Pharo Uniform Foreign Function Interface Architecture Design - Fourth Draft

Ronie Salgado

1 Introduction

Pharo is an environment that cannot be used in isolation, unless you can also write a full operating system in itself. Most of the modern and commonly operating systems are designed to be used by, programs that communicate with them using core libraries written in C/C++/Objective-C. In addition to that, there are lots of libraries written in C/C++ or other languages that can communicate with C.

Because of these reasons, Pharo has to have a proper way to communicate with the outside world, in other words, a good Foreign Function Interface or FFI. A FFI is an interface to be able to call methods/procedures written in one programming language from another language, but it also can allow to receive messages from the foreign language.

Typical use cases for an FFI is being able to interface with massive libraries and frameworks that is not worth to rewrite, such as the operating system, GUI toolkits such as Gtk+ and Qt, graphic image manipulation libraries, physics simulation engines, audio libraries.

Other typical uses are to write some performance critical portions of an application in a low-level language such as C/C++, but using high-level language as Smalltalk for the parts whose performance is not so important, for example a GUI. A typical example for this use case are video games, where all the graphics, physics and networking is done in C++, but all the game logic in a scripting language.

Currently in Pharo there are three FFIs: the old squeak FFI, the Alien and the NativeBoost FFI. Of those three FFI, in the latest version of Pharo only the NativeBoost FFI is supported, and is the fastest of the three of them. But they all have problems with either portability, callbacks or speed.

- The old Squeak FFI has problems with portability, callbacks and speed.
- Alien has problems with portability and speed.
- NativeBoost FFI is really fast, but it has problems with portability and it does not provide end user support for callbacks. With NativeBoost callbacks can be done, by writing their entry point and parameter mashalling manually, in platform specific assembly language.

Because of those problems, this document proposes the creation of a modular and pluggable unified foreign function interface or UFFI, that can have multiple back-ends, among them the current FFIs or something completely different.

1.1 A glance in NativeBoost FFI

The problems of the current NativeBoost FFI is that is not portable, has bad end user support for callbacks, has problems for making indirect calls, his documentation is not available in a centralized place for easy reference. But, despite of those problems, you can do anything with NativeBoost, because it gives you a powerful API into a dynamic recompiler, with an DSL for x86 assembler. In addition to that, the user interface of performing a NativeBoost call is really well done and the concept it uses can give you a really fast marshalling.

The design idea of NativeBoost consists in, using a primitive which takes the control of a Smalltalk method, check if a native function that knows that it has access to the VM and knows about marshalling was created and installed. If that native function exists, the primitive gives control to it by getting his offset in memory, casting it into a C function pointer with a known signature and then calling it.

In addition to the native function code, there is also meta-data for checking for the platform version of the code, and some house keeping.

Because of the simplicity of the approach taken by NativeBoost and his generality, one decision taken for making a new FFI system consists in taking from the NativeBoost FFI as most as possible for making the end-user interface. If an UFFI back-end does not generate code, it could install a function pointer with a well known signature into a static function that performs the actually call-out.

2 Architecture Design

There are three fundamental modules in a foreign function interface:

- Foreign Resource Management
- Foreign Function Calling
- Foreign Function Callback

2.1 Foreign Resource Management

Foreign Resource Management (FRM) concerns about managing the resources that has to be understood by both, foreign language methods and language local methods. This module should care about memory management, structures layout, reading and writing values in memory.

This resource management module should care about giving a full interface into the C memory in a portable way, and in his internal implementation it

only can relay in some kind of description of the underlying platforms, that can include things such as alignment, stack alignment.

Some types are really well defined such as sized integer as uint8, uint16, etc or floating point number that follows the IEEE-754 standard. Those resources must be handled in a portable way and this library should take into account the endianness of the machine only when needed. If the machine endianness cannot be detected and its needed, a Smalltalk exception must be raised.

For architecture and operating system dependents C types such as short, int, long, etc, they should be a description of each platform and operating system class that provides a mapping between the non-portable C type and the portable type.

The size of a pointer seems to be a special case. They have a well defined user interface and semantics associated with it. For purposes of layouts, they must be mapped into one of the fixed size integral types, unless there are some special requirements for alignment.

The fixed size primitive types won't be having a default alignment, because some of these types could not natively be supported by the underlying platform, and they could not have in these case the expected alignment, that is his own size.

2.2 Foreign Function Calling

The current NativeBoost callout interface is like this:

```
function....
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin error: errorCode >
    ^ self nbCall: #( retType functionName ( args ...) ) module: 'moduleName'
```

For the UFFI, the following callout is proposed:

```
function....
    <native>
    <primitive: #primitiveNativeCall module: #UFFIPlugin error: errorCode >
    ^ self ffiCall: #( retType functionName ( args ...) ) module: 'moduleName'
```

The native pragma is to give a potential opportunity for a fast as possible implementation, that avoids completely hitting a primitive trap, either at the VM level or the compiler level, but his actual support and specification must be optional. In a ideal world it would be preferred to only have that pragma because is easier to write and remember, but it would require patching at least the compiler, by turning the pragma into a *primitiveNativeCall*. Because patching the compiler can be a trouble and is preferred to be avoided, there is an initial plan for using both pragmas as is shown.

The primitive implementation mirrors the native boost one, but it also has to perform checks for additional ways of doing the actual call. It can have an option to dispatch the primitive into another VM plugin primitive handler.

When the native pragma is not supported by the compiler and VM, the primitive is going to be fired, which could mean some additional calls and context changing from highly optimized JITed code into a C procedure that takes care of handling the primitive.

The ffiCall also has to perform additional checks than nbCall. It has to look for a suitable ffi backend, that takes care of installing the callout data required for corresponding VM plugin, if there is someone.

Some UFFI backend would like to take care of the whole process from this point, such as the NativeBoost back end. Other backends may like to have the function signature already parsed and some type mapping already done. The backend plugin interface has to allow both options.

2.3 Foreign Function Callback

For receiving a callback, the Smalltalk entry point must be a class side method and is going to look like this:

```
callbackMethod_arg1: arg1 arg2: arg2 ...
    <callback: #( retType (CTypeoOf<arg1> arg1, CTypeOf<arg2>, ... )>

    " Here goes normal Smalltalk code "
    ...
```

This come from an important observation, callbacks do not come from nowhere, they come from a library that has received a C function pointer some time ago. For getting the C trampoline that does all the marshalling, the user is going to use something like this:

```
someObject registerSomeCallback: (self class >> #theCallback) ffiCallback
```

The ffiCallback message takes care about, parsing the callback pragma and sending into the FFI backends the data required to synthesize the C function trampoline into the Smalltalk code, that also performs all the marshalling needed. If the FFI backend cannot create the callback trampoline, it has to raise an exception.

3 Potential Implementation Details

3.1 C Function Call Taxonomy

The end user of a FFI only cares about calling a C function, by using a direct call to the function using his name or an indirect call by using a function pointer. By inspecting in how those calls are usually done, they can be classified into the following four cases:

- Calling a C function whose entry point is bound into a global symbol that can be looked by using a standard operating system.

- Calling a C function whose entry point was manually stored into a c function pointer, but whose value is not going change. This happens when the user needs to performs a manual lookup of the function entry point,
- Calling a C function whose entry point is stored in a table of function pointer, where the table itself can change but the index of the elements cannot. These case happens when doing some kind object-oriented programming in C.
- Calling a C function whose entry point is stored in a function pointer, whose value can change or the pointer itself changes. This is usually done for registering some callbacks.

Each one of these cases can be handled in a different way when the objective is performance. But, from those cases, the most important are performing a call into a global C function, function pointer whose value does not change and a function pointer that can change. The table lookup is just a subset when doing a call into a function pointer that can change. The exact syntax is going to be documented when it is done, but it has to follow the same spirit of the direct call syntax exposed in section about Foreign Function Calling.

4 Additional Backends

4.1 Alien

One of the objectives of making the UFFI is to be able to reuse the existing Alien FFI, for architectures in which adding a port for it is easier than making NativeBoost more portable.

4.2 LLVM

One interesting idea is to make a backend that uses LLVM compiler infrastructure, because it is designed for compiling optimized C function and it already has multiples JIT backends. This small project is not to make a full Pharo VM JIT based in LLVM, because LLVM is not designed for that and it would require a lot of patching of LLVM.

4.3 Static C Function Generator

Another potential backend, is a fake FFI. This backend parses all the FFI callout and callbacks in a preprocessing step, and generate for each one of them C code that can be statically linked to the VM, so it does not have to perform dynamic code generation, which is forbidden in some platforms and at the same it can give the same performance as NativeBoost, but in the cost of flexibility and losing the ability of modifying the uses of FFI in a running production system.

5 Implementations Plans

5.1 General Plans

They are three main modules to the UFFI architecture design, whose exact implementation are going to be done using the following criterias and steps:

- Attempt to produce a minimal overhead when using NativeBoost as a backend. This also includes needed refactoring of NativeBoost when needed.
- Use a TDD for defining external interfaces and internal implementations.
- Document using class comments and method comments during the early stages of implementation.
- Create a tutorial and a reference manual for UFFI in a place that can be find using a simple web search.
- If it is possible, a formal specification of UFFI that can be implemented in other Smalltalk dialects should be written, after having a full implementation in Pharo.

The special emphasis in documentation, is given because of the lack of an easy to find and consult centralized documentation, that it is required to allow the entrance of new people working in core features and also for users, that needs a simple step-by-step tutorial for getting started, and a full reference manual when they need more advanced features.

5.2 Foreign Resource Management

The plans for implementing the FRM consists in:

- Study NativeBoost way of doing things here.
- Start adding a C type mapping and specification, making a clear distinction between type specification and actual type marshalling.
- Refactor NativeBoost to start using the UFFI interface for resource management. Because NativeBoost installs an optimized method that does marshalling, the extra overhead for portability should be present only when compiling the native callout.

5.3 Foreign Function Calling

The objectives here consists in adding the uniform foreign call specification, which should do an automatic lookup of the actual backend implementation. This also include extracting some elements from NativeBoost, such as the function call specification parser.

For using the NativeBoost backend, this part should be really simple. Others backends are a lot more complex, because there is a pending documenting task for the VM interface and marshalling details.

5.4 Foreign Function Callback

Callbacks are not supported without using assembly code in NativeBoost. Because of that, the callbacks support has to be added from scratch. This is the last part that is going to be implemented in UFFI, and his implementation can be tricky.