The Opal Pharo Compiler

Steven Costiou, Stéphane Ducasse and Marcus Denker

April 9, 2020

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Opal overview

1.1 Roadmap (this subsection will be removed)

Big picture: mention everything => content overview of the rest of the document (small section about each other chap.). Give an overview of source code to bytecode compilation process.

Jorge said: I think that the first thing that we need is a description of the Opal model, with the different classes that take part in it. Then we can explain how the compilation and decompilation work. And also explain the intermediate representation. Finally, how to extend and change the compilation process.

1.2 Overview

Opal is the bytecode compiler for Pharo.

Opal is a flexible, configurable and adaptable. The Opal compilation process, is built around 3 steps, from source code to the bytecode (see Figure 1-1):

- Source code to abstract annotated syntax tree (see Fig. 1-2),
- Abstract syntax tree to intermediary representation,
- Intermediary representation to bytecode.

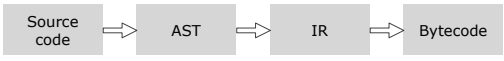

Figure 1-1 Opal complete compilation process.

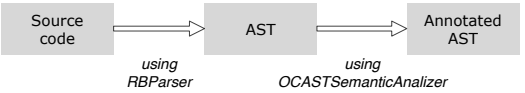

Figure 1-2 From source to annotated AST.

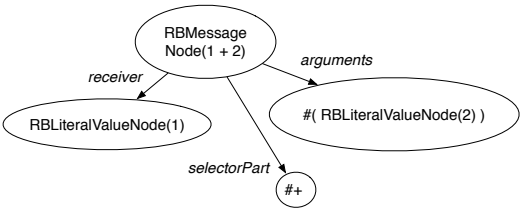

Figure 1-3 Generated tree for `'1 + 2'`.

This bytecode is transformed on the fly by the virtual machine in assembly. This last step is not the scope of OPAL.

1.3 From Source to Annotated AST

We explain the process described by Fig. 1-2.

AST

An Abstract Syntax Tree (AST) is a tree representation of the source code. The AST is easy to manipulate and scan it.

The ASTs used by Opal come from refactoring engine. It uses `RBParser` to generate ASTs, this step verifies the syntax. The structure of an AST is a simple tree.

Parsing an Expression

Using the message `parseExpression:`, we get an AST representing an expression (i.e., it means that you can only parse expressions and not methods which are not expression). Evaluate and inspect the following expression (See Figure 1-3).

```
t := RBParser parseExpression: '1 + 2'.
```

Let's try another example as shown in Figure 1-4, inspect the following expression:

```
RBParser parseExpression: 'one plus: two'.
```

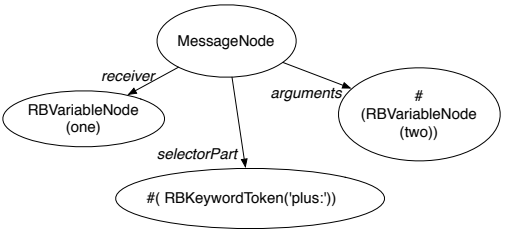

Figure 1-4 Generated tree for `'one plus: two'`.

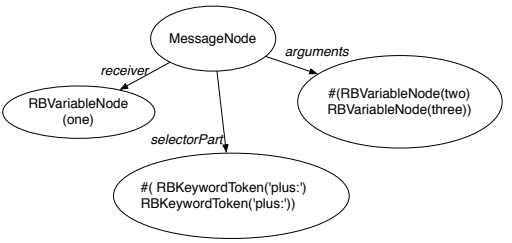

Figure 1-5 Generated tree for `'one plus: two plus: three'`.

Let's try a more complex example as shown in Figure 1-5, inspect the following expression:

```
RBParser parseExpression: 'one plus: two plus: three'.
```

Parsing a Method

You can also parse a the code of a method using the message `parseMethod:` instead of `parseExpression:`. We will get a `methodNode` object. The following code snippet produces a methodnode whose selector is `xPlusY` and the method body is a sequence of node containing statements. The first element is a `messageNode` with a receiver, a message selector and arguments.

```
RBParser parseMethod: 'xPlusY x + y'.
```

1.4 Annotating an Abstract Syntax Tree

Once parsed we can perform a semantic analysis of the AST. The goal of a semantic analysis is to add semantic data to the generated tree. One of the key function of the semantic analysis is to bind variables.

Because as we saw before the AST only checks the *syntax* of the code. A semantic analysis checks the semantics of the code: within the context of a

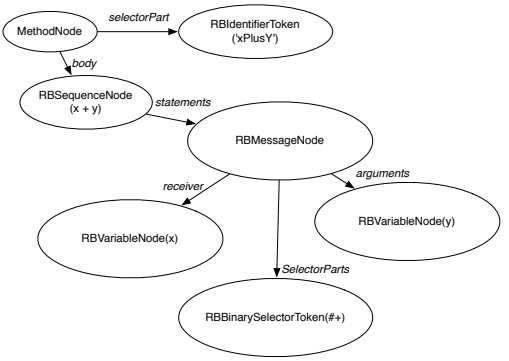

Figure 1-6 Generated tree for ==’Generated tree for ’xPlusY x + y’’==.

class we can check whether the code is valid. We can identify if a variable is undeclared or used out of scope.

The AST is annotated by visiting the graph with two visitors:

- `OCASTSemanticAnalyzer` performs the variable binding.
- `OCASTClosureAnalyzer` performs the closure analysis

TODO: explains what is variable binding. TODO: what is closure analysis.

Example

Let’s check with an example.

```
| ast |
ast := RBParser parseExpression: '1 + 2'.

"visit and annotated the AST for the closure analysis "
OCASTClosureAnalyzer new visitNode: ast.

"visit and annotated the AST for the var binding"
OCASTSemanticAnalyzer new
    scope: Object parseScope;
    visitNode: ast.
```

TODO: What do we get!!!!

All the data of binding is injected in the AST, when you inspect your AST you can see the value properties is now set to a dictionary. TODO: WHERE!!!!

1.5 Intermediate Representation

TODO: Please rewrite all that

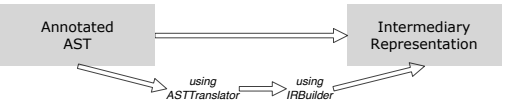

Figure 1-7 AST Annotated to Intermediary Representation.

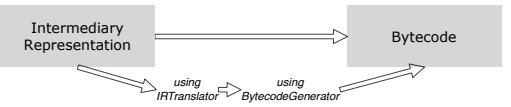

Figure 1-8 Intermediary Representation to Bytecode.

Once we obtain an AST annotated with semantic data, we can translate it into an intermediary representation (IR). The intermediary representation is a abstraction over bytecode structured in tree. The advantages of IR over bytecode is that it is higher-level. In addition, with IR we can plug different bytecode sets in the compilation process.

We could think about generating bytecode from a Pharo subset to LegoOS. In addition IR is easier to manipulate than bytecode itself. Usually the bytecode optimization will be realized after this step.

In many case you will not manipulate IR. You want to change jump, closure or push of the temp. It is a this level, if you want to indirect all the instance var access. It's at the level of the AST. we should simply rewrite this part, the IR is for :

- 1 different bytecode plug,
- 2 Bytecode optimization (easier to manipulate,more accurate, and it's more coherent),
- 3 small grain manipulation.

A Visitor walks an AST and builds its corresponding IR tree. `ASTTranslator` visits each node and for each node builds the corresponding IR node sequence. It uses the `IRBuilder` for this task. The `IRBuilder` offers all the infrastructure to add each possible node.

1.6 Bytecode

Once we have an IR tree, we will transform the IR tree in a bytecode sequence. We apply a new visitor but on the IR tree this time. Since IR is close to bytecode, the visitors visits each node and pushes the corresponding bytecode.

TODO: EXPLAIN BETTER we need a code snippet

1.7 Compilation?

Compiler Nodes

We should show Node Hierarhcy

Scope Analysis

I WANT TO UNDERSTAND SCOPE ANALYSE

Closure compilation

I WANT TO UNDERSTAND Closure compilation for REAL!

I WANT TO UNDERSTAND

cacheAST vs AST

I WANT TO UNDERSTAND

Decompilation

CHAPTER 2

The IR Builder

2.1 Roadmap

- Explain IR completely independently of everything else.
- How to use it:

- Overview how to compile IR -> bytecode. - How to decompile bytecode into IR - IRBuilder for Bytecode editing

CHAPTER 3

The Abstract Syntax Tree

3.1 Roadmap

- Explain RB, how to work with it, transforms, etc.
- Some words about source code to AST.

CHAPTER 4

Closure Compilation

4.1 Roadmap

Explains the closure model: (independent of the implementation)

- copying temps
- temps vectors, etc.

Start with the old one and then add the new blocks closure.

CHAPTER 5

Scope Analysis

5.1 **Roadmap??**

CHAPTER 6

Translations

6.1 Roadmap

Translation from AST to IR, which explains the ASTTranslator. Probably uses information from the AST and scope analysis chapter.

CHAPTER 7

Decompilation

7.1 Roadmap??

CHAPTER 8

Mapping

How does Pharo knows the source code interval corresponding to an executed bytecode?

8.1 What is the bytecode to AST mapping?

When executing code step by step in the debugger, the code going to be executed is highlighted so that we visually see where the execution is currently halted. Each time we step, the highlighted code is executed then the highlighted text is updated accordingly. This visualisation is available in methods and within blocks such as in the example below (Fig. 8-1).

The highlight indexes in the source code are encoded in the AST. Because at run time this is bytecode that is executed, tools need to recover the AST corresponding to the executed bytecode. To that end, those tools (e.g., the debugger) use a mapping that allows for recovering the AST corresponding to a particular bytecode.

But how is the AST recovered from the executed bytecode?

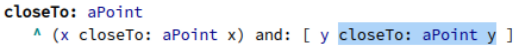

Figure 8-1 Stepping in a block: the next instruction that will execute is highlighted.

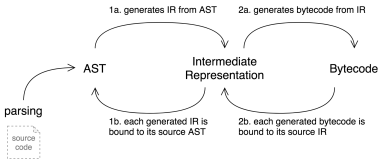

Figure 8-2 Stepping in a block: the next instruction that will execute is highlighted.

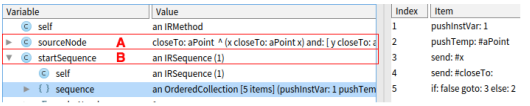

Figure 8-3 IR Mapping: source AST and IR sequence (instructions).

8.2 Mapping bytecode to AST: an overview of the mapping process

How generated bytecode is mapped to its corresponding AST is depicted in the overview below. Two mappings are stored by the intermediate representation (IR) of compiled code:

1. an IR <-> AST mapping
2. an IR -> bytecode mapping

The IR -> bytecode mapping is a sequence of bytecode corresponding to the IR. For instance for the IR of a method, the sequence is the bytecode of the method.

Fig 8-2 gives an overview of the mapping process at compile time. The first step is the parsing of the source code, which is transformed into an AST. Then every node of that tree is converted to an intermediate representation (1a). At each IR generation, the source AST from which it was just generated is mapped to that IR (1b). One AST therefore map to one IR and vice versa. Then each IR is converted to bytecode (2a). Similarly to the IR AST mapping, each generated bytecode is mapped by its source IR in an IR -> bytecode mapping stored by the IR (2b).

Fig. 8-4 shows an inspector opened on the IR of the closeTo: method from Fig. 8-1. That IR has a source node, which is the AST from which it was created (A). It also has a sequence of IR instructions (B).

Each IR instruction from the IR sequence has a bytecode index (Fig. 4). This index is the index of the last generated bytecode for that IR. Because many bytecode instructions can be generated for an IR instruction, the bytecode

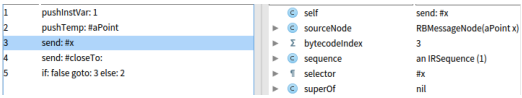

Figure 8-4 IR sequence instructions: each IR instruction knows the bytecode index of the last generated bytecode for that IR.

index is overwritten for each generated bytecode for that IR. Therefore, an IR instruction only references the index of the last bytecode it generated.

8.3 computing offsets

8.4 computing offsets with the sistav1 compiler and full block closures

8.5 Accessing the mapping at run time

Tools need to know what is the node that corresponds to an executed bytecode. This is the case for debuggers.

Tools endup calling the method `sourceNodeForPC:` that is defined as an interface for every compiled code (`CompiledMethod` and `CompiledBlock`) but that is actually implemented in the AST node representing the compiled code. Compiled methods will delegate this behavior to their method node and compiled blocks to their block node. In both case, the compiled code has to retrieve that node.

Let us have a look to the implementation in RBMethodNode:

```
RBMethodNode>>sourceNodeForPC: anInteger
  ^(self ir instructionForPC: anInteger) sourceNode
```

The ir is requested to provide the instruction that correspond to the program counter given as parameter. The implementation is shown below:

```
IRMethod>>instructionForPC: aPC
  | initialPC pc |
  "generates the compiledMethod and optimize the ir.
  Removes the side-effect of optimizing the IR while looking for
    instruction,
  which results in incorrect found instruction"
  initialPC := self compiledMethod initialPC.
  "For a given PC, the actual instruction may start N bytes ahead."
  pc := aPC.
  [ pc >= initialPC ] whileTrue: [
    (self firstInstructionMatching: [:ir | ir bytecodeOffset = pc ])
```

```
        ifNotNil: [:it |^it].
    pc := pc - 1 ].
  ^self "if we not found anything then this method is our target
    instruction"
```

Long story short: the instruction corresponding to the given PC is retrieved by enumerating all instructions and returning the first whose bytecode offset equals the PC.

But there are some subtleties.

The PC can be N bytes ahead of the current instruction (why?)

Because of that, we search the instruction by decrementing the PC. We first enumerate all instructions and look for one with a bytecode offset equal to the PC. If there are none, then we decrement the PC by 1 and search again over all the instructions.

If the decrementd PC ends up being lower than the initialPC, i.e. the bytecode offset of the first instruction of the method, then the instruction corresponding to the method is returned (the IRMethodinstance).

A method's bytecode offset does not start at o

A method starts at initialPC, that is never 0. It is computed in the compiled code object associated to the IR requesting the initial PC. As shown below, the value of the first PC of a method is the size in bytes of its literals.

```
CompiledCode>>initialPC
  "Answer the program counter for the receiver's first bytecode."

  ^ (self numLiterals + 1) * Smalltalk wordSize + 1
```

There are holes in the offsets

Because multiple bytecodes may refer to the same instruction, only the last computed offset is mapped. This means that a mapping can jump, e.g., from 46 to 48 with nothing in between. In the internal representation, offsets 46 and 47 actually correspond to the same instruction (i.e. to the same node), but only 46 is mapped to an instruction.

So, if we request the instruction for a program counter of 47, enumerating all instructions will give no result. Then as explained above, the PC is decremented and a new search is performed for offset 46. This time, an instruction will be found.

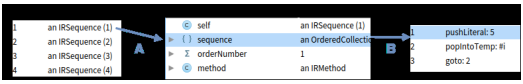

Figure 8-5 The first sequence of instructions of the sampleMethod IR.

How is computed the bytecode offset?

Each instruction has a bytecode index. This bytecode index is the delta between the bytecode offset of the first instruction in a compiled code and the instruction requesting its bytecode offset.

The bytecode offset is computed by adding the index to the initial PC minus one (why - 1?).

```
IRInstruction>>bytecodeOffset
  | startpc |
  startpc := self method compiledMethod initialPC.
  self bytecodeIndex ifNil: [^startpc].
  ^self bytecodeIndex + startpc - 1.
```

8.6 Overview of an IR method's structure

To give a general view of an IR method's structure, let us take a sample method that does nothing special.

```
sampleMethod
  | i |
  i := 5.
  [ i=0 ] whileFalse: [ i := i - 1 ].
```

In Figure 8-5, we see a list of 4 IR sequences on the far left of the picture. This is the start sequence of the sampleMethod IR (i.e., an IRMethod).

If we select the first sequence (A), we see the corresponding IRSequence instance: it has a number that is an integer representing its order in the sequence, and a sequence of IR instructions (B).

These instructions corresponds to the first statement the sampleMethod, plus an additional goto: instruction.

Let us explore these instructions. Figure 8-6 (A) shows the first IR instruction (pushLiteral: 5). We see its source node, that is the assignment in the code of sampleMethod, and its bytecode index used to compute the bytecode offset of the instruction.

Figure 8-6 (B) shows the goto: 2 instruction. This is a jump to the second sequence of the IRMethod instance, that we see in (C). These instructions represent the conditional in the while block of the second statement of sampleMethod.

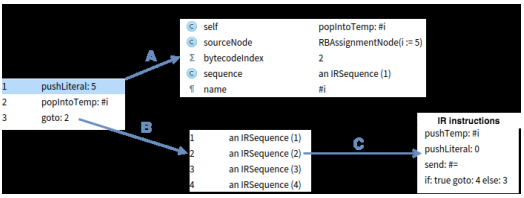

Figure 8-6 Navigating the IR instructions sequences of the sampleMethod IR.

The jump instruction is interesting, as it shows a conditional instruction that will jump either to the sequence of instructions that represents the body of the loop (the third sequence) or to the sequence of instructions which represents the next statement (the fourth sequence).

The following table shows the actual mapping between the bytecode offsets, the IRs and the nodes.

Why is goto:2 offset 41?

```
41  ->  pushLiteral: 5     ->  RBLiteralValueNode(5)
42  ->  popIntoTemp: #i    ->  RBAssignmentNode(i := 5)
41  ->  goto: 2            ->  RBMessageNode([ i = 0 ] whileFalse: [
    i := i - 1 ])
43  ->  pushTemp: #i       ->  RBTemporaryNode(i)
44  ->  pushLiteral: 0     ->  RBLiteralValueNode(0)
45  ->  send: #=           ->  RBMessageNode(i = 0)
46  ->  if: true goto: 4
                 else: 3   ->  RBMessageNode([ i = 0 ] whileFalse: [
    i := i - 1 ])
48  ->  pushTemp: #i       ->  RBTemporaryNode(i)
49  ->  pushLiteral: 1     ->  RBLiteralValueNode(1)
50  ->  send: #-           ->  RBMessageNode(i - 1)
51  ->  popIntoTemp: #i    ->  RBAssignmentNode(i := i - 1)
52  ->  goto: 2            ->  RBMessageNode([ i = 0 ] whileFalse: [
    i := i - 1 ])
54  ->  returnReceiver     ->  helperMethod12  | i |
                                               i := 5.
                                               [ i = 0 ] whileFalse:
    [ i := i - 1 ]
```

CHAPTER 9

Hidden Corners

9.1 Roadmap

Special stuff and super technical hidden stuff:

- Doit compilation
- Extra Bindings
- Optimization

- inlining - must be boolean explanation

- Compiler plugins
- Compiler options
- Compilation Context
- Environments / ProductionEnvironment

9.2 Special objects

Some messages such as the ones of Boolean are not actually sent. It is then impossible to redefine such messages (which ones). This reason is that the compiler optimize them by inlining them. What is the problem

9.3 Supporting Inlined Messages

There is an Opal modular extension to make sure that inlined messages can still be executed. (more explanation needed).

```
WHICH CLASS >> mustBeBoolean
  "Catches attempts to test truth of non-Booleans.  This message is
    sent from the VM.  The sending context is rewound to just before
    the jump causing this exception."
  ^ Boolean mustBeBooleanDeOptimize
    ifTrue: [ self mustBeBooleanDeOptimizeIn: thisContext sender  ]
    ifFalse: [ self mustBeBooleanIn: thisContext sender ]
```

To enable it, you can just execute

```
Boolean mustBeBooleanDeOptimize: false.
```

and it is turned off.

Note that `mustBeBooleanDeOptimizeIn:` is part of OpalCompiler, not the Kernel, so it does not take space there (the only thing in the Kernel is the switch to turn it on or off)

In Opal, it is just this one method:

```
WHICH CLASS >> mustBeBooleanDeOptimizeIn: context
  "Permits to redefine methods inlined by compiler.
  Take the ast node corresponding to the mustBeBoolean error,
    compile it on
  the fly and executes it as a DoIt. Then resume the execution of
    the context."

  | sendNode methodNode method ret |

  "get the message send node that triggered mustBeBoolean"
  sendNode := context sourceNode sourceNodeForPC: context pc - 1.
  "Rewrite non-local returns to return to the correct context from
    send"
  RBParseTreeRewriter new
    replace: '^ ``@value' with: 'ThisContext home return: ``@value';
    executeTree: sendNode.
  "Build doit node to perform send unoptimized"
  methodNode := sendNode copy asDoitForContext: context.
  "Keep same compilation context as the sender node's"
  methodNode compilationContext: sendNode methodNode
    compilationContext copy.
  "Disable inlining so the message send will be unoptimized"
  methodNode compilationContext compilerOptions: #(- optionInlineIf
    optionInlineAndOr optionInlineWhile).
  "Generate the method"
  OCASTSemanticCleaner clean: methodNode.
  method := methodNode generate.
  "Execute the generated method with the pc still at the optimzized
    block so that the lookUp can read variables defined in the
    optimized block"
  ret := context receiver withArgs: {context} executeMethod: method.
    "resume the context at the instruction following the send when
```

```
    returning from deoptimized code."
       context pc: sendNode irInstruction
    nextBytecodeOffsetAfterJump.
  ^ret.
```