Méta-interprétation pour la génération de compilateur Just-In-Time

Nicolas Margulies

Méta-interprétation pour la génération de compilateur Just-In-Time

Nicolas Margulies
Encadrants : Guillermo Polito, Pablo Tesone et Stéphane Ducasse
Équipe RMOD de l'INRIA Lille – Nord Eurpope
Juillet-août 2020

Le contexte général

Beaucoup de langages modernes, dans un souci de portabilité du code, exportent des programmes compilés pour une machine virtuelle, lesquels seront interprétés pour les exécuter. Dans un souci d'efficacité, ces interpréteurs (appelés machines virtuelles) implémentent également un compilateur, dit Just-In-Time, qui compile à l'exécution les méthodes les plus utilisées. Le temps passé à compiler est compensé par le temps gagné à l'exécution (plus rapide) du code compilé.

Le problème étudié

Les machines virtuelles récentes sont donc divisées en deux parties : l'interpréteur et le compilateur Just-In-Time, l'un interprétant et l'autre compilant le bytecode pour l'architecture sur laquelle on l'exécute. La sémantique du langage est donc renseignée séparément dans les deux parties de la machine virtuelle. Nous tentons donc de supprimer cette redondance en ayant un ensemble interpréteur-compilateur JIT qui ne renseigne qu'une fois la sémantique du bytecode.

La contribution proposée

L'approche choisie ici est de générer le compilateur à partir de l'interpréteur. Plus précisément, nous sommes partis d'un interpréteur existant et avons construit un compilateur qui va interpréter le comportement de celui-ci (d'où le terme de méta-interprétation), c'est-à-dire compiler le comportement qu'aurait eu l'interpréteur sur le code donné en entrée. Pour faire cela, le compilateur récupère le code source de l'interpréteur pour l'instruction considérée et compile ce comportement vers du code machine. L'écriture d'un compilateur en entier dépassant de beaucoup le temps imparti, j'ai principalement travaillé sur la première passe de compilation du code de l'interpréteur vers notre représentation intermédiaire et la vérification de la bonne préservation de la sémantique lors de cette opération.

Les arguments en faveur de sa validité

Cette solution permettrait d'obtenir un compilateur en réécrivant le moins possible de code implémentant la sémantique du bytecode. Le compilateur possède en effet deux 'facettes' : l'une, indépendante de l'interpréteur, est de compiler le langage dans lequel l'interpréteur est écrit, tandis que l'autre consiste à réimplémenter certains comportements, notamment les données stockées par l'interpréteur lors de l'exécution du code, mais ceci concerne plutôt les détails d'implémentation de l'interpréteur que la sémantique du bytecode.

Le bilan et les perspectives

Comme dit plus haut, deux mois étant un temps court au vu de la quantité de travail pour produire un compilateur complètement fonctionnel, je me suis surtout concentré sur une étape, mais les résultats sont encourageants. La compilation vers la représentation intermédiaire marche bien et certains cas simples fonctionnent de bout en bout (du bytecode à l'exécution du code machine généré).

Il reste encore à gérer quelques cas particuliers pour la première étape, et surtout à coder la partie de la représentation intermédiaire vers le code machine, ceci comprenant une passe d'affectation de registres, puis la traduction vers le code machine. Enfin, l'approche devra être validée par des tests sémantiques et de performance.

Meta interpretation for Just-In-Time compiler generation

Nicolas Margulies
Under Guillermo Polito, Pablo Tesone and Stéphane Ducasse's guidance
RMOD team, INRIA Lille – Nord Europe

From 07/01/2020 to 08/25/2020

Abstract

A lot of modern programming languages like Java, C# or Pharo (the language used for this internship), instead of being directly compiled for the target architectures are instead compiled as code for a virtual machine (often called bytecode due to its compactness). This virtual machine code is then interpreted in order to run it. Nowadays, those interpreters (called virtual machines) also implement a Just-In-Time (JIT) compiler whose purpose is to compile frequently used methods to machine code to speed up execution (in those cases, the time taken to compile is compensated by the increased execution speed). This causes the semantics of the language to be duplicated in both parts of the virtual machine (the interpreter and the JIT compiler). The topic of this internship is to avoid this redundancy by using an abstract interpreter on the existing bytecode interpreter (a meta-interpreter) to generate a JIT compiler. This could allow to have a single runtime implementation and then generate a JIT compiler from it.

Contents

1 Context

1.1 The Pharo language

This internship was done on the case of the Pharo VM. Pharo is an object-oriented programming language based on smalltalk. The design principle was to have a language with as few language constructions as possible, so everything is done in terms of objects and messages, even loops and conditionals. The Pharo VM in itself is implemented in a subset of the Pharo language called Slang, thus when developping upon the VM in a 'simulated' way, I could write Pharo code and use all the high-level tools available to work on Pharo code (editor, tests, debugger). This internship was thus spent working with Pharo both as a language to compile, and the language in which we were writing the compiler.

1.2 Bytecodes and stack machines

Before being executed, Pharo methods are compiled as platform independent bytecode methods, and an array containing the literals used in that method. The bytecodes are virtual machine code instructions, while the literals are the fixed values written in that method. For example, in the following method, 1 and 17 are literals. literals can also be strings, literal arrays or characters.

```
MyClass >> foo
    ^ 1 + 17
```

We can inspect the method above `MyClass » foo` and see its bytecodes and literals. In the list below the first number is the bytecode index (the 'adresses' for referencing jumps), the second is the instruction bytes (what is actually stored), and then there is a mnemonic of the instruction and arguments (it explains what this instruction does).

```
25 <76> pushConstant: 1
26 <20> pushConstant: 17
27 <B0> send: +
28 <7C> returnTop
```

The Pharo bytecode is based on a stack machine (in contrast with register machines). This means that most of the data between operations is exchanged through a stack. In the example above, the first instruction pushes a 1 into the stack and the second instruction pushes a 17. Then, the third instruction has to send a message `+`, so it pops two elements from the stack: one to use as the receiver, and one to use as an argument. When the execution of the message send finishes, the result is pushed to the stack. Finally, the last bytecode takes the top of the stack (the result of the addition), and returns it to the caller.

1.3 The Pharo bytecode interpreter

When a bytecode method is executed by the interpreter, the interpreter iterates all bytecodes of the method and executes a VM routine for each of them. The class implementing the bytecode interpreter is `StackInterpreter` For example, the `pushConstantOneBytecode` is the routine that pushes a 1 to the stack using the `internalPush:` method. Since pushing the value 1 is a very common operation, a special bytecode is used for it to avoid putting the 1 in the literal frame.

```
StackInterpreter >> pushConstantOneBytecode

    self fetchNextBytecode.
    self internalPush: ConstOne.
```

The method `pushLiteralConstantBytecode` pushes a generic literal value to the stack also using `internalPush:`. The value pushed is taken from the literal frame of the method, and the index is calculated from manipulating the `currentBytecode` variable. Bytecode 33 pushes the first literal in the frame (`33 & 16 = 1`), bytecode 34 pushes the second literal, and so on... The actual method is longer, but I only kept the gist of it.

```
StackInterpreter >> pushLiteralConstantBytecode
    self pushLiteralConstant: (currentBytecode bitAnd: 16r1F).
    self fetchNextBytecode
```

Finally, bytecodes such as the message send `+` are implemented as follows. First this bytecode gets the top 2 values from the stack. Then it checks if boths are integers, and if the result is an integer, in which case it pushes the value and finishes. If they are not integers, it tries to add them as floats. If that fails, it will perform a (slow) message send using the `normalSend` method. All the calls to the `areIntegers:and:`, `integervalueOf:` and `integerObjectOf:` are due to the fact that objects in Pharo are either 63-bit integers or pointers to the object in the heap, and help converting an integer to it's pointer-like form and vice versa.

```
StackInterpreter >> bytecodePrimAdd
    | rcvr arg result |
    rcvr := self internalStackValue: 1.
    arg := self internalStackValue: 0.
    (objectMemory areIntegers: rcvr and: arg)
        ifTrue: [result := (objectMemory integerValueOf: rcvr)
                    + (objectMemory integerValueOf: arg).
                 (objectMemory isIntegerValue: result)
                 ifTrue: [self internalPop: 2
                          thenPush: (objectMemory integerObjectOf: result).
                    ^ self fetchNextBytecode "success"]
        ifFalse: [self initPrimCall.
                  self externalizeIPandSP.
                  self primitiveFloatAdd: rcvr toArg: arg.
                  self internalizeIPandSP.
                  self successful ifTrue: [^ self fetchNextBytecode "success"].
    messageSelector := self specialSelector: 0.
    argumentCount := 1.
    self normalSend
```

1.4 The existing Cogit JIT compiler

When a bytecode method is executed several times, the Pharo virtual machine decides to compile it to machine code. The process is pretty similar to the interpretation, the compiler iterates over the method and for each bytecode executes a code generation routine. This means that (almost) all the VM methods implementing bytecode interpretation will have their JIT counterpart. For example, the machine code generator implemented for `StackInterpreter » pushLiteralConstantBytecode` is `Cogit » genPushLiteralConstantBytecode`.

```
Cogit >> genPushLiteralConstantBytecode
    ^self genPushLiteralIndex: (byte0 bitAnd: 31)

StackToRegisterMappingCogit >> genPushLiteralIndex: literalIndex "<SmallInteger>"
    <inline: false>
    | literal |
    literal := self getLiteral: literalIndex.
    ^ self genPushLiteral: literal
```

The JIT'ted version of the addition bytecode (`genSpecialSelectorArithmetic`) is slightly more complicated, but it pretty much matches what it is done in the bytecode.

2 Structure of the Compiler

The compiling process we went with for this internship is the following : Starting from a bytecode method, the compiler iterates over the bytecodes, as the interpreter and the existing compiler do. For each bytecode, the compiler fetches the corresponding method of the interpreter (this is the meta-interpretation step). The method is obtained as an AST of Pharo code. The compiler proceeds then to translate this Pharo code to an intermediate representation. Then registers are allocated whithin the intermediate representation, before using another compiler interface to compile to machine code. In terms of Pharo classes, the code is organised as follows :

— The `DRASTInterpreter` class handles the interpretation of the Pharo AST and the conversion to the `DRInstruction` intermediate representation.

— The interpreter generates IR instructions through an `IRBuilder` which handles both this and the register allocation

— Finally the `DRIntermediateRepresentationToMachineCodeTranslator` class translates the IR to machine code.

3 The (meta-)interpreter

The meta-interpreter operates as follows : given a sequence of bytecodes, it extracts for each bytecode the `StackInterpreter` (or any interpreter that has a similar table bytecode/methods) method associated and generates intermediate representation mirroring that method's behaviour. The following code shows how to set up the interpreter.

```
builder := DRIRBuilder new.
builder isa: #X64.

astInterpreter := DRASTInterpreter new.
astInterpreter vmInterpreter: Druid new newBytecodeInterpreter.
astInterpreter irBuilder: builder.

astInterpreter interpretBytecode: #[76].
```

3.1 Visiting the AST

To interpret the AST, we used what is called the visitor design pattern. The principle is the following : instead of having a recursive method that checks the class of the current node and act accordingly, we use polymorphism and let the method lookup do the class check. Then, to avoid having to add code to the AST classes, those classes implement a generic visiting protocol that calls back the right method of the interpreter. More precisely, to visit a node, one calls `aNode acceptVisitor: astInterpreter`. The `acceptVisitor:` methods all look the same, here's the example for the `RBSelfNode` class (corresponding to writing `self`).

```
RBSelfNode >> acceptVisitor: aProgramNodeVisitor
    ^ aProgramNodeVisitor visitSelfNode: self
```

Then, the interpreter implements all the `visitXXXNode:` methods that handle the specific cases. Moste of them are simple, like the following ones :

```
DRASTInterpreter >> visitSelfNode: aRBSelfNode
    ^ self receiver

DRASTInterpreter >> visitTemporaryNode: aRBTemporaryNode
    ^ currentContext temporaryNamed: aRBTemporaryNode name
```

3.2 Context reification

As the interpreter follows message sends, we have to analyze the scope of temporary variables. In order to do that, we must reify the context, which means creating a stack of context objects referencing the name of temporary variables and arguments. Each time a method or block is activated, a new context is pushed onto the stack, which will be used to read and write temporary variables. The context is also used to store information about where to jump when we encounter non-local returns. When a method or block returns, the current context is popped to return to the caller context.

3.3 Interpreting Message sends

The most important operation in a Pharo program are message sends. Message sends are used not only for normal method invocations, but also for common operators such as additions (`+`) and multiplications (`*`) and control flow such as conditionals (`ifTrue:`) and loops (`whileTrue:`). However, some of these messages require special treatments when generating code. For example, a multiplication should directly generate a `DRMultiply` instruction and not require interpreting how that multiplication is implemented (especially since this can fail for primitives). Conditionals should generate jump instructions (especialy since looking up the implementation of those conditionals can't be done for comditions not known at compilation time).

To manage such special cases, we keep a table in the interpreter that maps (special selector -> special interpretation). When we find a message send in the AST, we lookup the selector in the table. If we find a special case in the table, we invoke that special entry in the interpreter. Otherwise, we lookup the method in the receiver's class and activate the new method, which will recursively continue the interpretation

```
DRASTInterpreter >> visitMessageNode: aRBMessageNode
    | arguments astToInterpret receiver |
    "First interpret the arguments to generate instructions for them. If this is
    a special selector, treat it specially with those arguments. Otherwise,
    lookup and interpret the called method propagating the arguments"
    receiver := aRBMessageNode receiver acceptVisitor: self.
    arguments := aRBMessageNode arguments collect: [ :e | e acceptVisitor: self ].
    specialSelectorTable
        at: aRBMessageNode selector
        ifPresent: [ :selfSelectorToInterpret |
            ^ self perform: selfSelectorToInterpret
                   with: aRBMessageNode
                   with: receiver
                   with: arguments ].
    astToInterpret := self
        lookupSelector: aRBMessageNode selector
        receiver: receiver
        isSuper: aRBMessageNode receiver isSuper.
    ^ self
        interpretAST: astToInterpret
        withReceiver: receiver
        withArguments: arguments.
```

For example, the following code shows how the `internalPush:` and `<` are mapped to their special behaviour, which is creating a push instruction or a comparison one in the intermediate representation.

```
DRASTInterpreter >> initialize
    super initialize.
    irBuilder := DRIRBuilder new.

    specialSelectorTable := Dictionary new.
    ...
    specialSelectorTable
        at: #internalPush:
        put: #interpretInternalPushOn:receiver:arguments:.
    specialSelectorTable at: #< put: #interpretLessOn:receiver:arguments:.
DRASTInterpreter >> interpretInternalPushOn: aRBMessageNode
        receiver: aStackInterpreterSimulatorLSB
        arguments: aCollection
    ^ irBuilder push: aCollection first.
DRASTInterpreter >> interpretLessOn: aRBMessageNode
        receiver: aReceiver
        arguments: arguments
    ^ irBuilder greater: aReceiver than: arguments first
```

The case of `+` is a bit longer because we add constant folding into the mix. This means that if the Pharo code is adding two constants (or values that are constants at compilation time), they are added in a new constant instead of generating an add instruction.

```
interpretSumOn: aRBMessageNode receiver: aReceiver arguments: arguments

    self assert: arguments size = 1.

    (aReceiver isConstantInteger and: [ arguments first isConstantInteger ])
        ifTrue: [ ^ irBuilder newConstant: aReceiver value + arguments first value ].

    ^ irBuilder add: aReceiver and: arguments first
```

Basically everything that should create IR instructions gets this treatment : arithmetic operations, comparisons, conditionals, loops, interacting with memory... There are also some special cases to make our AST interpreter mimic some of the `StackInterpreter`'s behaviour, and thus don't generate instructions. For instance, the `fetchNextBytecode` method makes the interpreter move to the next byte in the bytecode list.

```
DRASTInterpreter >> interpretFetchNextBytecodeOn: aMessageSendNode
        receiver: aReceiver
        arguments: arguments
    self fetchNextInstruction

DRASTInterpreter >> fetchNextInstruction
    currentBytecode := instructionStream next
```

4 The intermediate representation

The intermediate representation used here is a 'low-level' intermediate representation, meaning it's pretty close to machine code, but still allows optimizations. The IR instructions are subclasses of `DRInstruction`, and (for the most part) have between 0 and 2 arguments and a result field. I will thus represent them as their mnemonic, their arguments and an arrow pointing towards the result. For instance, `ADD 1 l1 -> l2` adds 1 to the `l1` register and puts the result in the `l2` register.

4.1 Operands : constants and registers

As shown in the example above, our IR instructions may take several operands, which are all elements of the `DRValue` class. There are three types of operands : constants, registers ans indirections. When displaying them, constants will just be represented as their value.

There are two kinds of registers : platform-independent logical registers, and platform-specific physical registers. Logical registers are indexed by integers. They will be represented with an `l`, followed by the index of the register. Physical registers represent the registers of the target architecture, and their list needs to be initialized before register allocation.

Finally, indirections are used to use a value as a memory address. As such, the indirection `(l1)` represents the memory value at the address `l1`, not the value assigned to the `l1` register.

4.2 Static Single Assignment (SSA) form and Phi instructions

Before register allocation, the IR is in SSA form, meaning that each 'variable' is only assigned once, and is assigned before being read, those 'variables' being our logical registers. This is done by creating a new logical register as the result of each instruction created, thus ensuring that they will only get assigned once. This form allows for much easier managment of the code (liveness analysis, dead code elimination, register allocation...), but needs the addition af a new instruction : phi instructions (`DRPhi`).

Consider the following code, and it's IR translation (missing the control flow parts and the end) :

```
SomeClass >> aMethod: anArgument
    | temp |
    (anArgument <= 0)                "CMP l0 0"
        ifTrue: [ temp := 0 ]        "MOV 0 -> l1"
        ifFalse: [ temp := 1 ].      "MOV 1 -> l2"
    ^ temp                           "???"
```

Here, the two assignments to `temp` will get assigned to different logical registers, but we don't know which one will be executed at compilation time, so we can't use either one as the return value. That's where Phi instructions are used. Before returning, a `DRPHI` instruction is insterted. They hold a collection of (other) instructions that should be 'unified' at this point. The ??? are thus replaced by `PHI (MOV 0 -> l1 MOV 1 -> l2) -> l3` and the `l3` register is returned. These Phi instructions will be resolved at register allocation time, either with a move instruction, or by assigning all three logical registers to the same physical one.

Another important (and tricky) use case of Phi instructions are non-local returns. When exiting a method, a Phi instruction regrouping all the return instructions within the method needs to be created : for instance if the previous method was written as follows, there would still need to be a Phi instruction insterted. The tricky part is to keep the correct semantics while avoiding to put the Phi instruction when unnessary.

```
SomeClass >> aMethod: anArgument
    (anArgument <= 0)        "CMP l0 0"
        ifTrue: [ ^ 0 ]      "MOV 0 -> l1"
        ifFalse: [ ^ 1 ].    "MOV 1 -> l2"
                             "PHI (MOV 0 -> l1 MOV 1 -> l2) -> l3"
```

4.3 Basic Blocks and control flow

When writing IR instructions, the builder doens't write them as a sequence, but rather whithin so-called basic blocks (the name is to differenciate them from Pharo blocks). Each block consists of a sequence of instructions that are always executed sequentially. These basic blocks are then both the nodes of the control flow graph of the method being compiled and the targets for jumps. We have two kind of jumps, standard jumps and conditional branches, which have a true and

false target for jumps (the choice is made according to the previous instruction, which must be a condition, e.g. a comparison), to allow for clearer control flow visualization and manipulation. Before generating machine code, this graph representation needs to be translated : jumps are given labels placed at the head of blocks for targets, conditionals are translated according to the target architecture and the code is written in a linear way (finally trivial jumps and labels are removed).

4.4 The role of the builder

As already mentionned above, the builder helps building IR instructions in many ways :

— It handles the storing of the instructions into basic blocks and creates debug information :

```
DRIRBuilder >> conditionalBranch
    ^ self appendInstruction: DRConditionalBranch new.

DRIRBuilder >> appendInstruction: anInstruction
    anInstruction astNode: currentNode.
    currentBasicBlock appendInstruction: anInstruction.
    ^ anInstruction.

DRIRBuilder >> pushNewBasicBlock
    currentBasicBlock := DRBasicBlock new.
    basicBlocks add: currentBasicBlock.
    ^ currentBasicBlock
```

— It maintains the SSA form on each instruction created by putting the result into a new logical register (this also means keeping track of the last logical register index)

```
DRIRBuilder >> add: aValue and: anotherValue
    ^ self appendInstruction: (DRAdd new
        leftOperand: aValue;
        rightOperand: anotherValue;
        result: self newLogicalRegister;
        yourself).

DRIRBuilder >> newLogicalRegister
    | aNewRegister |
    aNewRegister := DRLogicalRegister named: 'l' ,
        nextLogicalRegisterIndex printString.
    nextLogicalRegisterIndex := nextLogicalRegisterIndex + 1.
    ^ aNewRegister
```

The builder also handles register allocation, which means mapping the logical registers to physical registers. This also means checking that such mappings do not interfere with one another, resolving Phi instructions and, when the number of live variables exceed the number of physical registers, choosing which variables to 'spill' (store into the memory) and inserting corresponding moves. I did not however touch that part of the compiler.

5 The compiler interface

Once the interpreter finishes its job, the irBuilder will contain the intermediate representation instructions. We can then assign registers and generate machine code from them using the `DRIntermediateRepresentationToMachineCodeTranslator` class.

```
astInterpreter irBuilder assignPhysicalRegisters.
mcTranslator := DRIntermediateRepresentationToMachineCodeTranslator
    translate: builder instructions
    withCompiler: cogit.
```

```
address := cogit methodZone freeStart.
endAddress := mcTranslator generate.
```

`DRIntermediateRepresentationToMachineCodeTranslator` iterates all the instructions and uses the visitor pattern to generate code for each of them. It uses the current Pharo compiler as a backend to generate the actual machine code (it calls its code-generating routines).

```
DRIntermediateRepresentationToMachineCodeTranslator >> translate: aCollection
    aCollection do: [ :anIRInstruction | anIRInstruction accept: self ].

DRIntermediateRepresentationToMachineCodeTranslator >> visitPush:
    aPush value isConstant
        ifTrue: [ ^ self genPushConstant: aPush value value ].
    ^ self genPushRegister: aPush value name.

DRIntermediateRepresentationToMachineCodeTranslator >> genPushConstant: anInteger
    instructions add: (self gen: PushCw operand: anInteger).

DRIntermediateRepresentationToMachineCodeTranslator >> genPushRegister: aName
    instructions add: (self gen: PushR operand: (self translateRegister: aName)).

DRIntermediateRepresentationToMachineCodeTranslator >> gen: opcode operand: operandOne
    ^ cogit gen: opcode operand: operandOne
```

6 Conclusion

The topic of this internship was to work on creating a new kind of JIT compiler which uses the interpreter as a source for the semantics of the language instead of specifying them directly. After my contribution, the structure of the compiler is in place, with some basic examples working from the bytecode to the machine code. My contribution however was mainly focused on the meta-interpreter, which works quite fine, building a valid intermediate representation. Are still left some special cases covering arithmetic operations and conditionals that I didn't encounter, register allocation and some cases for machine code translation.

7 References

— Linear Scan Register Allocation for the Java HotSpot™ Client Compiler http://www.ssw.uni-linz.ac.at/Research/Papers/Wimmer04Master/

— Practical partial evaluation for high-performance dynamic language runtimes https://dl.acm.org/doi/10.1145/3062341

— Structure and Interpretation of Computer Programs http://web.mit.edu/alexmv/6.037/sicp.pdf

— Fun with Interpreters https://github.com/SquareBracketAssociates/Booklet-FunWithInterpreters/releases/download/ with-interpreters-wip.pdf

— Trace-Based Register Allocation (https://gitlab.inria.fr/RMOD/vm-papers/-/blob/master/compilation+JIT/2016_T based

— Paper explaining the motivations behind Sista Bytecode https://github.com/SquareBracketAssociates/Booklet-PharoVirtualMachine/raw/master/bib/iwst2014_A

— https://github.com/unicorn-engine/unicorn

— https://github.com/guillep/pharo-unicorn

— http://llvm.org/

— https://github.com/guillep/pharo-llvmDisassembler