NativeBoost

Igor Stasenko
May, 2012
Pharo Conf
What is NativeBoost?

- A plugin for VM which allows you to run machine code generated in image
- A set of utilities at language side which helping you to generate machine code and interact with VM
- It is more a philosophy than technology
A philosophy

• **ALL** interesting stuff should happen at language side
• No need to recompile VM each time you need to change something
• You should be able to ship your code in smalltalk. And it should work out of the box.
How does it work

- We’re extending a CompiledMethod trailer to carry a native code
- All native code is invoked via single primitive, provided by NativeBoost plugin: `#primitiveNativeCall`

```plaintext
someMethod: x y: y z: z
  <primitive: #primitiveNativeCall
    module: #NativeBoostPlugin>
...
```
Project components

- AsmJit - an assembler
- NativeBoost-Core - the core implementation
- NativeBoost-Unix/Mac/Win32 - a platform-specific support code
- Tests
- Examples
An x86 assembler as it is (just in smalltalk ;)

```
| asm |
asm := AJx86Assembler new.
asm
    push: asm EBP;
    mov: asm ESP -> asm EBP;
    mov: 1024 -> asm EAX;
    mov: asm EBP -> asm ESP;
    pop: asm EBP;
    ret;
    bytes.
```

```
asm
    mov: EAX ptr - 1 -> EAX;
    mov: EBX ptr + ECX * 2 - 5 -> EAX.
```

```
asm
    label: #label1;
    nop;
    nop;
    nop;
    jz: #label1.
```

+ x64 NOW!
NativeBoost-Core

• A top-level interface (NativeBoost class)
• VM interface (NBInterpreterProxy)
• FFI callout interface (NBFFICallout)
• C argument(s)/return type marshaling (NativeBoost-Core-Types)
• interface for generating native functions: NBNativeFunctionGen
NativeBoost interface

- contains code for bootstrapping a NativeBoost on target platform
- provides a default interface for external memory management (#alloc: / #free: )
- provides a default interface for loading external libraries and looking up their symbols
- subclasses taking care about platform-specific nuances
NBInterpreterProxy

• InterpreterProxy is a table of functions pointers - a public API of VM (sqVirtualMachine.h/.c)

• NBInterpreterProxy main purpose is interacting with VM: retrieving method’s arguments, accessing object’s state etc

• some VM functions may trigger GC, therefore we have a limitation: generated native code should be relocation agnostic
NBFFFI::Callout

- responsible for generating a machine code to make foreign calls
- support for different calling conventions (currently - cdecl and stdcall)
- provides a simple default interface for making foreign calls
First foreign call

man getenv ...

NAME
getenv, putenv, setenv, unsetenv -- environment variable functions

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdlib.h>

char *
getenv(const char *name);

RETURN VALUES
The getenv() function returns the value of the environment variable as a NUL-terminated string. If the variable name is not in the current environment, NULL is returned.
Calling getenv...

getEnv: name

<primitive: #primitiveNativeCall
module: #NativeBoostPlugin error: errorCode>

^ self nbCall: #( String getenv( String name)
) module: NativeBoost CLibrary
The magic

• initially, a compiled method is just a method with primitive

• on a first call a primitive fails, leading to entering a method body

• NBFFICallout then generating machine code, installs it into caller’s method and retry the message send

• machine code embedded into a method => its life cycle same as method where its installed
Forming a foreign call in detail

getEnv: name
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

^ NBFFICallout cdecl: #( 
   String getenv ( String name, ... )
)

module: NativeBoost CLibrary

cdecl - call convention
String - return type
getenv - function name
String - argument type
name - argument name

module - the module name or its handle, where to look for a function
Passing arguments

HeapAlloc Function

Allocates a block of memory from a heap. The allocated memory is not movable.

Syntax

```c
LPVOID WINAPI HeapAlloc(
    __in   HANDLE hHeap,
    __in   DWORD dwFlags,
    __in   SIZE_T dwBytes
);
```

- `dwFlags` [in]
  - `HEAP_GENERATE_EXCEPTIONS` 0x00000004
  - `HEAP_NO_SERIALIZE` 0x00000001
  - `HEAP_ZERO_MEMORY` 0x00000008

Naive approach

heapAlloc: aHeap flags: aFlags size: numberOfBytes

<primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ^ NBFFICallout stdcall: #(
LPVOID HeapAlloc (HANDLE aHeap, DWORD aFlags, SIZE_T numberOfBytes))
  module: #Kernel32

NBWin32Heap>>allocate: numBytes
^ self heapAlloc: heap flags: 0 size: numBytes

NBWin32Heap>>zalloc: numBytes
^ self heapAlloc: heap flags: HEAP_ZERO_MEMORY size: numBytes
Clever approach

```plaintext
NBWin32Heap>>alloc: numberOfBytes
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ NBFFICallout stdcall: #(

LPVOID HeapAlloc (self , 0 , SIZE_T numberOfBytes)

) module: #Kernel32
```

```plaintext
NBWin32Heap>>zalloc: numberOfBytes
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ NBFFICallout stdcall: #(

LPVOID HeapAlloc (self , HEAP_ZERO_MEMORY , SIZE_T numberOfBytes)

) module: #Kernel32
```
Types

• support for basic C types: int, float etc
• type aliases: map a name to one of the basic types
• C structures (see NBExternalStructure and subclasses)
Custom types

• subclass NBExternalType
• (demonstrate NBUTF8StringExample)
Getting rid of bloat

heapAlloc: aHeap flags: aFlags size: numberOfBytes

<primitive: #primitiveNativeCall module: #NativeBoostPlugin>

^ self call: #( .... )

It’s just a smalltalk code
Examples & Demo
Future plans

• integrate callback mechanism
• support for non-blocking call mode
• integration with JIT
The end