a pointer to an integer type of the same size as `wchar_t`.

atom
atom(*Encoding*)
: As `string` and `string`(*Encoding*) but when used for output arguments the variable is unified with an atom rather than a string.

struct(*Name*)
: Argument is a structure of the given name. If this is an input argument, a pointer to a structure must be supplied. If it is an output argument a pointer is allocated and returned.

union(*Name*)
: Argument is an union of the given name. Otherwise this is handled the same way as `struct`(*Name*).

enum(*Name*)
: Argument is an enum of the given name. An enum member is mapped to a Prolog atom. The wrapper translates between the atom and C integer that represent the enum encoding. Input and output arguments are handled the same way as `int`.

enum
: This can be used for enum arguments if the name of the enum is not important or unknown. Conversion to or from atoms is performed as with the `enum`(*Name*) specification. Note that an enum can also be specified as `int`, which stops converstion between atoms and the integer representation.

PredicateName(*Arg, ...*)
: Argument is a *closure*. The arguments of the function term are the predicate arguments for the predicate to which the closure is mapped and is subject to the same rules as the predicate arguments described in this list. See section 2.3

For *output* arguments (the function return or using a pointer argument that is filled by the function) we need to deal with *ownership transfer*. This is achieved using the construct `~`(*Type, FreeFunc*). See section 2.2 for details.

Finally, the following constructs are supported:

Function(*Arg*, **...**) **as** *PredicateName*
: Link the C function *Function* to the Prolog predicate *PredicateName* (an atom) rather than using the same name for the predicate and C function. This can be used to avoid conflicts with, e.g., built-in predicates or to allow for wrapper predicate that is named after the original C function.

[*Specification***]**
: If the specifiation is wrapped in a list, it is *optional*, i.e., if the target function cannot be found in the supplied headers it is silently ignored. This can be used together with `:- if(current_predicate(Name/Arity)).` to deal with portability issues.

The arguments to this directive are expanded using rules for `c_define/2`.

c_define(*+Term, -Replacement*)
: This *hook* is used for compiling `c_import/3`. As `c_import/3` is processed using `term_expansion/2` we cannot use rules. This is impractical as libraries and the called C functions and their arguments may be complex to write and version or platform dependent. This can be resolved by defining the predicate `c_define/2` and making it available in the context of the `c_import/3` directive. See `examples/python/python.pl` for examples.

Note that `c_define/2` has **no** relation to C `#define`. See also `cpp_macro/1`.

2.1 Calling variadic functions (f(x,y,...)

The current version has limited support for variadic functions, i.e., functions whose prototype argument list ends with `...`. The arguments before the `...` are processed normally. Remaining arguments must be declared and are trusted. This implies it is not yet possible to call such functions with arbitrary arguments, but it is possible to call them with a specific argument list. For example, we can print two floating point numbers using the code below. Note that we use the *... as name* construct which allows is to create multiple call patterns to printf()

```
:- use_module(library(ffi)).

:- c_import("#include <stdio.h>",
            [ '-lc' ],
            [ printf(string, double, double, [void]) as printf_dd
            ]).

test(X, Y) :-
    printf_dd("You entered %f and %f\n", X, Y).
```

Excess arguments (those matching the `...`) must use C types rather than Prolog types. The ffi library automatically applies the *default parameter type promotion rules* and thus, e.g., `short` is passed as `int`.

2.2 Data ownership considerations

An important aspect of C interfaces is dealing with *ownership* of data. Arithmetic values are simply copied, avoiding this issue. Arrays (including strings) and structures however are often dynamically allocated on the *heap* and must be released at some point. We distinguish three conventions:

1. The called function returns a pointer to a static memory area. In this case the receiver can read the returned data and must copy data it wishes to keep before calling the API again. This style lost popularity as it is not *thread-safe*.

2. Many modern APIs force the receiver to allocate memory for receiving the data. Especially for small amount of data (often structures) the (C) client can use an *automatic* variable to receive the data, e.g.,

   ```
   { struct stat buf;

     if ( stat(".", &buf) == 0 )
       ...
   }
   ```

   This style is problematic if the amount of data is unknown and therefore the buffer can be too small. In this case the API often returns incomplete data and an error indicating the data is incomplete and often the required size of the buffer.

3. Finally, the API may return a pointer to dynamically allocated memory. In this case the API also provides a function that allows the client to tell the API it is finished with the data. The release function may either decrement a *reference count* or immediately destroy the data. For example, most of the functions from the C library that use this style demand the caller to call free() when they no longer need the data.

The Prolog FFI can deal with all three styles. In the second scenario we cannot use automatic variables as the lifetime of the Prolog call is not immedately related to a C stack frame. Instead, we

use `c_alloc/2` to allocate space for the output. Memory allocated with `c_alloc/2` is subject to *atom garbage collection*. For the first and last option, the FFI library creates a pointer object just as `c_alloc/2`. As the called function created the object we cannot call free() on it.

The default is to do nothing. This is correct for the first style but causes memory leaks for the third style. In this case, the argument may be declared using `~`(*Type, FreeFunc*). Below are a couple of examples. The first is a trivial example based on the C standard library function strdup(). The other two examples are taken from the Python interface. Most of the Python interface functions return a pointer to a `PyObject` type where the receiver is required to call Py_DECREF() when done. PyDict_New() returns such an object and PyErr_Fetch() returns three objects using the output argument calling convention. Note that we call MyPy_DECREF() which is defined in the small support library because (1) Py_DECREF() is a macro and (2) Prolog will call the release function in the garbage collection thread and Python does not allow Py_DECREF() to be called asynchronously.

```
1.  strdup(string, [~(string, free)])
2.  'PyDict_New'([~(*'PyObject', 'MyPy_DECREF')])
3.  'PyErr_Fetch'(-(~(*'PyObject', 'MyPy_DECREF')),
                  -(~(*'PyObject', 'MyPy_DECREF')),
                  -(~(*'PyObject', 'MyPy_DECREF')))
```

2.3 Handling closures

The term *closure* refers to a pointer to a C function that is passed to a C function or placed in a structure. The registered function is called on certain events. An example is qsort() from the standard C library that accepts a closure to compare two elements of the array that is to be sorted. In this section we explain how qsort() can be used through this library.[2] Below we declare the function.

```
:- use_module(library(ffi)).
:- c_import("#include <stdlib.h>",
            [ libc ],
            [ qsort(*void, int, int, qcompare(*void, *void, [int]))
            ]).
```

The declaration above creates a predicate `qsort/3` that performs a callback on the predicate `qcompare/3`. Note that, unlike in C, we provide a predefined predicate that is called. This is done such that the library can create a closure object that matches the requested function type. If the called predicate needs to be dynamically determined there are two options:

1. Use "*Function*(*Arg*, ...) as *PredicateName*" to bind different variations to different predicates.
2. Call the specific closure predicate from the generic one after making it available as a hook definition or global variable.

In our example we create `qcompare/3` as a function that operates on an integer array. The example uses `c_load/4` to load an `int` indirectly over the `void*` argument. Prolog `compare/3` is used to compare the elements and the result is translated to an `int` to comply with the qsort() calling convention.

[2] The qsort() function is used for educational purposes and because it is widely available in C runtime libraries.

```
qcompare(Ptr1, Ptr2, Diff) :-
    c_load(Ptr1, 0, int, I1),
    c_load(Ptr2, 0, int, I2),
    compare(DiffA, I1, I2),
    diff_int(DiffA, Diff).

diff_int(<,-1).
diff_int(=, 0).
diff_int(>, 1).
```

Given this, we can sort an array of integers:

```
sort_int_array(Ptr, Length) :-
    c_sizeof(int, ESize),
    qsort(Ptr, Length, ESize).
```

A complete example to create an array of C `int` from a Prolog list, sort it and translate the result back to Prolog is below.

```
sort_int_list(List, Sorted) :-
    length(List, Count),
    c_alloc(Data, int[Count]),
    fill_array(List, 0, Data),
    c_sizeof(int, ESize),
    qsort(Data, Count, ESize),
    read_array(0, Count, Data, Sorted).

fill_array([], _, _).
fill_array([H|T], I, Data) :-
    c_store(Data[I], H),
    I2 is I + 1,
    fill_array(T, I2, Data).

read_array(I, Count, Data, [H|T]) :-
    I < Count,
    !,
    c_load(Data[I], H),
    I2 is I + 1,
    read_array(I2, Count, Data, T).
read_array(_, _, _, []).
```

The predicate `c_store/2` provides support for assigning a struct field with a closure that calls a Prolog predicate.

2.4 Accessing preprocessor constants

C libraries tend to make constants available using C preprocessor macros. In many cases, these are combined using *bitwise or* to form *flags*. The example below is a commonly seen snippet in C.

```
int fd = open("myfile", O_CREATE|O_WRONLY);
```

Constants like `O_CREATE` are made available from Prolog by defining the predicate `cpp_const/1` before using the `c_import/3` directive. This cause `c_import/3` to add code to the header to initialize global variables with the macro value. After parsing the AST fragment that initializes the constant is evaluated to an integer, float or string literal.

cpp_const(-*Name*)
: is nondet. This hook predicate can be defined in the same module as from where the `c_import/3` directive is used. It causes `c_import/3` to create clauses `cpp_const/2`, `cpp_const`(*Name, Value*). These clauses are used by a rule for system:term_expansion/2 if library(ffi) is imported into the current context. This causes the following terms to be expanded:

  Atom
  : An atom that matches a defined `cpp_const/2` fact is expanded into the related constant.

  'C'(:*Expression*)
  : Expand *Expression* and evaluate it. If the expression is module qualified add the indicated module to the modules in which rules for `cpp_const/2` are searched. The evaluation of *Expression* recognises the C operators `|`, `~` and `&`. The remainder is handed to Prolog `is/2`. For example, we can write the following

```
cpp_const('O_CREATE').
cpp_const('O_WRONLY').

:- c_import("#include <sys/types.h>
             #include <sys/stat.h>
             #include <fcntl.h>
             #include <unistd.h>",
            [ '-lc' ],
            [ open(string, int, [int]) as c_open
            ]).

test(File) :-
    c_open(File, 'C'('O_CREATE'|'O_WRONLY'), Fd),
    posix_status(Fd, open, file, File),
    ....
```

2.5 Module awareness

User defined types (structures, unions and enums) are compiled into Prolog predicates. These definitions are local to the module that uses the `c_import/3` statement.

3 Accessing C data

Interaction with native C functions requires the ability to work with C data structures from Prolog. Arithmetic types (various sizes of integers and floating point numbers) are simple as such parameters are easily converted from Prolog numbers and the return value is easily converted back. Arithmetic types have no interesting internal structure and need no memory management such as `malloc()` and `free()`.

Arrays and structures however do have internal structure and typically do need memory management. Managing such data is achieved in two layers. The high level layer reasons in terms of abstract types, while the low level layer deals with pointers and access to primitive C (scalar) data types.

3.1 Pointers

The core of the memory access functions is formed by a SWI-Prolog *blob* of type `c_ptr`. Such a blob wraps a C pointer. It has the following properties:

- The **type** is an atom or a term `struct(Name)`, `union(Name)` or `enum(Name)` that represents the C type of an element.
- The **size** is an integer representing the size of an element in bytes.
- The **count** represents the number of elements. It is `-1` if this is not know.
- An **indirection level**. If 0 (zero), it is a pointer to an (array of) `object(s)` of the indicated type. If 1 it is a pointer to a pointer of objects of the indicated type, etc.
- An optionally associated *free* function is called if the blob is garbage collected by the atom garbage collector.

Pointer blobs are created using predicates `c_alloc/2`, `c_cast/3` and `c_load/2` if the addressed object is not a scalar type. Pointer blobs are also created by C functions if the return value is a pointer or an argument is declared as an *output* argument returning a pointer. A pointer allocated with `c_alloc/2` *owns* the pointer, reclaiming the associated memory as the blob is garbage collected. Function return and output pointers may be declared to *own* the pointe using the `~(Type,Free)` type declaration. See `c_import/3`.

Pointer blobs are subject to (atom) garbage collection. Atom reference counts are used to avoid collecting of pointers that depend on other pointers. Notable `c_load/2` references the original pointer if it returns a pointer inside the area of the original pointer and `c_store/2` references the *Value* if the *Value* is a pointer.

3.2 Types

A type is either a primitive type or a constructed type.

3.2.1 Basic types

The following basic types are identified:

Signed integers `char`, `short`, `int`, `long` and `longlong`

Unsigned integers `uchar`, `ushort`, `uint`, `ulong` and `ulonglong`

Floats `float` and `double`

Pointers `*(Type)`, `closure` (pointer to a function)

In addition, the type `wchar_t` is recognised by the library to facilitate portable exchange of Unicode text represented as wide character strings.

3.2.2 Constructed types

The constructed types are `struct(Name)`, `union(Name)` and `enum(Name)`. The `c_import/3` directive extracts types that are (transitively) reachable from imported functions to the current module. In addition, types can be defined using `c_struct/1` and `c_union/1`. Such declarations can be used to create and access C binary data without using library functions.

c_struct(+*Name, +Fields*)
Declare a C structure with name *Name*. *Fields* is a list of field specifications of the form:

- `f(Name, Type)`

Where Type is one of

- A primitive type (`char`, `uchar`, ...)
- `struct(Name)`
- `union(Name)`
- `enum(Name)`
- *(Type)
- `array(Type, Size)`

This directive is normally used by `c_import/3` to create type information for structures that are involved in functions that are imported. This directive may be used explicitly in combination with the C memory access predicates to read or write memory using C binary representation.

c_union(+*Name, +Fields*)
Declare a C union with name *Name*. *Fields* is a list of fields using the same conventions as `c_struct/2`.

The defined types may be examined using the following interface:

c_current_enum(*?Name, :Enum, ?Int*)
True when Id is a member of *Enum* with Value.

c_current_struct(*:Name*) *[nondet]*
c_current_struct(*:Name, ?Size, ?Align*) *[nondet]*
Total size of the struct in bytes and alignment restrictions.

c_current_struct(*:Name*) *[nondet]*
c_current_struct(*:Name, ?Size, ?Align*) *[nondet]*
Total size of the struct in bytes and alignment restrictions.

c_current_struct_field(*:Name, ?Field, ?Offset, ?Type*)
Fact to provide efficient access to fields

c_current_union(*:Name*) *[nondet]*
c_current_union(*:Name, ?Size, ?Align*) *[nondet]*
Total size of the union in bytes and alignment restrictions.

c_current_union(*:Name*) *[nondet]*
c_current_union(*:Name, ?Size, ?Align*) *[nondet]*
Total size of the union in bytes and alignment restrictions.

c_current_union_field(*:Name, ?Field, ?Type*)
Fact to provide efficient access to fields

c_current_typedef(*:Name, :Type*) *[nondet]*
True when *Name* is a typedef name for *Type*.

c_expand_type(*:TypeIn, :TypeOut*)
Expand user defined types to arrive at the core type.

c_type_size_align(*:Type, -Size, -Alignment*) *[det]*
True when *Type* must be aligned at *Alignment* and is of size *Size*.

3.2.3 The high level interface

c_alloc(*-Ptr, :TypeAndInit*) *[det]*
Allocate memory for a C object of Type and optionally initialse the data. *TypeAndInit* can take several forms:

A plain type Allocate an array to hold a single object of the given type.

Type[Count] Allocate an array to hold *Count* objects of *Type*.

Type[] = Init If Init is data that can be used to initialize an array of objects of Type, allocate an array of sufficient size and initialize each element with data from Init. The following combinations of Type and Init are supported:

char[] = Text Where Text is a valid Prolog representation for text: an atom, string, list of character codes or list of characters. The Prolog Unicode data is encoded using the native multibyte encoding of the OS.

`char(Encoding)` **[] = Text** Same as above, using a specific encoding. Encoding is one of `text` (as above), `utf8` or `iso_latin_1`.

Type[] = List If Data is a list, allocate an array of the length of the list and store each element in the corresponding location of the array.

Type = Value Same as `Type[] = [Value]`.

To be done
- : error generation
- : support enum and struct initialization from atoms and dicts.

c_cast(*:Type, +PtrIn, -PtrOut*)
Cast a pointer. *Type* is one of:

address
Unify *PtrOut* with an integer that reflects the address of the pointer.

(*[Count], Type***)**
Create a pointer to *Count* elements of *Type*.

Type
Create a pointer to an unknown number of elements of *Type*.

c_load(*:Location, -Value***)** *[det]*
Load a C value indirect from *Location*. *Location* is a pointer, postfixed with zero or more one-element lists. Like JavaScript, the array postfix notation is used to access array elements as well as struct or union fields. *Value* depends on the type of the addressed location:

Type	**Prolog value**
scalar	number
struct	pointer
union	pointer
enum	atom
pointer	pointer

c_store(*:Location, +Value***)**
Store a C value indirect at *Location*. See `c_load/2` for the location syntax. In addition to the conversions provided by `c_load/2`, `c_store/2` supports setting a struct field to a *closure*. Consider the following declaration:

```
struct demo_func
{ int (*mul_i)(int, int);
};
```

We can initialise an instance of this structure holding a C function pointer that calls the predicate `mymul/3` as follows:

```
    c_alloc(Ptr, struct(demo_func)),
    c_store(Ptr[mul_i], mymul(int, int, [int])),
```

c_nil(*-Ptr***)** *[det]*
Unify *Ptr* with a (void) NULL pointer.

c_is_nil(*@Ptr***)** *[semidet]*
True when *Ptr* is a pointer object representing a NULL pointer.

3.2.4 The low level interface

The low-level interface is build around a SWI-Prolog *blob* that represents a C pointer with some metadata. A *blob* is similar to a Prolog atom, but blobs are typed and they are intended to deal with binary data.

c_calloc(*-Ptr, +Type, +Size, +Count*) *[det]*

Allocate a chunk of memory similar to the C `calloc()` function. The chunk is associated with the created *Ptr*, a *blob* of type `c_ptr` (see `blob/2`). The content of the chunk is filled with 0-bytes. If the blob is garbage collected by the atom garbage collector the allocated chunk is freed.

Arguments

Type	is the represented C type. It is either an atom or a term of the shape `struct(Name)`, `union(Name)` or `enum(Name)`. The atomic type name is not interpreted. See also `c_typeof/2`.
Size	is the size of a single element in bytes, i.e., should be set to `sizeof(Type)`. As this low level function doesn't know how large a structure or union is, this figure must be supplied by the high level predicates.
Count	is the number of elements in the array.

c_free(*+Ptr*) *[det]*

Free the chunk associated with *Ptr* by calling the registered release function immediately. This may be used to reduce the memory foodprint without waiting for the atom garbage collector. The blob itself can only be reclaimed by the atom garbage collector.

The type release function is non-NULL if the block as allocated using `c_alloc/2` or a function was associated with a pointer created from an *output* argument or the foreign function return value using the `~(Type, Free)` mechanism.

c_disown(*+Ptr*) *[det]*

Clear the *release function* associated with the blob. This implies that the block associated with the pointer is not released when the blob is garbage collected. This can be used to transfer ownership of a memory blob allocated using `c_alloc/2` to the foreign application. The foreign application must call PL_free() from the SWI-Prolog API to release the memory. On systems where the heap is not associated with a foreign module, the C library `free()` function may be used as well. Using `free()` works on all Unix systems we are aware of, but does **not work on Windows**.

c_alloc_string(*-Ptr, +Data, +Encoding*) *[det]*

Create a C `char` or `wchar_t` string from Prolog text *Data*. *Data* is an atom, string, code list, char list or integer. The text is encoded according to *Encoding*, which is one of `iso_latin_1`, `utf8`, `octet`, `text` or `wchar_t`. The encodings `octet` and `iso_latin_1` are synonym. The conversion may raise a `representation_error` exception if the encoding cannot represent all code points in *Data*. The resulting string or wide string is nul-terminated. Note that *Data* may contain code point 0 (zero). The length of the string can be accessed using `c_dim/3`. The reported length includes the terminating nul code.

This predicate is normally accessed through the high level interface provided by `c_alloc/2`.

c_load(*+Ptr, +Offset, +Type, -Value*) *[det]*

Fetch a C arithmetic value of *Type* at *Offset* from the pointer. *Value* is unified with an integer or floating point number. If the size of the chunk behind the pointer is known, *Offset* is validated to be inside the chunk represented by *Ptr*. Pointers may

c_load_string(+*Ptr, -Data, +As, +Encoding*) *[det]*

c_load_string(+*Ptr, +Length, -Data, +As, +Encoding*) *[det]*

Assuming *Ptr* points at text, either `char` or `wchar_t`, extract the value to Prolog. The `c_load_string/4` variant assumes the text is nul-terminated.

Arguments

As	defines the resulting Prolog type and is one of `atom`, `string`, `codes` or `chars`
Encoding	is one of `iso_latin_1`, `octet`, `utf8`, `text` or `wchar_t`.

c_load_string(+*Ptr, -Data, +As, +Encoding*) *[det]*

c_load_string(+*Ptr, +Length, -Data, +As, +Encoding*) *[det]*

Assuming *Ptr* points at text, either `char` or `wchar_t`, extract the value to Prolog. The `c_load_string/4` variant assumes the text is nul-terminated.

Arguments

As	defines the resulting Prolog type and is one of `atom`, `string`, `codes` or `chars`
Encoding	is one of `iso_latin_1`, `octet`, `utf8`, `text` or `wchar_t`.

c_store(+*Ptr, +Offset, +Type, +Value*) *[det]*

Store a C scalar value of type *Type* at *Offset* into *Ptr*. If *Value* is a pointer, its reference count is incremented to ensure it is not garbage collected before *Ptr* is garbage collected.

c_offset(+*Ptr0, +Offset, +Type, +Size, +Count, -Ptr*) *[det]*

Get a pointer to some location inside the chunk *Ptr0*. This is currently used to get a stand-alone pointer to a struct embedded in another struct or a struct from an array of structs. Note that this is **not** for accessing pointers inside a struct.

Creating a pointer inside an existing chunk increments the reference count of *Ptr0*. Reclaiming the two pointers requires two atom garbage collection cycles, one to reclaim the sub-pointer *Ptr* and one to reclaim *Ptr0*.

The `c_offset/5` primitive can also be used to *cast* a pointer, i.e., reinterpret its contents as if the pointer points at data of a different type.

c_typeof(+*Ptr, -Type*) *[det]*

True when *Type* is the *Type* used to create *Ptr* using `c_calloc/4` or `c_offset/6`.

Arguments

Type	is an atom or term of the shape `struct(Name)`, `union(Name)` or `enum(Name)`. *Type* may be mapped in zero or more *(*Type*) terms, representing the levels of pointer indirection.

c_sizeof(+*Type, -Bytes*) *[semidet]*

True when *Bytes* is the size of the C scalar type *Type*. Only supports basic C types. Fails silently on user defined types.

c_alignof(+*Type, -Bytes*) *[semidet]*

True when *Bytes* is the mininal alignment for the C scalar type *Type*. Only supports basic C types. Fails silently on user defined types. This value is used to compute the layout of structs.

c_address(+*Ptr, -Address*) *[det]*

True when *Address* is the (signed) integer address pointed at by *Ptr*.

c_dim(+*Ptr, -Count, -ElemSize*) *[det]*

True when *Ptr* holds *Count* elements of size *ElemSize*. Both *Count* and *ElemSize* are 0 (zero) if the value is not known.

4 library(clocations): Define resource locations for the ffi library

This module provides the mapping from library names to concrete files that can be loaded. This is used by `c_import/3`. While C compilers typically allow one to specify a library as, e.g., `-lm`, the actual naming and physical location of the file providing this library is compiler and system dependent.

This module defines `c_lib_path/2` to find the concrete file implementing a C library. Hooks may be used to extend this predicate:

- `library_path_hook/2` is called first by `c_lib_path/2` and may redefine the entire process.
- `cpu_alias/2` may be used if `ldconfig -p` is used to verify that a library is compatible with the current architecture of the SWI-Prolog process.

cpp(-*Command, -Argv*) *[det]*

Provide the *Command* and *Argv* for `process_create/3` to call the C proprocessor reading the C input from standard input.

c_lib_path(+*Spec, -Path, +Options*) *[det]*

Find a shared object from *Spec*. *Spec* is one of:

Concrete file If spec is an atom or string denoting an existing file, this is used.

Alias(File) Handled to `absolute_file_name/3` using the options `access(execute)` and `extensions(['',Ext])`, where *Ext* is the value for the Prolog flag `shared_object_extension`

Plain atom Platform dependent search. Currently implemented for

- Systems that support `ldconfig -p` (e.g., Linux)

Additional search strategies may be realised by defining rules for `library_path_hook/2` with the same signature.

To be done Extend the platform specific search strategies.

ffi:library_path_hook(+*Name, -Path, +Options*) *[semidet,multifile]*

Multifile hook that can be defined to resolve a library to a concrete file. The hook is tried as first option by `c_lib_path/2`.

ldconfig(*?Name, ?Path, ?Version, ?Flags*) *[nondet]*

True when *Name* is the base name of a library in the ldconfig cache.

Arguments	
Name	is the base name of the library, without version or extension.
Path	is the absolute file name of the library
Version	is the version extension as an atom
Flags	is a list of atoms with flags about the library

ldconfig_flush
Flush the library cache maintained for ldconfig.

5 library(cerror): C interface error handling

This module provides common routines to map error codes into apropriate actions in Prolog. Below is a typical example mapping the `statfs()` function:

```
:- module(libc_files,
          [ statfs/2
          ]).
:- use_module(library(cinvoke)).

:- c_import("#include <sys/vfs.h>",
            [ libc ],
            [ statfs(+string, -struct(statfs), [-int])
            ]).

statfs(File, FsStat) :-
    statfs(File, FsStat, Status),
    posix_status(Status, statfs, file, File).
```

posix_status(+*Code*) *[det]*
posix_status(+*Code, +Action, +Type, +Argument*) *[det]*
These predicates may be used to map POSIX `int` error return status into a suitable Prolog response. If *Code* is non-negative the predicate simply succeeds. For other cases it retrieves the error code using `c_errno/1` and translates the error into a suitable Prolog exception.

posix_ptr_status(+*Code*) *[det]*
posix_ptr_status(+*Code, +Action, +Type, +Argument*) *[det]*
Handle the return code from POSIX functions that return a NULL pointer on error.

posix_raise_error *[det]*
posix_raise_error(+*Action, +Type, +Argument*) *[det]*
Raise an error from a POSIX `errno` code.

Errors `posix_error(Errno, String)`

6 Portability and platform notes

This interface requires a C preprocessor and C header files that are compatible with the libraries that must be accessed. So far we used `gcc` and the closely compatible `clang` and `MinGW` compilers.

To make ffi work with other compilers the predicate `cpp/2` defined in `clocations.pl` must be extended or hooked using ffi:cpp_hook/2 to tell this library how to call the C preprocessor such that it reads from standard input and writes to standard input. If this is not possible `cdecls.c` must be extended to deal with proprocessors that cannot read from standard input.

C compilers often come with language extensions that are notably used in the standard headers which are needed to access system libraries and are often included from headers shipped with many projects. These may result in warnings from our C parser. The way to diagnose and fix such issues is to first collect the C output by enabling a debug channal and reloading your Prolog file:

```
?- debug(ffi(dump(cpp_output, 'myfile.h'))).
?- [myfile].
```

Now, examine `myfile.h` and find the declarations that cannot be parsed. Next, run the code below, replacing the argument of `ast/1` with a string containing the text that could not be parsed, try to find out why it fails and extend the tokenizer from `ctokens.pl` and/or the parser in `cparser.pl`,

```
$ swipl test/debug_types.pl
?- ast("failed declaration").
```

Finding resources such as libraries is implemented by `clocations.pl`. This file must be extended to improve support on platforms.

6.1 Windows

We have used this interface successfully with MinGW. Make sure that the directory holding `gcc` is in `%PATH%`.