# Conditional Breakpoints on Top of a Reflective Layer

Thomas Dupriez
ENS Paris-Saclay / Software Language Lab, Vreije Universiteit Brussels
thomas.dupriez@ens-paris-saclay.fr

## Abstract

***Keywords*** Debugger, Tool, Stack, Breakpoint

Conditional breakpoints are a very common tool used by developers to acquire knowledge about the execution of a program. Often an IDE feature, the developers place them at strategic points of a program and specify conditions for them. When running the program, the execution will be stopped if it reaches a conditional breakpoint whose condition is fulfilled.

In this paper we focus on a way of implementing these conditional breakpoints: by using the reflective layer of the language itself.

How Truffle+Ruby implements it: [SVDVH14] - Ruby's core library implements the set_trace_func function, allowing tools to register a method to be called each time the interpreter encounters certain events, such as moving to a new line of code, entering or leaving a method or raising an exception. - inactive wrapper nodes are introduced in the AST at all positions where the user may place a breakpoint (every line). These nodes only check if a method is registered for their line of code. If there is one, they swap themselves with an active node that can contain arbitrary java (because that's the host language of Graal-Truffle) code (a sub ast tree), for example stopping the execution and opening a debugger. Active nodes also check whether a method is registered for their line of code. If there is not, they swap themselves back to an inactive node. The arbitrary code can contain an *if* statement only opening a debugger if a given condition is met.

What's interesting is that since this debugging code is added on the program's AST, it is compiled and optimised by the compiler as if it was part of the method's code.

How we implemented it: - Using metalink [Den08] to add, in the bytecode of the method, a message send to a custom object - That custom object has one method, which checks the condition of the breakpoint and, if it is satisfied, stops the execution and opens a debugger - However, this method doesn't execute in the same scope as the debugged method, so the condition cannot use variables, self, super or thisContext (or rather, they will not have the value they have in the debugged method) - To remedy that, we used a metalink feature to forward the value of thisContex in the debugged method to the custom object. We then use various ast manipulations in the condition [describe them] to replace references to variables, self, etc with message sends to the forwarded thisContext, to get the values these would have had if the condition was executed in the debugged method's scope.

Another option we considered was [read emails]

## 1. Introduction

**Contextual Breakpoints.** What if a developer had the potential to create more powerful and adaptable breakpoints? What would such breakpoints look like? What kind of runtime information should they have access to to be both useful and efficient? their objects? How can we navigate the

## 2. Debugging Terminology

**Debugging** *i.e.,* In Pharo, a program entry point is the first context of a Process, usually executing the Block>>#newProcess method.

## 3. A Real Complex Debugging Scenario

[1].

```
$ ./pharo Pharo.image test "Pillar.*"
[...]
3182 run, 3182 passes, 0 failures, 0 errors.
```

Checking the tests, the pillar developers observe that the bug happens in an apparently unrelated piece of code, the PREPubMenuJustHeaderTransformer>>actionOn: method.

```
PREPubMenuJustHeaderTransformer>>actionOn: anInput
  ^ (self class writers
    includes: anInput configuration outputType writerName)
    ifTrue: [ maxHeader := self maxHeaderOf: anInput input.
```

[1] We setup a repository explaining in details the steps to reproduce the bug in https://github.com/guillep/pillar-bug

```
    super actionOn: anInput ]
  ifFalse: [ anInput ]
```

*conditional breakpoints* by attaching arbitrary boolean expressions to breakpoints, yielding breakpoints that only trigger if the attached expression evaluates to *true*.

```
self haltIf: [ CurrentExecutionEnvironment value
      isTestEnvironment ].
```

Figure 1: Breakpoint that only triggers while tests are run. In Pharo

## References


[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[SVDVH14] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 2:1–2:13, New York, NY, USA, 2014. ACM.