Lowcode: Extending Pharo with C Types to Improve Performance

Ronie Salgado
Universidad de Chile

Stéphane Ducasse
RMoD, INRIA, Lille

```
NativeStructure subclass: #WMVector3F
    layout: StructureLayout
    slots: { #x &=> #float.
             #y &=> #float.
             #z &=> #float.
             #padw &=> #float }
    classVariables: {  }
    category: 'WodenMath-Core-LinearAlgebra'
```

```
+ other
    <argument: #other type: #(SelfType object)>
    <returnType: #(SelfType object)>
    ^ self class x: x + other x y: y + other y z: z + other z
```

Dynamic languages vs static languages

- Performance gap between them
- Static languages are interpreted” by the CPU
- Cost of marshalling when doing FFI
- Runtime type checking for primitive operations
- Big cost when having to manipulate memory directly

What we did?

- Extending the VM with low-level bytecodes
- Type system with primitive types and object.
- Smalltalk compiler extended with primitive types, type inference and type checking
- Benchmarks
- Performance improvement between 50-400%

Lowcode: low-level bytecodes

- New (~270) low-level bytecodes
- Implemented as Sista “inline primitives”
- Operations with primitive data (int32, int64, float32, float64)
- Marshalling/unmarshlling
- Pointer load/store
- Local stack frame
- Native C function call

Specification and implementation

- Byte codes are specified formally in a XML
- Virtual machine implementation generated from the Spec
- Additional VM stack for native data that is not inspected by the GC
- Shadow Native callout stack in the interpreter (not in the jit)

Instruction specification


```xml
<instruction opcode="1037" mnemonic="float32Add">
    <name>Float32 addition</name>
    <description>
        This instruction performs the addition of two
        single precision floating point numbers.
    </description>
    <arguments />
    <stack-arguments>
        <float32 name="first" />
        <float32 name="second" />
    </stack-arguments>
    <stack-results>
        <float32 name="result" aliased="true" />
    </stack-results>
    <semantic language="Smalltalk/Cog">
        self AddRs: second Rs: first.
        self ssPushNativeRegisterSingleFloat: first.
    </semantic>
    <semantic language="Smalltalk/StackInterpreter">
        result := first + second.
    </semantic>
</instruction>
```

Extensible Type System

- Primitive types, primitive data references, pointers and object
- Type syntax based in Smalltalk syntax
- Types are parsed by sending a message
- Sending #asLowcodeType to an array or a Symbol

Types

Abstract Type	C Type	Lowcode Type	Size in Bytes
#void	void	NA	NA
#char	int8_t	int32	1
#short	int16_t	int32	2
#int	int32_t	int32	4
#long	int64_t	int64	8
#float	float	float32	4
#double	double	float64	8
#(void pointer)	void*	pointer	4 or 8
#(int pointer)	int*	pointer	4 or 8
#(float pointer)	float*	pointer	4 or 8
#object	NA	oop	NA

Agregate types

- Using Slots for defining structures and unions

```
NativeStructure subclass: #WMVector3F
    layout: StructureLayout
    slots: { #x &=> #float.
             #y &=> #float.
             #z &=> #float.
             #padw &=> #float }
    classVariables: {  }
    category: 'WodenMath-Core-LinearAlgebra'
```

Extending the compiler

- Extensions to the semantic analyzer:
  - Type annotations
  - Type checking
  - Local type inference
  - Special messages for type conversión
  - Trivial accessors and trivial constructors marked with a pragma are inlined

Trivial Accessors and constructors

```
x: value
   <accessor>
   <argument: #value type: #float>
   x := value

x: x y: y z: z
   <constructor>
   <argument: #(x y z) type: #float>
   ^ self new x: x; y: y; z: z; yourself
```

Lowcode Method Sample

```
normalized
   <var: #il type: #float>
   | il |
   il := ((x*x) + (y*y) + (z*z)) sqrt.
   il > 0.00001 asNativeFloat
      ifTrue: [ il := 1.0 asNativeFloat / il ]
      ifFalse: [ il := 0.0 asNativeFloat ].
   ^ self class x: x * il y: y * il z: z * il
```

Generated CompiledMethod

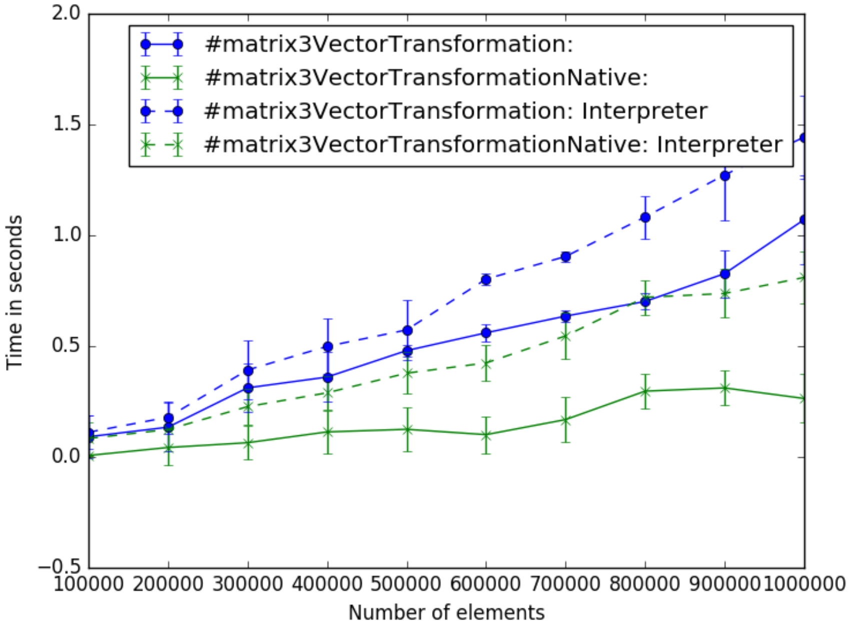
Raw | Pragmas | Bytecode | Source | Ir | AST | Header | Meta

```
label: 1
localFrameSize: 4
pushReceiver
firstFieldPointer
pointerAddConstantOffset: 0
loadFloat32FromMemory
pushReceiver
firstFieldPointer
pointerAddConstantOffset: 0
loadFloat32FromMemory
float32Mul
pushReceiver
firstFieldPointer
pointerAddConstantOffset: 4
loadFloat32FromMemory
pushReceiver
firstFieldPointer
pointerAddConstantOffset: 4
loadFloat32FromMemory
float32Mul
float32Add
pushReceiver
firstFieldPointer
pointerAddConstantOffset: 8
loadFloat32FromMemory
```

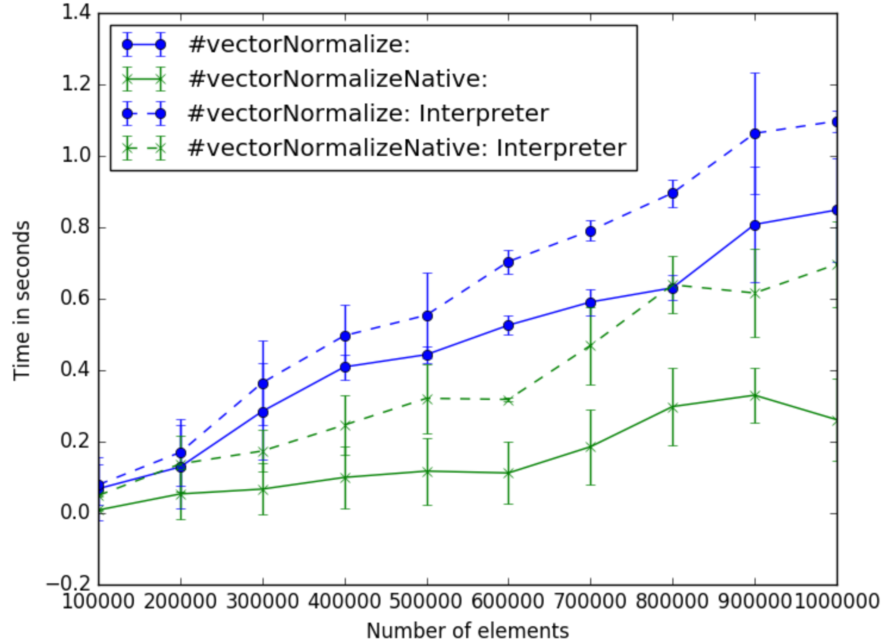
Benchmarks

- Executed with the JIT and the Interpreter VM
- Basic linear algebra operations used commonly in 3D graphics:
  - 3x3 matrix with matrix multiplication (2.96, 1.05)
  - 3x3 matrix with 3D vector multiplication (4.73, 1.63)
  - 3D vector normalization (3.82, 1.72)

Benchmarks



#matrix3VectorTransformation:
#matrix3VectorTransformationNative:
#matrix3VectorTransformation: Interpreter
#matrix3VectorTransformationNative: Interpreter
Time in seconds
Number of elements
2.0
1.5
1.0
0.5
0.0
−0.5
100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000



#vectorNormalize:
#vectorNormalizeNative:
#vectorNormalize: Interpreter
#vectorNormalizeNative: Interpreter
Time in seconds
Number of elements
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000

Conclusions

- No performance regressions in the Interpreter only VM
- Big performance improvement
- Not many changes are required to the Pharo methods

Future work

- Unchecked pointers and arrays
- More inlining (maybe working with Sista)
- Calling C functions directly avoing the FFI
- Making a C compiler

Thank you!