Redesigning FFI calls in Pharo

Exploiting the baseline JIT for more performance and low maintenance

Bianchi Juan Ignacio
Polito Guillermo

Roadmap

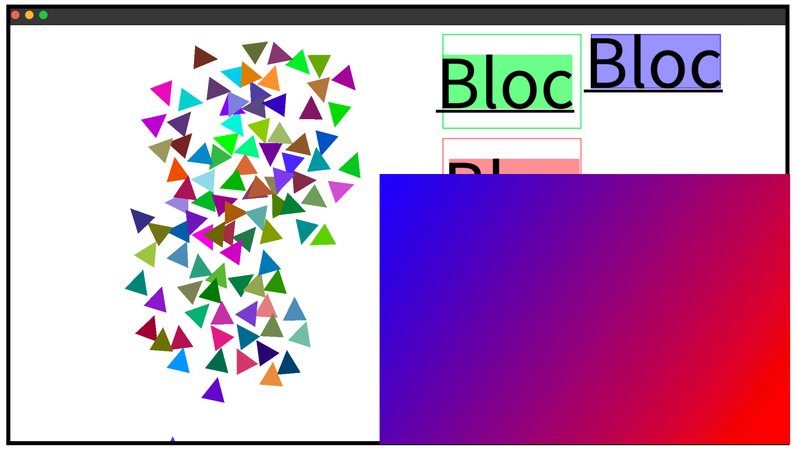
- **FFI and why do we need it**
- Current FFI implementation and its problems
- Our new design and how it solves those problems
- Early results

Foreign Function Interfaces

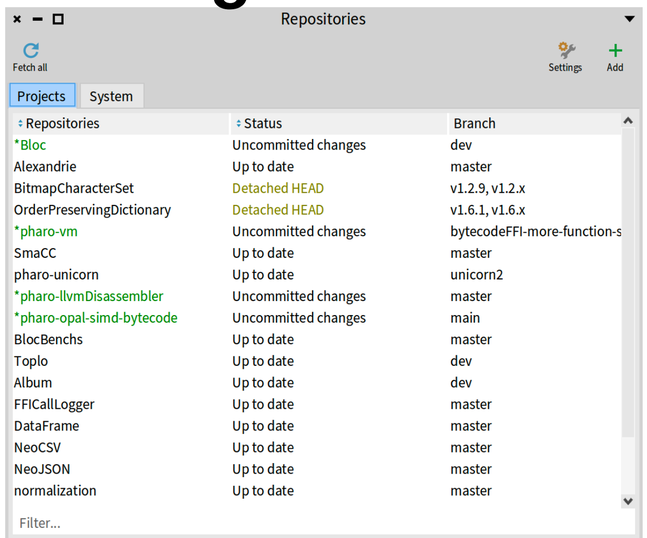
- Mechanism to interoperate between languages
- For example, calling C from Pharo.
- Based on a *binary contract, a.k.a.* an ABI (Application Binary Interface)

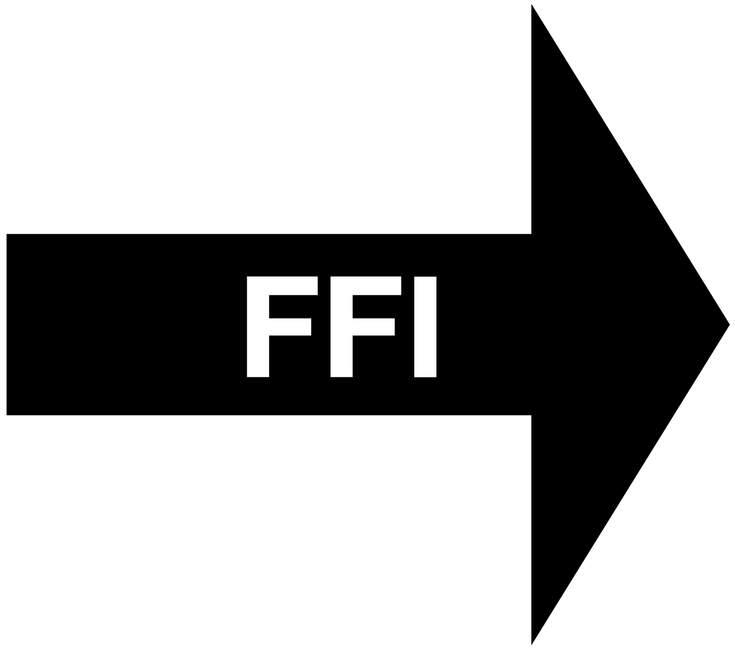
Pharo calls into C a lot using FFI

Bloc



Iceberg

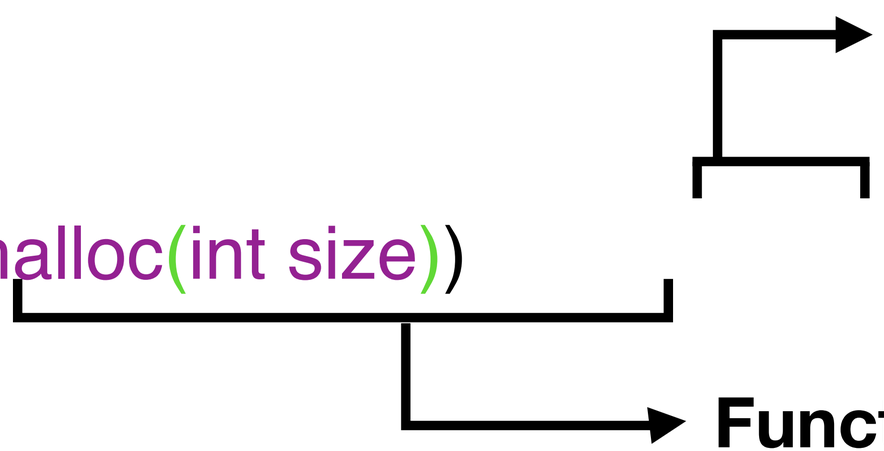




etc.

Foreign Function Interfaces Example

```
malloc: size

    ^ self ffiCall: #(void* malloc(int size))
```



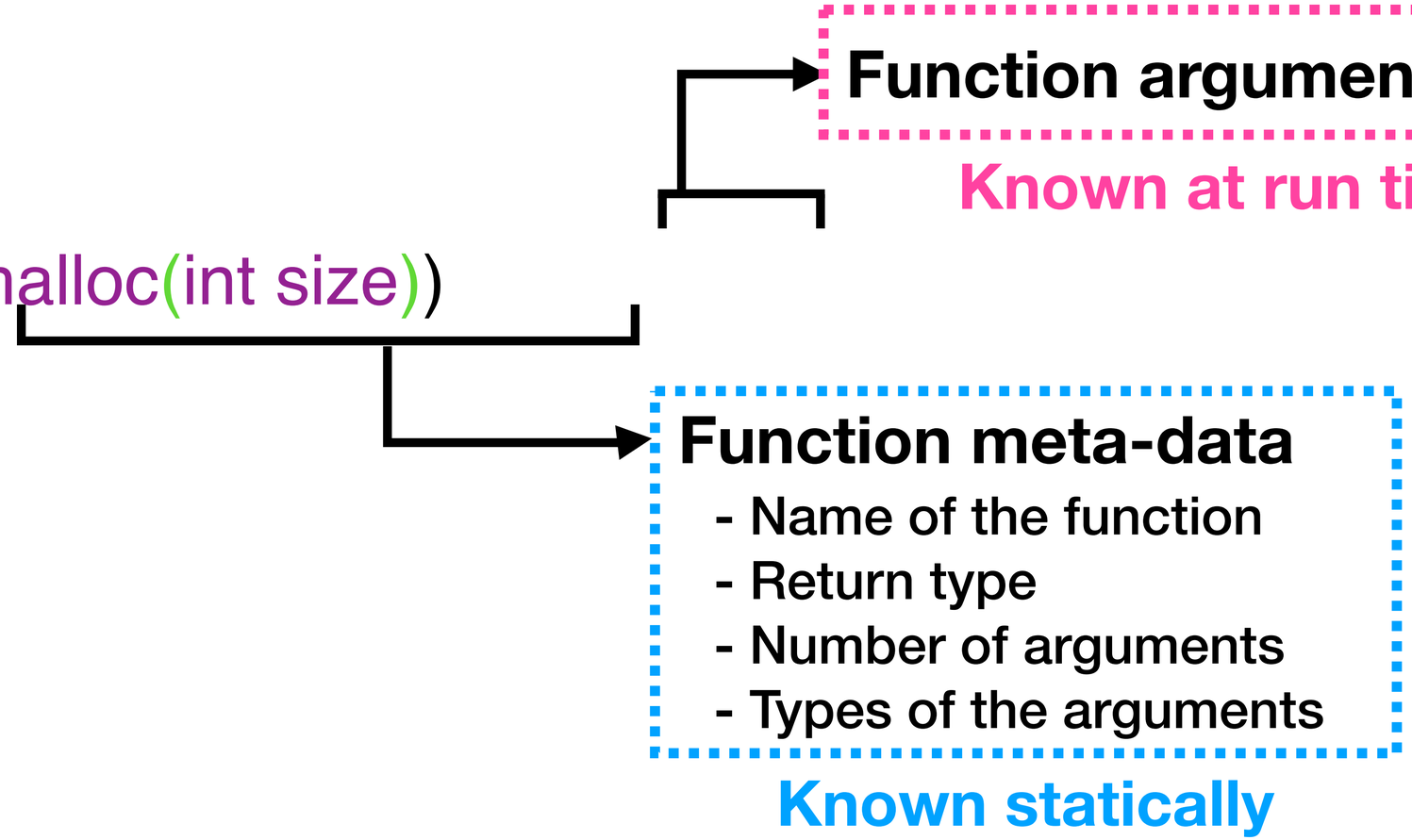
Function argument

Function meta-data

- Name of the function
- Return type
- Number of arguments
- Types of the arguments

Foreign Function Interfaces Example

```
malloc: size

    ^ self ffiCall: #(void* malloc(int size))
```



Roadmap

- FFI and why do we need it
- **Current FFI implementation and its problems**
- Our new design and how it solves those problems
- Early results

One FFI Primitive to rule them all

- **Only** in the interpreter

```
Interpreter >> primitiveFFICallout
    | functionMetadata argumentArray |
    functionMetadata := self pop.
    argumentArray := self pop.
    argumentArray := self marshallArguments: argumentArray
                            usingFunctionMetadata: functionMetadata.
    result := self
        ffiCall: functionMetadata
        arguments: argumentArray.
        …
```

One FFI Primitive to rule them all

- **Only** in the interpreter
- Function meta-data known

at run time

```
Interpreter >> primitiveFFICallout
    | functionMetadata argumentArray |
    functionMetadata := self pop.
    argumentArray := self pop.
    argumentArray := self marshallArguments: argumentArray
                                usingFunctionMetadata: functionMetadata.
    result := self
        ffiCall: functionMetadata
        arguments: argumentArray.
        …
```

One FFI Primitive to rule them all

- **Only** in the interpreter
- Function meta-data known

at run time

- Run time checks of arguments

```
Interpreter >> primitiveFFICallout
    | functionMetadata argumentArray |
    functionMetadata := self pop.
    argumentArray := self pop.
    argumentArray := self marshallArguments: argumentArray
                              usingFunctionMetadata: functionMetadata.
    result := self
        ffiCall: functionMetadata
        arguments: argumentArray.
        …
```

One FFI Primitive to rule them all

- **Only** in the interpreter
- Function meta-data known

at run time

- Run time checks of arguments
- Supports all cases with libffi

```
Interpreter >> primitiveFFICallout
    | functionMetadata argumentArray |
    functionMetadata := self pop.
    argumentArray := self pop.
    argumentArray := self marshallArguments: argumentArray
                            usingFunctionMetadata: functionMetadata.
    result := self
        ffiCall: functionMetadata
        arguments: argumentArray.
    ...
```

Analyzing the current implementation

- **Pros** 
  - Simple maintenance: single implementation, leveraging libffi

- **Cons** 
  - General solution incurs high overhead for all cases

Analyzing the current implementation

The most used signatures are often the same ones

	Value
(#void #pointer)	10468
(#void #pointer #double #double)	3840
(#void #pointer #double #double #double)	1308
(#void #pointer #pointer #sint32)	1307
(#void #pointer #pointer #pointer)	1077
(#sint32 #pointer)	587
(#sint32 #pointer #pointer #pointer #pointer)	25
(#uint32 #pointer)	14
(#void #pointer 'TFPointerToStructType')	12

	Value
(#void #pointer)	11269
(#void #pointer #double #double)	4124
(#void #pointer #double #double #double)	1417
(#void #pointer #pointer)	1413
(#void #pointer #pointer #sint32)	1411
(#void #pointer #pointer #pointer)	95
(#sint32 #pointer)	92
(#void #pointer 'TFPointerToStructType')	24

	Value
(#void #pointer)	10177
(#void #pointer #double #double)	4058
(#void #pointer #double #double #double)	2029
(#void #pointer #pointer #pointer)	1085
(#sint32 #pointer)	591
(#sint32 #pointer #pointer #pointer #pointer)	28
(#sint32 #pointer #pointer #pointer)	2
(#sint32 #pointer 'TFPointerToStructType' #pointer #sint32)	1

	Valu
(#void #pointer)	6735
(#void #pointer #double #double)	6556
(#void #pointer #double #double #double)	1672
(#void #pointer 'TFPointerToStructType')	1639
(#void #pointer #pointer #pointer)	1069
(#sint32 #pointer)	563
(#sint32 #pointer #pointer #pointer #pointer)	44

Goal: Redesign FFI to take advantage of the JIT compiler

- **Pros**
  - Simple maintenance: single implementation, leveraging libffi

    => Keep maintenance low

- **Cons**
  - General solution incurs high overhead for all cases

    => Specialize compilation for common function signatures

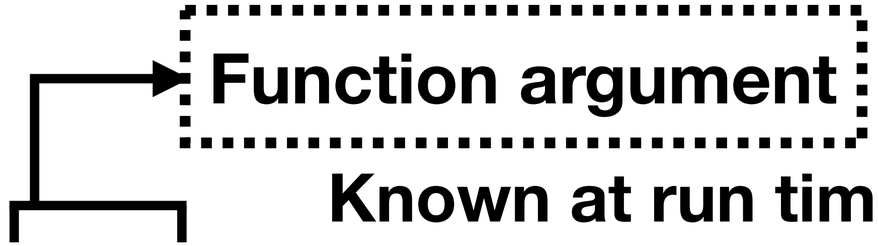
Challenges

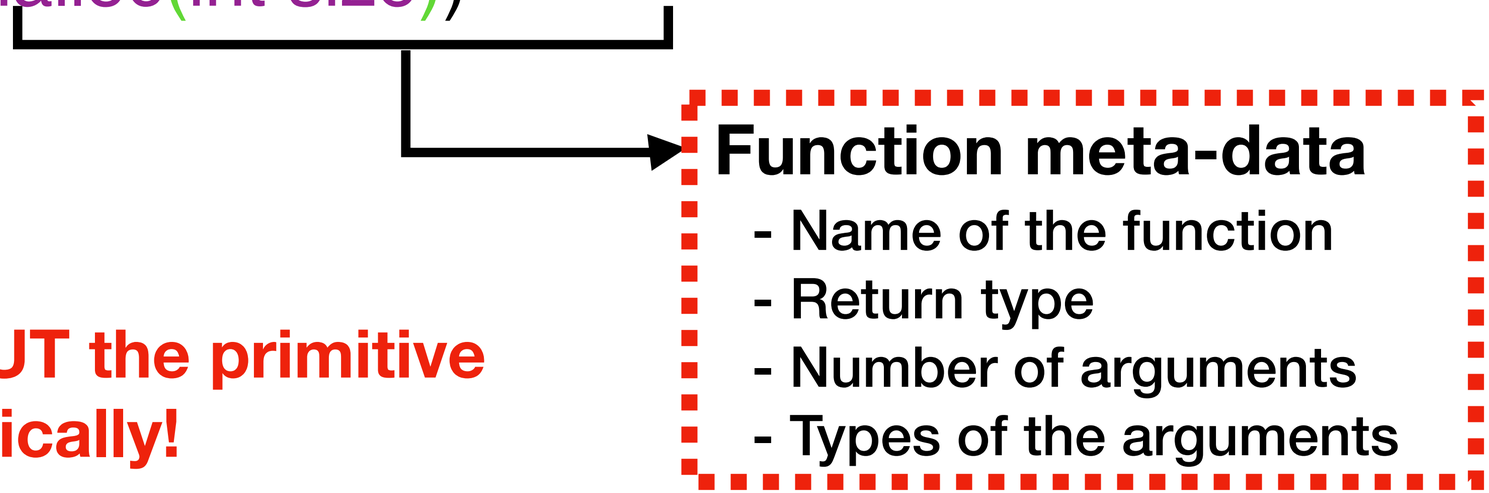
- **VM Primitives do not allow specialization:** Cogit JIT compiler does not support specializing a method/primitive with respect to an argument.

- **Missing compilation context:** Function meta-data is available as a run time argument

Missing compilation context

```
malloc: size

    ^ self ffiCall: #(void* malloc(int size))
```





Known statically BUT the primitive does not use it statically!

It treats it as another run-time argument

Roadmap

- FFI and why do we need it
- Current FFI implementation and its problems
- **Our new design and how it solves those problems**
- Early results

Design Principle: Separate Fast from Slow

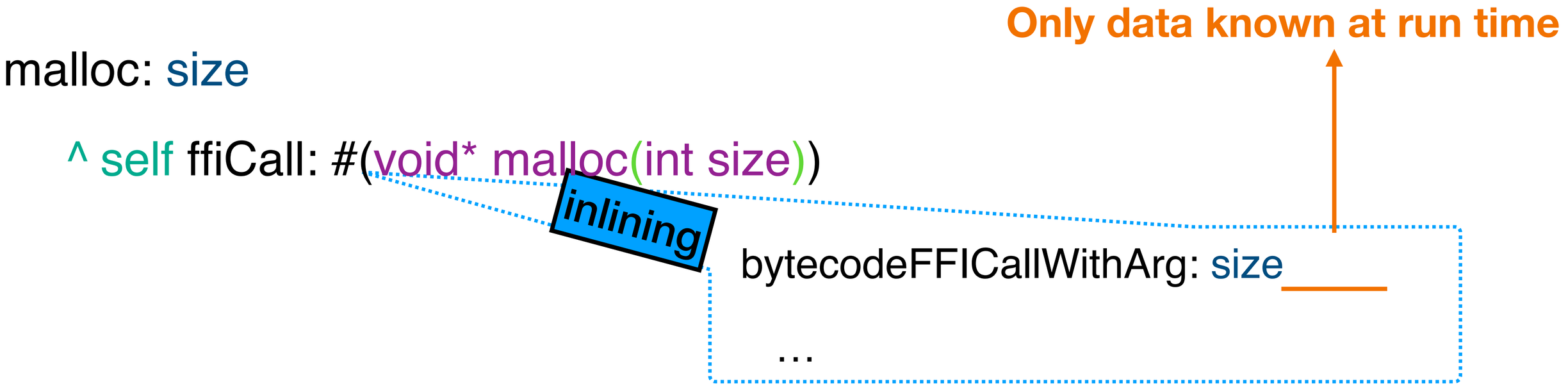
Fast path

- Common function signatures
- JIT specialized
- Leverages compile-time information

Slow path

- All function signatures
- Relies on current primitive
- Performs just like before

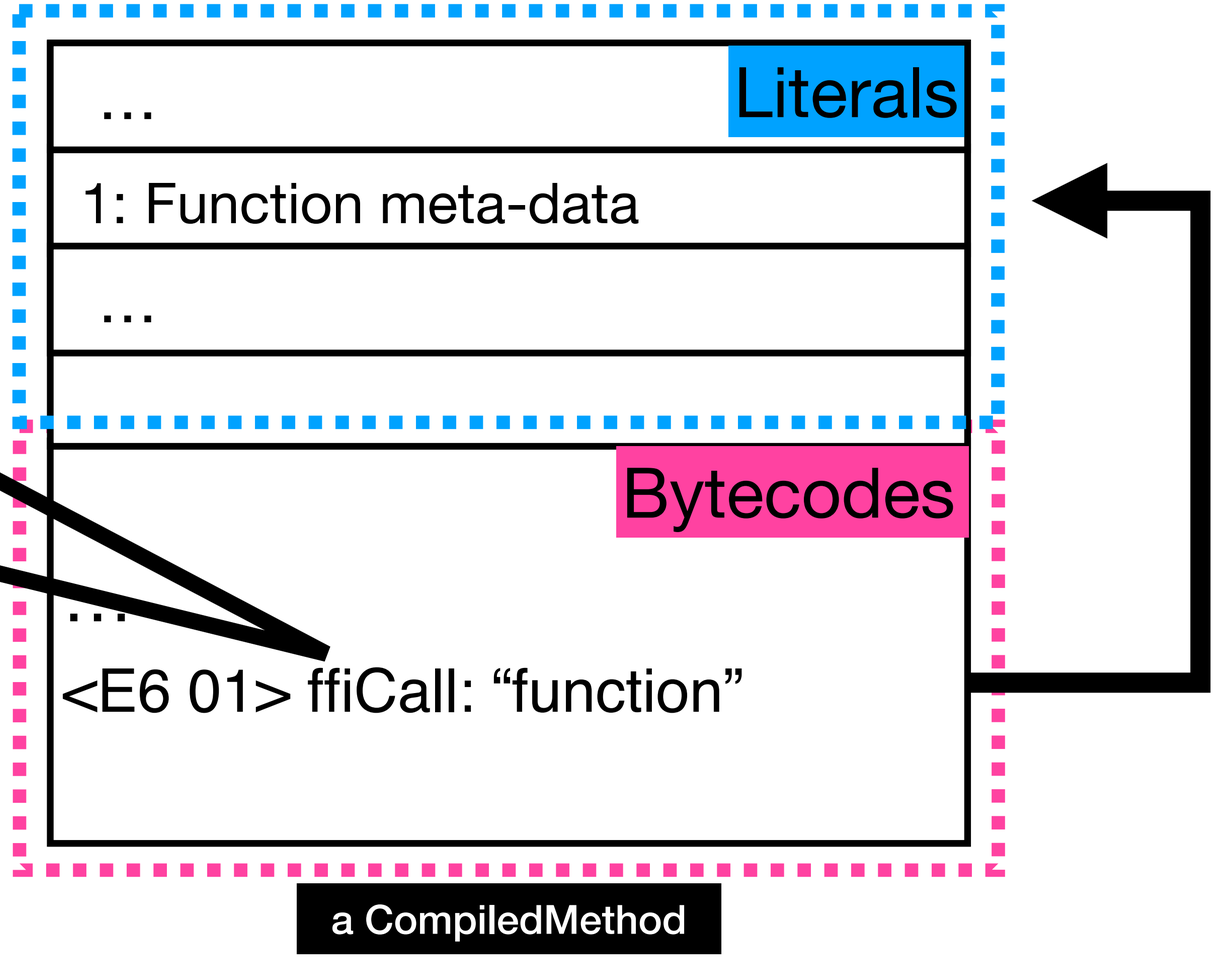
Our solution is based on a new bytecode instruction



The bytecode gets inlined so there is no run time call then we access the function meta-data at compile time.

Accessing to the function meta-data at compile time

```
JIT >> bytecodeFFICall
    | functionMetadata |
    functionMetadata := self getFirstLiteral.


    …
```



Specialize marshaling at compile time

```
JIT >> bytecodeFFICall
    | functionMetadata result |
    functionMetadata := self getFirstLiteral.

    ...

    self popAndMarshallArgumentsUsing: functionMetadata

    ...

    self marshallAndPushResult: result.
```

Convert the arguments (Pharo objects) to C types before passing them to the function

Inverse process with the return value

Specialize function call avoiding libffi

```
JIT >> bytecodeFFICall
    | functionMetadata result |
    functionMetadata := self getFirstLiteral.
    ...
    self popAndMarshallArgumentsUsing: functionMetadata

    self putArgumentsInRegistersUsing: functionMetadata.
    self Call: functionMetadata functionAddress.
    result := self getResultFromRegister.

    self marshallAndPushResult: result
    ...
```

Do the call ourselves:

- **Prepare the arguments**
- **Generate a call instruction**
- **Get the result from register**

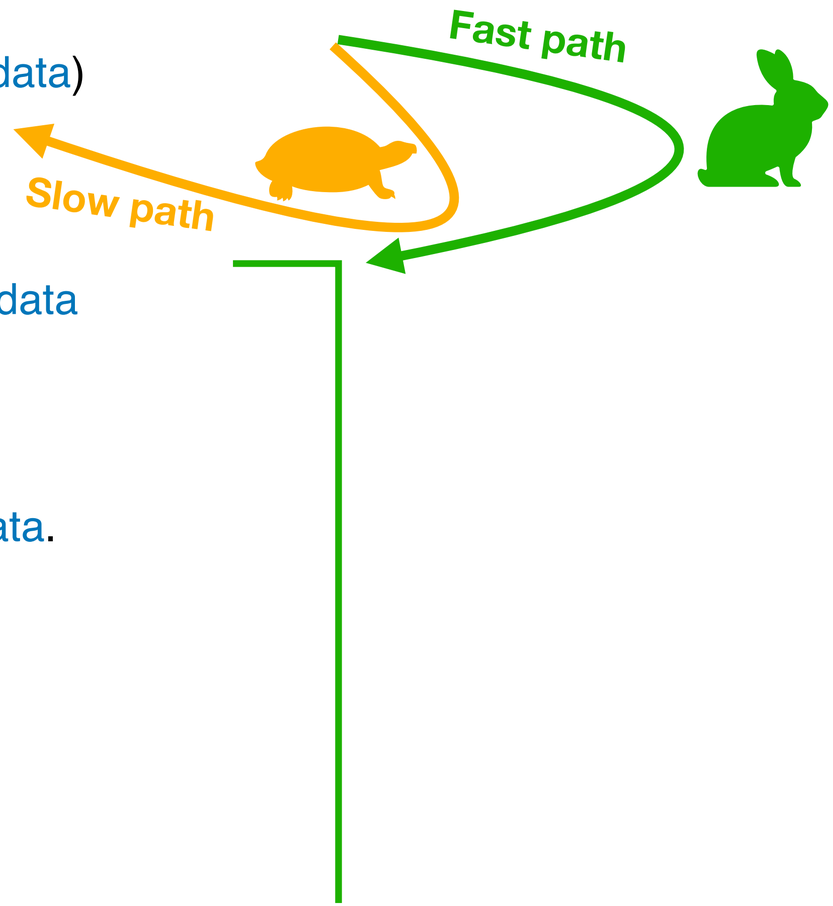
Fallback

For the *unoptimized* signatures and error handling, fallback to the current primitive

```
JIT >> bytecodeFFICall
    | functionMetadata result |
    functionMetadata := self getFirstLiteral.
    (self isFunctionSignatureOptimizable: functionMetadata)
        ifFalse: [ self fallbackToPrimitive ].

    self popAndMarshallArgumentsUsing: functionMetadata
        ifSomeError: [ self fallbackToPrimitive ].

    self putArgumentsInRegistersUsing: functionMetadata.
    self Call: functionMetadata functionAddress.
    result := self getResultFromRegister.

    self selfMarshallAndPushResult: result
        ifSomeError: [ self fallbackToPrimitive ].
```

Key idea: At compile time we detect which path we take

```
JIT >> bytecodeFFICall
    | functionMetadata result |
    functionMetadata := self getFirstLiteral.
    (self isFunctionSignatureOptimizable: functionMetadata)
        ifFalse: [ self fallbackToPrimitive ].

    self popAndMarshallArgumentsUsing: functionMetadata
        ifSomeError: [ self fallbackToPrimitive ].

    self putArgumentsInRegistersUsing: functionMetadata.
    self Call: functionMetadata functionAddress.
    result := self getResultFromRegister.

    self selfMarshallAndPushResult: result
      ifSomeError: [ self fallbackToPrimitive ].
```

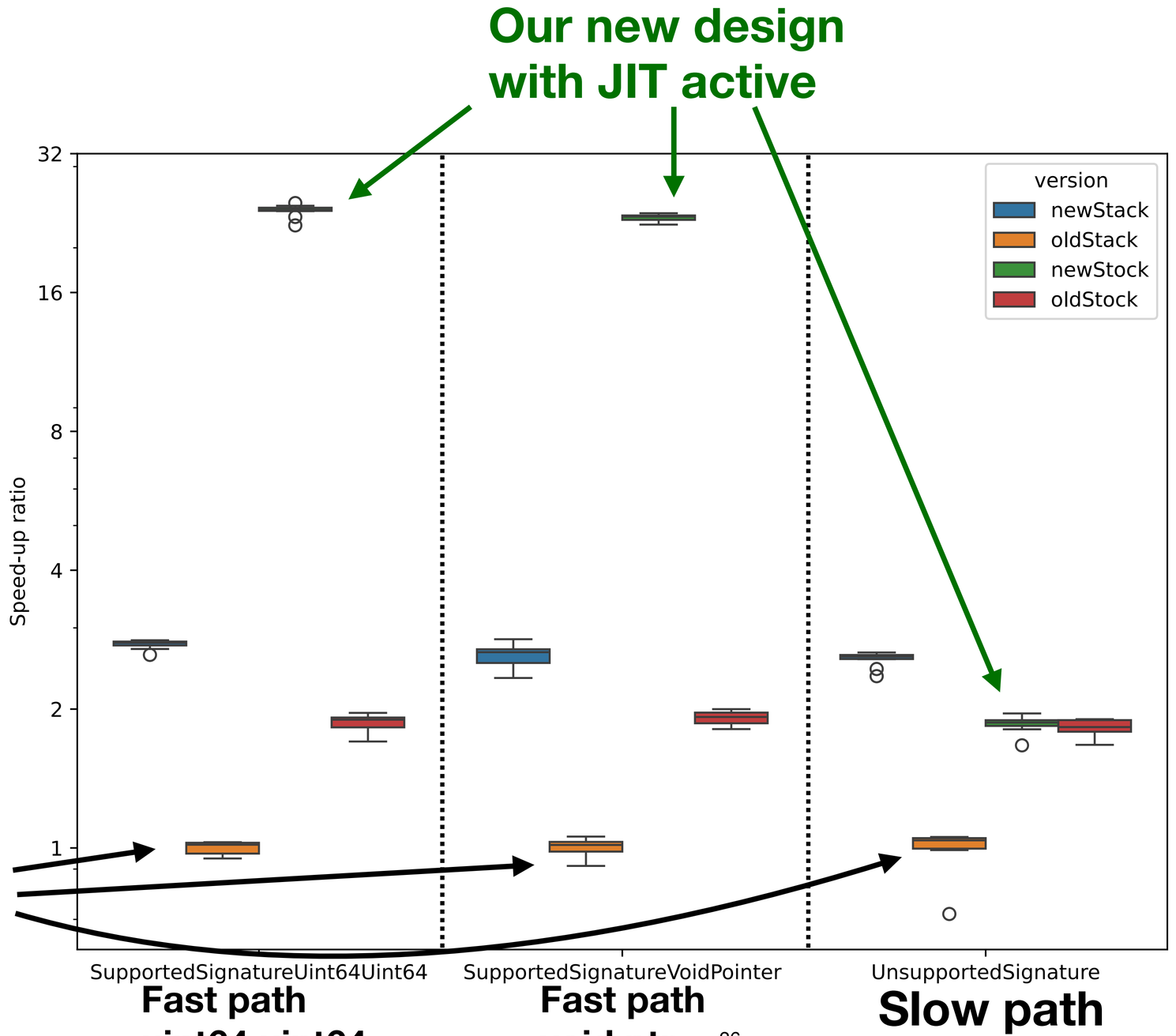


Roadmap

- FFI and why do we need it
- Current FFI implementation and its problems
- Our new design and how it solves those problems
- **Early results**

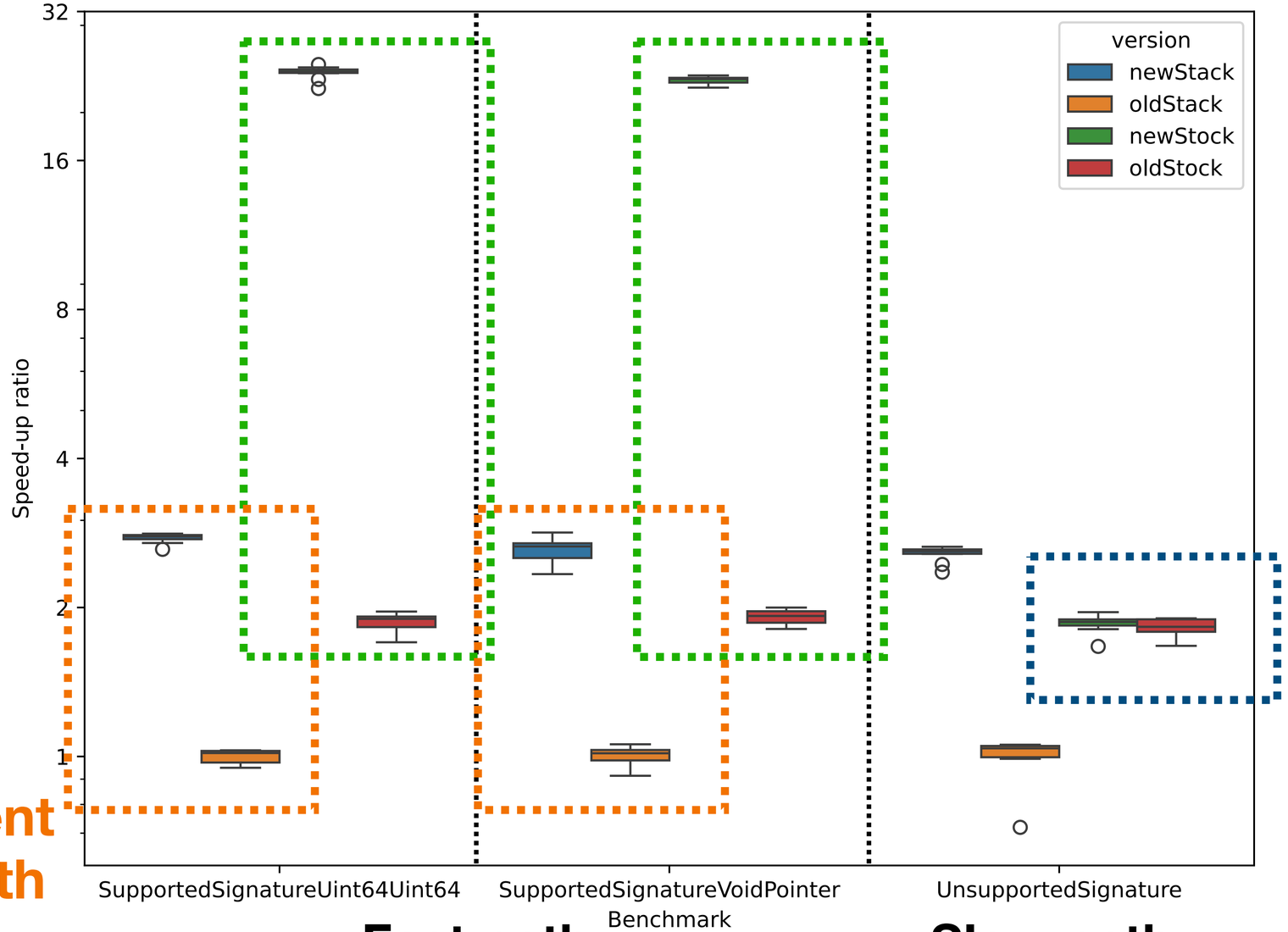
Results

higher is better



Results

12x improvement
on the fast path
with JIT active



No impact on
the slow path!

3x improvement
on the fast path
with no JIT

More in the paper

You can find a more detailed description of how it all works in the article

Redesigning FFI calls in Pharo: exploiting the baseline JIT for more performance and low maintenance

Bianchi Juan Ignacio[1], Polito Guillermo[1]

[1]*University of Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France*

Abstract

The Pharo programming environment heavily relies on a lot of different C functions. Such functionality is implemented through a Foreign Function Interface (FFI). Pharo implements FFI calls through a single primitive that implements all call cases. This generalization of behavior has performance drawbacks. In this paper, we present a new design for FFI calls. The key goal of the new design is to obtain better performance for the most used callout signatures while keeping maintenance low.

Keywords

Pharo, FFI, JIT

1. Introduction

The Pharo programming environment heavily relies on a lot of different C functions. Such functions are accessed through a Foreign Function Interface (FFI) that provides access to libraries respecting a common binary interface (ABI). Typically, those functions are written in the C programming language and compiled through a standard compiler such as GCC or Clang. For example, Pharo's IDE and graphical environment use libraries such as Cairo and SDL implemented in C. These graphics components are just one of the many users of C libraries that reside outside of Pharo.

As of today, all Pharo FFI calls are handled by a single primitive receiving as argument the signature

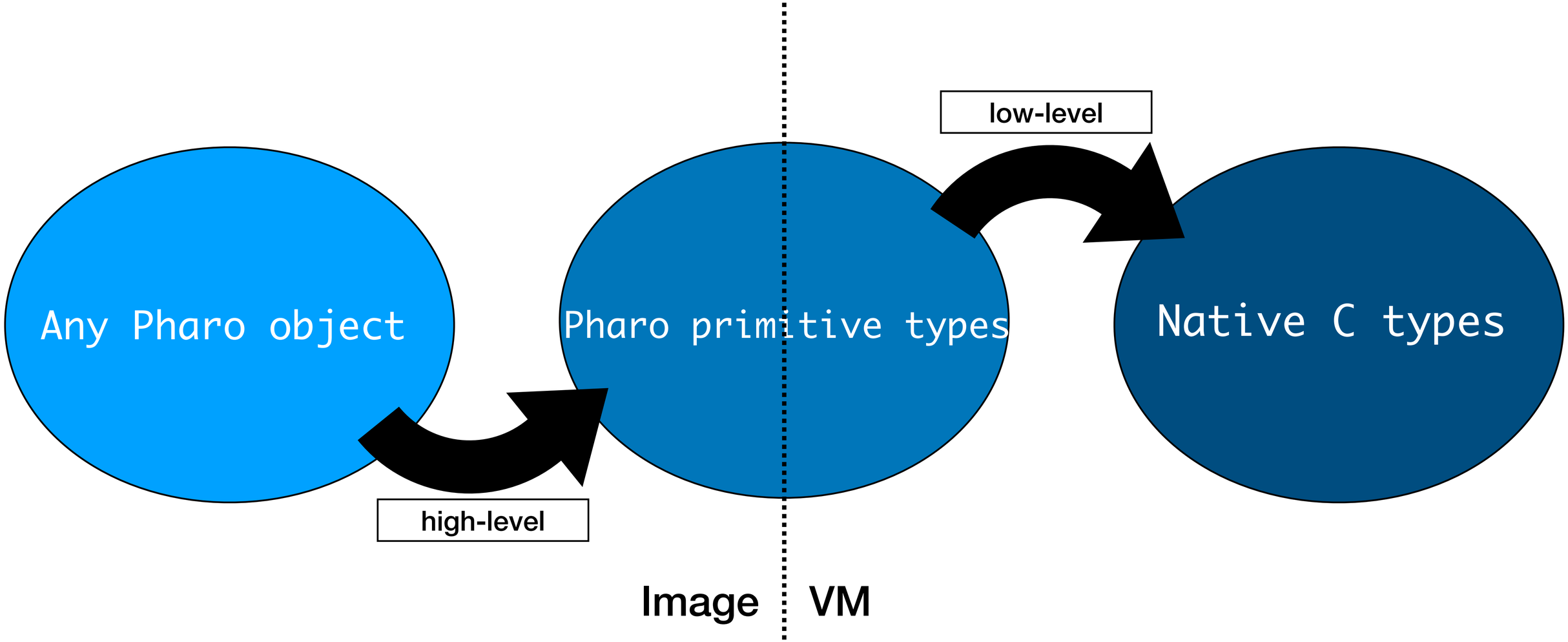
Conclusion

- Current primitive is too generic
- Introduced a new FFI call design for Pharo that is **faster** for the most commonly used function signatures
- Achieved up to **12x improvement** over the current implementation
- Slow path performs just like before

Extra - Marshaling

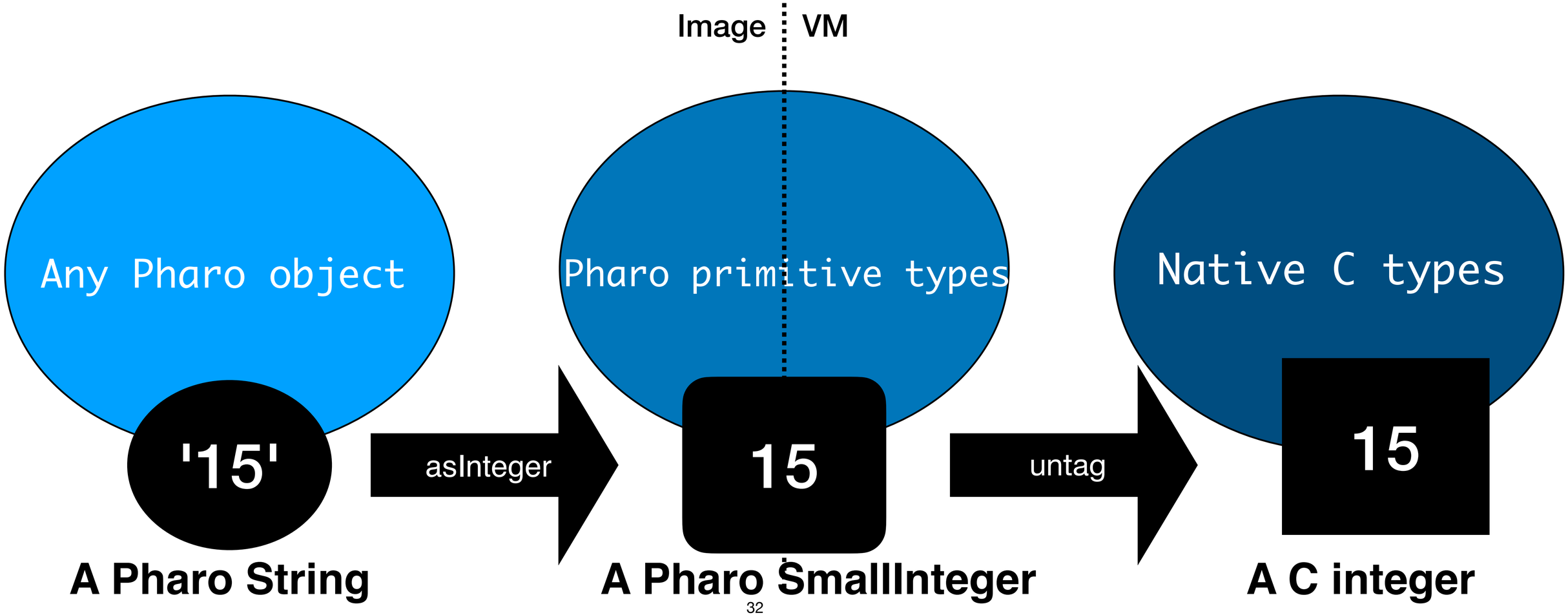
Two levels of marshaling

There is a high-level marshaling and a low-level one



Example

Consider a C function that takes an integer but from Pharo we call it with a String



Specializing marshaling at compile time

- The function meta-data will tell us how many and the type of the arguments
- We obtain them from the stack and convert them to their corresponding C native types
- For each type of value, the conversion ***Pharo type -> C type*** will be different

Specializing marshaling at compile time: Example

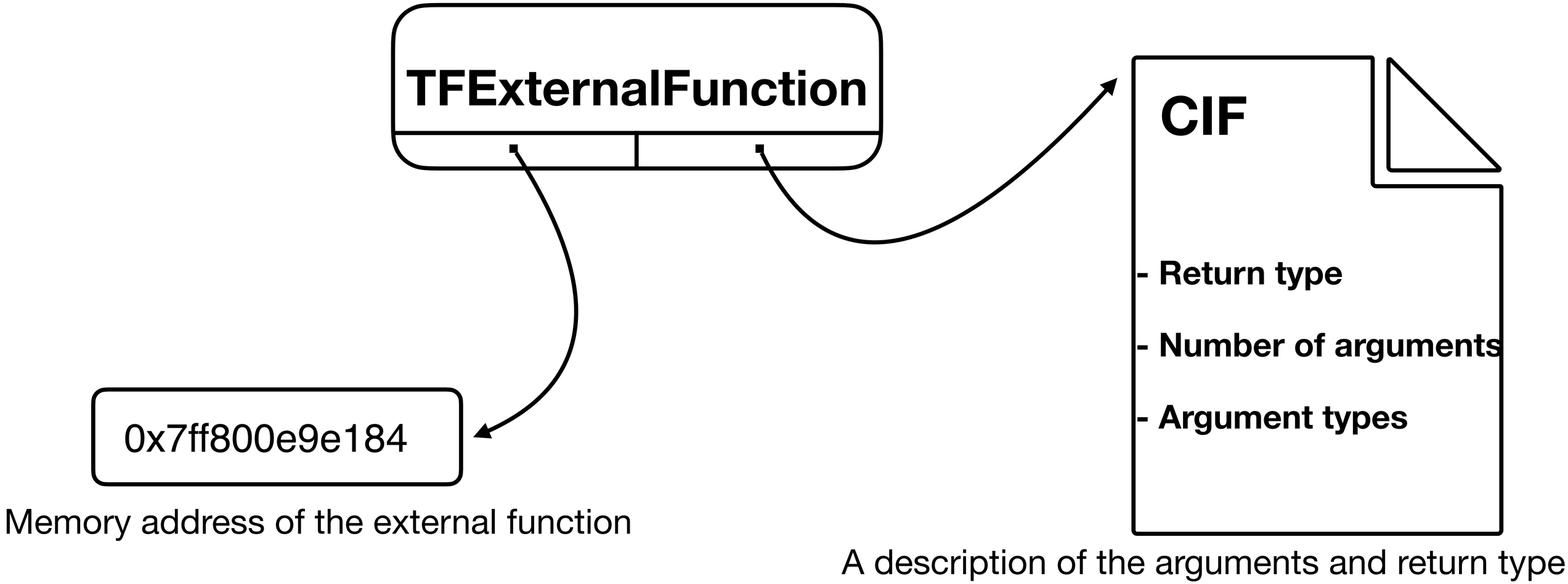
- The function meta-data tells us that the function has only an argument and its type is uint32_t
- So the machine code we generate would look like this:

```
jumpBadArg := objectRepresentation genJumpNotSmallInteger: RegisterForArg0.

objectRepresentation genConvertSmallIntegerToIntegerInReg: RegisterForArg0.

self CmpCq: 0 R: RegisterForArg0.

jumpBelowRep := self JumpLess: 0.

self CmpCq: UINT32_MAX R: RegisterForArg0.

jumpAboveRep := self JumpGreater: 0.

…
```

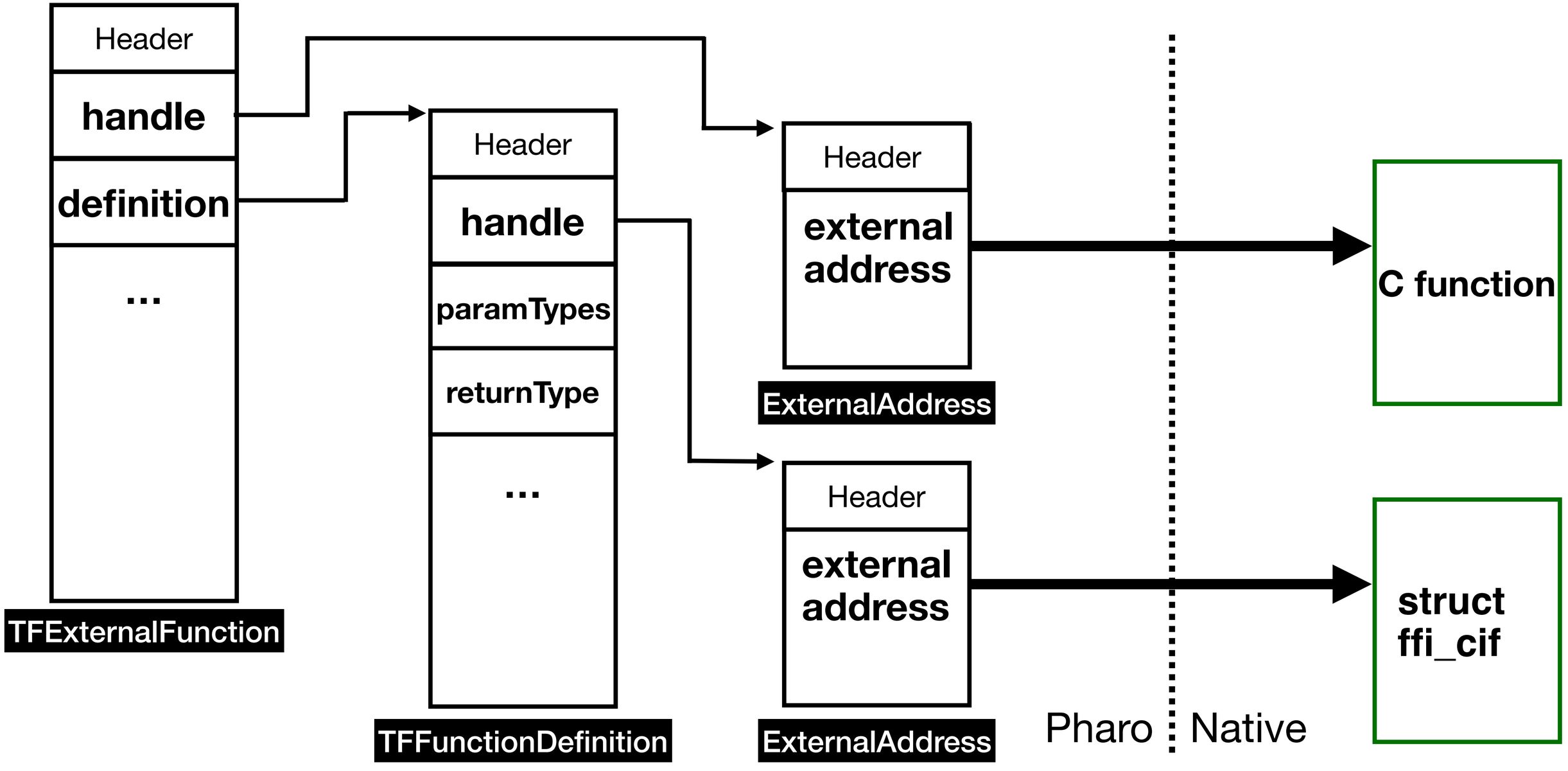
Extra - TFExternalFunction

TFExternalFunction

A description of the external function



Layout of a TFExternalFunction object



Extra - libffi

libffi

Pharo calls the `ffi_call` function defined by libffi

```
void ffi_call(ffi_cif *cif,
              void (*fn)(void);
              void *rvalue,
              void **avalue);
```

Pointer to the CIF

Pointer to the function to call

We have this information inside our TFExternalFunction object

The holder `rvalue` is where the external function given by the `fn` pointer will put the return value

The holder `avalue` is the way we pass the arguments to the external function

Holders for the arguments and return value

Essentially, it is a memory location we reserve beforehand for the external function to put something in there