The Squeak Foreign Function Interface and its implementation demystified

By Nicolas Cellier

This is an attempt to document the Squeak FFI both from the point of view of usage, and the point of view of implementation. It is thus a reference document rather than just a user manual with recipes for solving most common problems.

This document will identify some of the limitations, and propose some improvements. It is thus not a definitive and ultimate reference, but rather a snapshot of the Squeak FFI at this time of writing. It should be a guide for making the Squeak FFI evolve toward a more powerful, well understood and safe implementation.

Note: this document applies to the modernized FFI plugin a.k.a. the ThreadedFFIPlugin.

About the author

I'm an engineer working in the domain of signal and control theory. I have designed or participated to the design of autopilots for all sort of vehicles on the surface of water, underwater, in the air or in space. I have also always worked on the design of tools dedicated to such activity. That's where Smalltalk come into play.

I've been intrigued by Smalltalk language after reading a byte magazine at the end of my student cursus in 1987, and had the chance to start using Smalltalk for my first job in 1988, with Smalltalk-V then with Parcplace Smalltalk-80 V2, Objectworks then Visualworks. With this active technology intelligence, and the opportunities given by french company Aerospatiale (one main component of today's EADS), two engineers could prototype in one year what would become the main tool for control design in Aerospatiale-EADS for the next 15 years. Of course, more work was necessary to transform the prototype into a product, and even more to continuously improve the product. But that remarkable achievement is due in large part to the efficiency of Smalltalk for rapid design.

I have joined the naval industry at end of 2003, and since, I rarely use Smalltalk professionally. But I have found the way to continue Smalltalking by regularly contributing to the Squeak project, both by improving the libraries, and the virtual machine.

During these years of engineering, it's always been vital to leverage the existing and standard libraries. An example of such lever is Smallapack, a library for interfacing Smalltalk to Lapack – the famous package for matrix algebra. Smallapack is a rewrite of some of the tools I had developed for Aerospatiale. A more recent example is an attempt to interface HDF5, a format and a library often used in aerospace industry for exchanging large structured data. This bring us back to FFI.

I have adopted and strongly support Squeak, but it's not always been easy to use Squeak FFI, because there are a few dark corners and not much documentation available. This bring us back to this document, sorry for the long digressions.

An overview of Squeak FFI

A Smalltalk world is composed of objects achieving some task by sending messages to other objects. In response to those messages, objects execute generally small list of instructions

gathered in a method. But what if we want to execute instructions which are already written in a foreign language? Squeak provides two main alternatives:

- Add an extension to the virtual machine that wrap this function call into a new primitive. Unfortunately, those primitives are kind of static: we have to stop Squeak, compile the plugin, restart Squeak, add the code for invoking the primitive, and eventually retry. Not exactly the lively experience of Smalltalk.
- Dynamically link an external library and call a function in that library directly from within Squeak, without interrupting. This sounds more appealing: that's FFI.

As introduction, here is a quick overview of the main class composing Squeak FFI. Since we want to call an external function that resides in an external library, it's not unsurprising to find an ExternalLibrary and ExternalLibraryFunction classes reflecting those two fundamental entities.

An ExternalLibrary is a proxy to an entity known to the OS (identified via a handle). Its main role is to dynamically load the library (typically via dlopen() or LoadLibraryEx() function calls), and to find external functions by their name or number, and return an appropriate handle known to the OS for calling the function.

An ExternalLibraryFunction holds the handle known to the OS for calling the function, and also hold the specification of the function interface:

- What calling conventions are used
- What is the return type of the function
- What parameters the function take, and which is the type of those parameters

It is neither a surprise to find an ExternalType class for reflecting those types.

Once the interface established, the purpose of FFi is to exchange values with the external function, passed via those parameters or returned back by the function. But in Smalltalk, we mostly handle Objects, not directly values – except for very simple objects which mostly carry a value, like Integer, Boolean, Character or Float. It is thus necessary to model specialized objects that reflects the external values, that is the ExternalStructure, ExternalUnion, ExternalData and ExternalAddress classes. ExternalStructure and ExternalUnion are a reflexion of C data of type struct and union, ExternalAddress a reflexion of pointers to a memory area outside Smalltalk memory. ExternalData is more mysterious, the name is not specific at all. We will have to come back to it later.

We have shortly introduced all the classes composing the FFI-Kernel category. Those classes are the face of FFI visible at image side. There is also the invisible face hidden in the VM, the FFI plugin. We will see it in the more advanced details of implementation.

Specifying the external function interface

Since Smalltalk is messages all the way down, an external function will integrate seamlessly in the Smalltalk world if it can be invoked in response to an ordinary message. That's exactly what FFI provides: instead of instructions written in Smalltalk, a FFI method has a hook for interfacing to the external function.

The syntax for this hook looks a bit like the one used for linking a primitive: method in the VM: its uses a form of the so called pragma syntax – or better named annotation syntax.

Those annotation syntax take the form of a sequence of literals laid out in a pair of enclosing angle brackets < >.

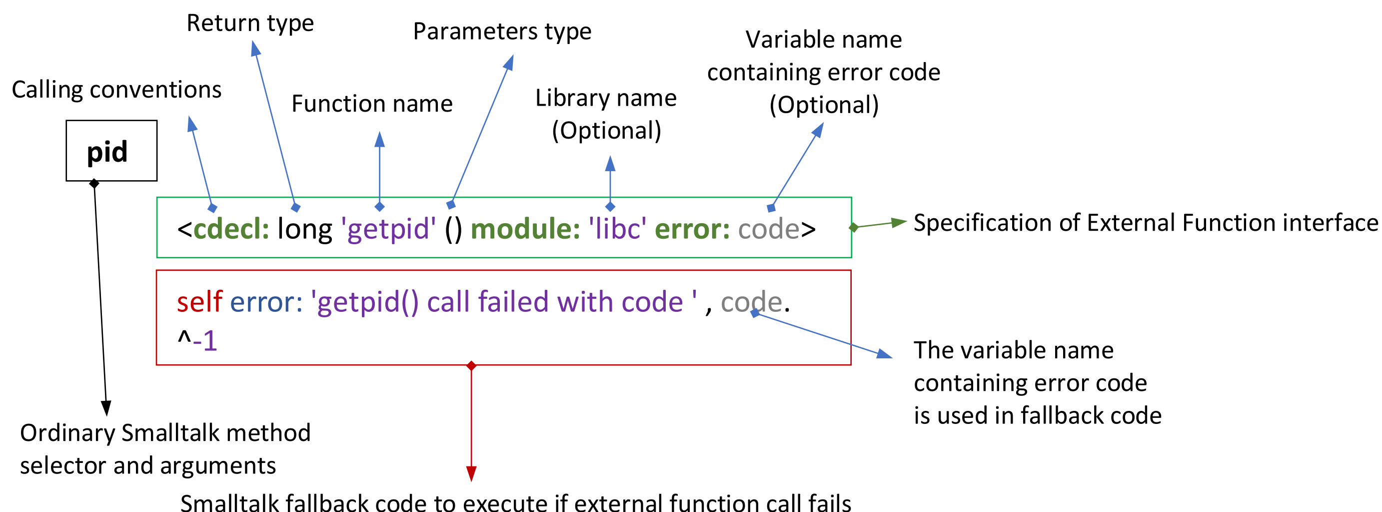
Like for primitives, the regular Smalltalk instructions that are following the pragma will be executed as fallback if ever the execution of foreign function failed. Else, if the external function call succeeds, control is returned directly without executing those Smalltalk instructions.

There are several reasons why the call may fail, like the library or function was not found, or that a mismatch has been found between the type specified in the interface and the actual arguments, or the specified type is not consistent.

Pragma or annotations works with keyword, a lot like the keyword message model. For FFI, the possible keywords are the following:

- First one is calling convention cdecl: or apicall:
- optionally followed by module: specification
- optionally followed by error: specification

A simple example without parameter looks like this:



Regular pragmas or annotations have a single literal parameter per keyword. However, this is not the case for FFI, which was created before the generalization of annotations. As we can see, 3 literals follow the calling convention keyword:

- a return type
- the name of the function (or eventually number in Windows DLL)
- a list of function parameter types (one per argument of the Smalltalk message)

The other two optional keywords module: and error: get a single parameter, the module name (a String) and the name of a temporary variable (word or String) that will contain the error code upon return in case of failure. Note that the variable must not be declared in temporaries bar | |, this is automatic, otherwise the Parser will signal an error.

Note that the receiver plays no role in the external call, only arguments of the message are passed to the external function, not the receiver. FFI methods are sort of utility methods. There are two main schools for the choice of class implementing the method:

- anywhere related to the utility (here pid could be implemented in SmalltalkImage so that we can write Smalltalk pid and get the pid of the Smalltalk process).
- In a class reifying the library.

In the last case, the moduleName can be factored out as a class side message. This can be convenient if we want to support multiple platforms/os and if the name of library is platform specific.

Specifying the external library

Specifying the library directly in the function interface

As we saw in the example of previous chapter, the name of library can be specified directly in the External function interface specification. That's the simplest but not unique solution.

Specifying the library at class side

What if we omit this module name? Then

The list of possible FFI errors

The list of errors can be found in FFIConstants class>>initializeErrorConstants. Following table give an image of that method.

Beware: the error code returned when the external call fails is the FFI code below plus an offset, so that both regular primitive error code and FFI specific error code can be distinguished. See ExternalFunction class>>externalCallFailedWith:

Table 1 The list of Squeak FFI Errors

FFI name	FFI code	Notes
FFINoCalloutAvailable	-1	No callout mechanism available – OBSOLETE ?
FFIErrorGenericError	0	A call to an external function failed – may happen for whatever reason a primitive may fail
FFIErrorNotFunction	1	The first literal of a FFI method is not an ExternalLibraryFunction
FFIErrorBadArgs	2	The ExternalLibraryFunction is malformed (does not have expected flags, argTypes not corresponding to num parameters + 1, …)
FFIErrorBadArg	3	Attempt to pass an argument of wrong class to a pointer or to a structure type
FFIErrorIntAsPointer	4	Attempt to pass an Integer value (or Character or Float) to a pointer type
FFIErrorBadAtomicType	5	The specification of an atomicType is not valid
FFIErrorCoercionFailed	6	The coercion of argument to integer value or pointer value failed
FFIErrorWrongType	7	The type specified has unconsistencies.
FFIErrorStructSize	8	Attempt to pass an ExternalStructure value which has not the same size as the type specification
FFIErrorCallType	9	Unsupported calling convention
FFIErrorBadReturn	10	Attempt to return a structure/union but no corresponding ExternalStructure class was found at image side

FFIErrorBadAddress	11	The handle of the ExternalLibraryFunction is not a specification of valid address
FFIErrorNoModule	12	No module was given for loading external function address
FFIErrorAddressNotFound	13	The external function was not found in specified module
FFIErrorAttemptToPassVoid	14	Attempt to pass a void value. It's possible to pass void *, or return void (nothing), but not to pass void!
FFIErrorModuleNotFound	15	The module specified was not found
FFIErrorBadExternalLibrary	16	The handle of the ExternalLibrary is not a specification of valid address
FFIErrorBadExternalFunction	17	The ExternalLibraryFunction name is not a ByteArray or ByteString
FFIErrorInvalidPointer	18	The ExternalAddress point to some Smalltalk memory, which is forbidden (objects in Smalltalk memory can be relocated leading to dangling pointers)
FFIErrorCallFrameTooBig	19	The function requires more than 16k bytes to pass arguments

Specifying the types

Simple atomic types

FFI has support for passing signed and unsigned integer of different size and also floating-point types (single or double precision). Atomic types are specified with a simple word known to FFI, each corresponding to a C type. Those types are initialized in ExternalType class>>initializeAtomicTypes.

Table 2 The atomic types supported by Squeak FFI

FFI name	**C type**	**Notes**
void	void	For specifying void pointers or absence of returned value
bool	uint8_t	For passing Smalltalk Boolean
byte	uint8_t	For passing Smalltalk Integer (or ByteArray if pointer)
sbyte	int8_t	“
ushort	uint16_t	For passing Smalltalk Integer (or DoubleByteArray if pointer)
short	int16_t	“
ulong	uint32_t	For passing Smalltalk Integer (or WordArray if pointer)
long	int32_t	“
ulonglong	uint64_t	For passing Smalltalk Integer (or DoubleWordArray if pointer)
longlong	int64_t	“
char	unsigned char	For passing Smalltalk Byte Character (or ByteString if pointer)
schar	signed char	“
float	float	Single precision float
double	double	Double precision float

Beware, in this table the FFI word 'long' does not correspond to C type long. It did so at the era of 32 bits, and was less ambiguous than 'int' at the time when some platforms did have a 16 bits int (DOS being an example). But at the era of 64 bits, the C type long is rather int64_t for LP64 platforms (though still int32_t in LLP64 platforms – mostly windows64).

We can also notice several different 8 bits integer. That is because C type char does not really distinguish if the type represents a boolean, a character code, or an integer. In Smalltalk, Boolean, Character and Integer are separated classes with different behavior. Historically, FFI has duplicated the C type, one for converting Smalltalk Character, one for converting byte Integer. But nowadays, no such distinction is made for atomic values.

The type 'bool' is declared as being 1 byte long at image side, but is converted to (4 bytes) int in the plugin. That's a mismatch that can be troubling, but has no real consequence, because both are passed the same way in all supported platforms ABI.

What Smalltalk object can be passed to a parameter of atomic integer type? All the objects in the following table can, with or without restriction.

Table 3 value taken into account for atomic integer type

Smalltalk class	**Value retained**	**Restrictions**
UndefinedObject	0	None
False	0	None
True	1	None
Character	charCode	None
SmallInteger	value	None
LargePositiveInteger	value	Value must be < (1<<32) on 32 bits VM for target type up to int32_t Value must be < (1<<64) on 64 bits VM for target type up to int32_t Value must be < (1<<64) for uint64_t Value must be < (1<<63) for int64_t
LargeNegativeInteger	value	Value must be >= -(1<<31) on 32 bits VM for target type up to int32_t Value must be >= -(1<<63) on 64 bits VM for target type up to int32_t Value must be >= -(1<<63) for int64_t

What must be understood is that no bound check is performed on Character nor SmallInteger values. According to the C type, and up to 32 bits, the retained value is truncated to 1, 2 or 4 bytes before being passed to the external function (like a C conversion with overflows, it is equivalent to a modulo operation).

For example when passing a wide Character (i.e. Character euro) to a signed or unsigned char, a modulo will be applied to the charCode, and the external call won't fail. Character euro charCode = 16r20AC, thus 16AC is passed to an unsigned char, or -16r54 to a signed char.

However, bound checks are performed on LargeInteger values. For int64_t and uint64_t, it is mandatory that the LargeInteger value fits on target type, otherwise the primitive fails.

For types up to 32 bits, the LargeInteger value may overflow, but not more than would fit on machine Word size (4 bytes on 32 bits VM, 8 bytes on 64 bits VM).

It may seem odd that passing (1<<64-1) to a parameter expecting an int32_t would not fail on a 64 bits VM, but passing the same value to a parameter expecting an int64_t would fail! But this is not a description of how things should be, just how they currently are.

For floating point type, the same integer values can be passed, but without the specific int64_t and uint64_t rules, they will be converted to single precision float, or double.

Squeak Float (which are IEEE 754 double) will also be passed unchanged to double, or cast to single precision if target is single precision float. This later conversion may imply rounding or overflow to +/- infinity.

Table 4 value taken into account for atomic float type

Smalltalk class	Value retained	Restrictions
UndefinedObject	0	None
False	0	None
True	1	None
Character	charCode	None
SmallInteger	value	None
LargePositiveInteger	value	Value must be < (1<<32) on 32 bits VM for target type up to int32_t Value must be < (1<<64) on 64 bits VM for target type up to int32_t
LargeNegativeInteger	value	Value must be >= -(1<<31) on 32 bits VM for target type up to int32_t Value must be >= -(1<<63) on 64 bits VM for target type up to int32_t
Float	Value	None

Composite types

For passing one of the C composite types (struct or union), it is necessary to first create an ExternalStructure or ExternalUnion subclass which will serve as proxy to the value of corresponding C type.

The argument must then be an instance of the subclass of ExternalStructure or ExternalUnion, and the type specification used in FFI external function interface must be the name of that subclass.

It is then possible to instantiate the ExternalStructure subclass in Smalltalk memory and pass that object to the external function call. It is also possible to have such external function returning a new instance of the the ExternalStructure subclass when the external function returns a struct by value.

EXAMPLE PLEASE

WHAT ABOUT STRUCT ALIGNMENT?

The type aliases

Other C types, enum, bitfields

In C, enum are just specific int types. Thus enum are treated like alias to int type.

EXAMPLE WANTED

There is no support for bitfield in struct. Bit fields must be replaced by integer of appropriate length, and it is user responsibility to decompose the bits of the integer field using available bit operation in Squeak (bitAt: bitShift: bitOr: bitAnd: allMask: anyMask: etc…).

Pointers: passing values by reference

So far, we only considered values. On important feature of C is the ability to pass a reference to a value (a pointer the memory where the value is located). The purpose is

- Either to avoid a costly copy on stack frame for big objects
- Or to have the value modified by the external function

In modern C, it's possible to declare a pointer to a constant, read from right to left: type const *. If we would have the information that the intent is read-only, then it would be possible to automatically convert a value, copy it in a container of expected C type and call the external function with address of container. But in Squeak FFI, we have no mean to distinguish those usages. It is thus mandatory to pass an instance of the following classes:

- An ExternalAddress pointing to some external heap;
- A ByteArray of appropriate length that may act as the container;
- An ExternalStructure corresponding to the type specification;
- An ExternalData corresponding to the type specification.

Note that the ExternalStructure and ExternalData both have a handle that may point to an ExternalAddress or to a ByteArray. The last two cases enable type checking while the first two completely bypass the type checks.

Opaque handles

In a number of libraries, some struct never need to be allocated in Smalltalk memory, and their contents is never accessed directly: because the struct will be allocated by some specialized external function, returned by address, and passed to other specialized function. Much like a Smalltalk object, the internal states of such opaque struct are hidden and we operate on it only via function call, like we would operate on a Smalltalk object only by sending messages. In this case, it's best to not detail the layout of the struct, but just pass a void * pointer.

The most proper way to represent such opaque handle is to create an aliased type, either to a void *, or to an integer of sufficient length for holding a pointer (an intptr_t, but we do not have such type support in FFI), or just an empty stucture.

```
fields
	^#(nil	'void*')


fields
	^Smalltalk wordSize = 4
		ifTrue: [#(nil	'ulong') "an opaque 32bit handle"]
```

```
                    ifFalse: [#(nil   'ulonglong') "an opaque 64bit handle"]
fields
          ^#()
```

Beware, though, in the latter case, having this aliased type as struct member may lead to ambiguous encoding of type specification (compiledSpec) – we'll see why in dedicated chapter.

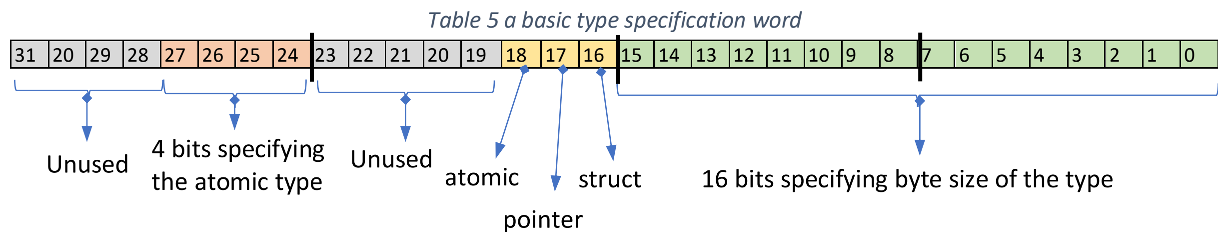
Passing Smalltalk objects to the external function

Implementation details of type checking

The types are encoded in so called compiledSpec. A compiledSpec is a WordArray, that is an array of 32bits unsigned words.

Each 32bits word is a basic type specification, in which the least two significant bytes encode the size of the data in bytes, the next byte holds some flags for interpreting the type and the most significant byte encodes the atomic type (if any).

Table 5 a basic type specification word



Note that if the pointer bit is set, then the size specified in least two bytes is that of a pointer sizeof(void *), that is 4 in 32 bits VM, 8 in 64 bits VM.

Table 6 The interpretation of the 3 atomic-pointer-struct bits

A	P	S	How to decode the atomic-point-struct type flags
0	0	0	Unused - illegal
0	0	1	A structure (defined by value)
0	1	0	A pointer to a structure or other unspecified type (void *)
0	1	1	A pointer to struct – possible but currently unused
1	0	0	An atomic type (passed by value)
1	0	1	Unused - illegal
1	1	0	A pointer to an atomic type
1	1	1	Unused - illegal

The bits specifying the atomic type in the most significant byte, are set if and only if the kind of type is atomic. Note that the encoding can be found in ExternalType class>>initializeAtomicTypes

Table 7 The encoding of atomic types supported by Squeak FFI

27	26	25	24	**FFI name**	**C type**	**Notes**
0	0	0	0	void	void	For specifying void pointers
0	0	0	1	bool	uint8_t	For passing Smalltalk Boolean
0	0	1	0	byte	uint8_t	For passing Smalltalk Integer
0	0	1	1	sbyte	int8_t	"
0	1	0	0	ushort	uint16_t	"
0	1	0	1	short	int16_t	"
0	1	1	0	ulong	uint32_t	"
0	1	1	1	long	int32_t	"
1	0	0	0	ulonglong	uint64_t	"
1	0	0	1	longlong	int64_t	"
1	0	1	0	char	unsigned char	For passing Smalltalk Byte Character
1	0	1	1	schar	signed char	"
1	1	0	0	float	float	Single precision float
1	1	0	1	double	double	Double precision float
1	1	1	0			UNUSED
1	1	1	1			UNUSED

Note that bits 25 to 27 colored in blue are equal to the byteSize for the atomic integer types (codes 2 to 9), while bit 24 specify the signedness. This bit trick is not used in the implementation. However, it suggests that we could have even more compact representation if necessary.

Simple types like atomic types or pointers to atomic types have a compiledSpec made of a single word as described above. For example, ExternalType double compiledSpec will be a WordArray with: 16r0D040008, which we decode as atomic type=16r0D=2r1101=double, flags=16r04=atomic, sizeof=16r0008 bytes.

Composite types like struct and union have a more elaborate compiledSpec. The first word describes the structure or union as a whole (that is the byteSize holds the whole size of the struct including padding and alignment bytes). The last word is a stop word, which marks the end of the struct description. This stop word is necessary in order to support nested structures (struct whose field is another struct). Between those first and last word, the type of each field is described with a single word if atomic, or several words if composite – ending with the stop word.

The stop word is encoded with a single struct bit set to 1, all other set to zero (that is 65536 or 16r00010000).

If we take a simple struct {double d; int i;}; the struct is 8 for d+4 for i+4 for alignment padding, that is 16 bytes. The first word of compiled spec is thus 16r00010010.

The next two words are the double type spec 16r0D040008, and the int32_t type spec 16r07040004. The last word is the stop word 16r00010000.

What about pointers to struct

For a pointer to struct, we could expect to have both struct+pointer flags set. That's not the case. The basic type specification has just pointer, and is thus equivalent to void * 16r00020004 or 16r00020008 on 32 and 64bits VM respectively.

What about aliased types

An aliased atomic type has the same basic type specification as the target atomic type. For example, is Size_t is an alias to uint32_t (on a 32 bits VM), then its compiledSpec will be that of uint32_t, 16r06040004. However, the pointer to a Size_t will be defined as that of a struct, void *, and that probably qualify as a bug. It will not crash, but void * will bypass type checking which is not a good thing.

Proposal for future enhancement:

We can see that we have unused bits in this spec. It's good to have some for being future proof. But we also see that some information is redundant: the size of atomic type could be encoded in the encoded type. Hence, we could just keep 3 bits for the atomic type:

- 000=void
- 001=bool
- 010=unsigned integer
- 011=signed integer
- 100=unsigned char
- 101=signed char
- 110=floating point
- 111 reserved

Also there is a flag for struct but none for union. It is necessary to distinguish struct from union in SysV x86_64 ABI when testing if the composite value can be passed via stack or registers.

The kind of registers of SysV x86_64 ABI could also be cached in some of the unused bits.

Those compiledSpec are accessible to the FFI Plugin thru following object graph: the first literal of a FFI method is an instance of ExternalLibraryFunction. This object points to the argTypes (the first of which being the function return type). Each argType is an instance of ExternalType. Each external type point to a compiledSpec and a referentClass. The

referentClass is either nil, or a subclass of ExternalStructure if the type is that of an existing structure. The compiledSpec and referentClass are the essential information which serve the control of parameter marshalling.

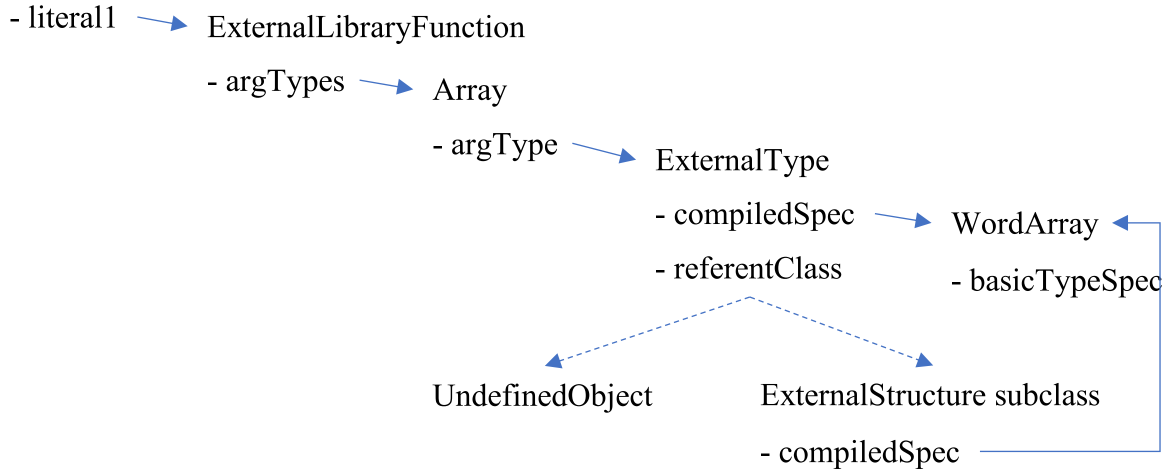


Figure 1 Object graph: FFI CompiledMethod pointing to ExternalType referentClass and compiledSpec

In order to understand what is a possible combination of argument type specification and actual argument object, it is necessary to dive into the FFI Plugin. The code is a bit complex with a lot of logic, and the result of reverse engineering is presented in following table. We can observe a few unexpected behaviors:

-

Table 8 Currently possible combination of type and object (top) versus expected or wished (bottom)

<table>
<tr><th colspan="5">argSpecType</th><th>Method</th><th colspan="11">arg oop/oop class</th></tr>
<tr><th colspan="3">argSpec= compiledSpec</th><th colspan="2">argClass=referentClass</th><th></th><th>nil</th><th>true</th><th>false</th><th>Integer</th><th>Character</th><th>Float</th><th>String</th><th>ExternalStruct</th><th>ExternalData</th><th>ExternalAddress</th><th>ByteArray</th></tr>
<tr><th>Atomic</th><th>Pointer</th><th>Structure</th><th>nil</th><th>ExternalStructure</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th colspan="2">isStruct = true</th><th colspan="2">Unsafe : no type checking</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>ffiArgByValue:</td><td colspan="6">push value</td><td colspan="5">FFIErrorCoercionFailed</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>ffiArgByValue:</td><td colspan="7">FFIErrorCoercionFailed</td><td colspan="2">FFIErrorCoercionFailed</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>ffiAtomicArgByReference: / ffiAtomicStructByReference:</td><td>Null pointer</td><td colspan="5">FFIErrorCoercionFailed</td><td>memcpy on stack</td><td>FIErrorCoercionFaile</td><td>push pointer</td><td>memcpy on stack</td><td>memcpy on stack if not threaded</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>ffiAtomicArgByReference: / ffiAtomicStructByReference:</td><td colspan="7">FFIErrorCoercionFailed</td><td>FIErrorCoercionFaile</td><td>oop class inherits argClass push</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>ffiPushStructureContentsOf:</td><td colspan="9">FFIErrorBadArg</td><td>memcpy on stack</td><td>memcpy on stack if not threaded</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>ffiPushStructureContentsOf:</td><td colspan="7">FFIErrorCoercionFailed</td><td>oop class inherits argClass</td><td>oop class inherits argClass</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>ffiPushStructureContentsOf:</td><td colspan="9">FFIErrorBadArg</td><td>memcpy on stack</td><td>memcpy on stack if not threaded</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>ffiPushStructureContentsOf:</td><td colspan="7">FFIErrorCoercionFailed</td><td>oop class inherits argClass</td><td>oop class inherits argClass</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>FFIErrorWrongType</td><td colspan="11">FFIErrorWrongType</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>FFIErrorWrongType</td><td colspan="11">FFIErrorWrongType</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>FFIErrorWrongType</td><td colspan="11">FFIErrorWrongType</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>FFIErrorWrongType</td><td colspan="11">FFIErrorWrongType</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>FFIErrorWrongType</td><td colspan="11">FFIErrorWrongType</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>FFIErrorWrongType</td><td colspan="7">FFIErrorCoercionFailed</td><td colspan="2">FFIErrorWrongType</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>ffiPushPointerContentsOf:</td><td>Null pointer</td><td colspan="6">FFIErrorCoercionFailed</td><td>push pointer</td><td>push pointer</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>ffiPushPointerContentsOf:</td><td colspan="7">FFIErrorCoercionFailed</td><td>oop class inherits argClass</td><td>oop class inherits argClass</td><td colspan="2">FFIErrorCoercionFailed</td></tr>
</table>

argSpecType					Method	arg oop/oop class										
argSpec=compiledSpec			argClass=referentClass			nil	true	false	Integer	Character	Float	String	ExternalStruct	ExternalData	ExternalAddress	ByteArray
Atomic	Pointer	Structure	nil	ExternalStructure									isStruct = true		Unsafe : no type checking	
1	0	0	1	0	ffiArgByValue:	push value						FFIErrorCoercionFailed				
1	0	0	0	1	ffiArgByValue:	push value						FFIErrorCoercionFailed				
1	1	0	1	0	ffiAtomicArgByReference: / ffiAtomicStructByReference:	Null pointer	FFIErrorCoercionFailed					memcpy on stack	FFIErrorCoercionFailed	push pointer if type match	memcpy on stack	memcpy on stack if not threaded
1	1	0	0	1	ffiAtomicArgByReference: / ffiAtomicStructByReference:	Null pointer	FFIErrorCoercionFailed					memcpy on stack	push pointer if type match	push pointer if type match	memcpy on stack	memcpy on stack if not threaded
0	0	1	1	0	FFIErrorWrongType	we cannot push an undefined structure by value										
0	0	1	0	1	ffiPushStructureContentsOf:	FFIErrorCoercionFailed							memcpy on stack if type match	memcpy on stack if type match	FFIErrorCoercionFailed	
0	1	1	1	0	ffiPushStructureContentsOf:	Null pointer	FFIErrorCoercionFailed							push pointer if type match	push pointer	push pointer if not threaded
0	1	1	0	1	ffiPushStructureContentsOf:	Null pointer	FFIErrorCoercionFailed					memcpy on stack	push pointer if type match	push pointer if type match	FFIErrorCoercionFailed	
1	0	1	1	0	FFIErrorWrongType	FFIErrorWrongType : we do not want to handle										
1	0	1	0	1	FFIErrorWrongType											
1	1	1	1	0	FFIErrorWrongType											
1	1	1	0	1	FFIErrorWrongType											
0	0	0	1	0	FFIErrorWrongType											
0	0	0	0	1	FFIErrorWrongType											
0	1	0	1	0	FFIErrorWrongType											
0	1	0	0	1	FFIErrorWrongType											

Color	Meaning
(purple)	Unconsistent spec, should not happen
(yellow)	named type with no corresponding class
(light orange)	Aliased atomic type
(red)	FFI return error
(light green)	Valid FFI argument on some condition
(green)	Valid FFI argument

The limitations of Squeak FFI

No support for array

No type longer than 65535 bytes

No support for pointer arity

What if the Smalltalk value is outside bounds of C type value?

What is the search path of ExternalLibrary?

How to handle cross platform and platform specific libraries?

What if a function takes more than 15 arguments?

Why ExternalData inherits from ExternalStructure?

The callback to Smalltalk code.

The threaded FFI.