Multi-level Debugging for Interpreter Developers

Bastian Kruck* Stefan Lehmann† Christoph Keßler*
Jakob Reschke* Tim Felgentreff† Jens Lincke† Robert Hirschfeld†
* {firstname.lastname}@student.hpi.de
† {firstname.lastname}@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Abstract

Conventional debuggers require programmers to work on multiple levels of abstraction at once when inspecting call stacks or data. This demands considerable cognitive overhead and deep system knowledge of all implementation technologies involved. When developing an interpreter, programmers often create a dedicated debugger to have a higher-level perspective on the client-language; the resulting use of multiple debuggers at once leads to mental context switches and needs an elaborated method.

We present an integrated debugging tool in which interpreter developers define and select the levels of abstraction on which they focus. Our debugger provides them with an abstraction-specialized view. We consider both *host-language* and *guest-language levels*, since either may be levels of interest in a debugging session. We show how this separation into *host-language levels* can ease the debugging of applications through filtering call stacks and specializing call stack representation on levels.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors–Debuggers, Interpreters; D.2.5 [**Software Engineering**]: Testing and Debugging–Debugging aids, Diagnostics;

Keywords Language Workbenches; Squeak; IDE; DSL; Inter-language debugging; Language-oriented programming

1. Introduction

A language-specific and still feature-rich tooling is essential when building and using a language [1]. While modern IDEs provide a number of desirable features, those features still need to be implemented for a particular language. This might be a time-consuming and complex challenge [6, 8]. Language workbenches [4, 7] are intended to help DSL developers use modern IDE functionality without much effort. However, these projects mainly focus on compiling DSLs statically and do not support creating interpreters.

When debugging an interpreter, developers need to reason about the interpreter's state based on the encoded host-language representation. All of the guest-level program's data, call stack and static information are wrapped into internal representations of the host VM. As a result, developers need to have knowledge of both the guest and the host language to understand the interpreter's behavior and isolate the root causes of errors [9]. Additionally, the developer needs to understand the executed guest program to trace its faulty behavior at the interpreter level.

Modularity '16, March 14 - 18, 2016, Málaga, Spain
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4033-5/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2892664.2

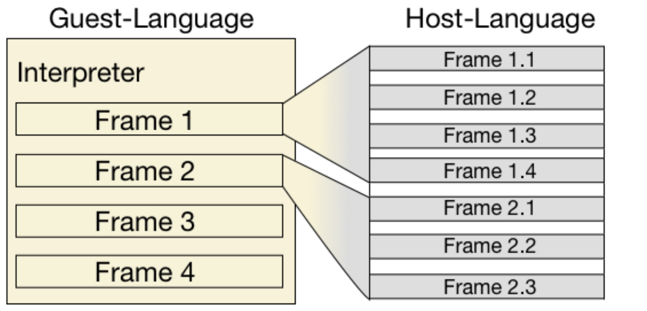


Figure 1. Debugging an interpreter requires interaction in the context of the guest as well as the host language.

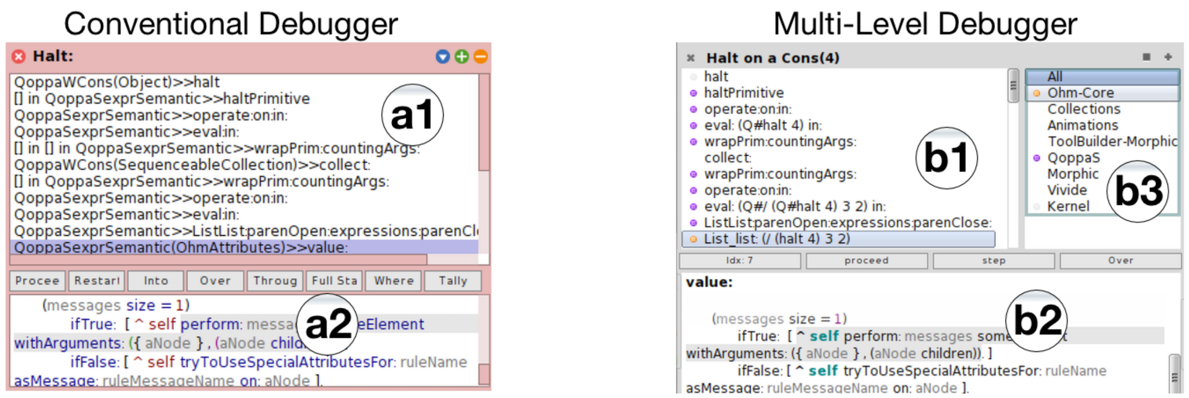
Conventional debuggers show the guest-language state encoded in the host-language level representation only [7]. Most debuggers provide a one-dimensional list of stack-frames to navigate through the active call chain. As exemplified in Figure 1, an interpreter contains another execution state itself that corresponds to the host-language stack in a one-to-many relationship. Interpreter-Programmers pay high cognitive overhead to manually map those two execution environments onto each other. Furthermore, they are overwhelmed with host-level information while trying to step through the user program only. This cognitive effort prevents gaining an overview and requires developers to bind their analytical resources to the decoding instead of the problem domain.

To reduce the number of manually followed abstractions, some developers of interpreters open an additional debugger for the guest language [5]. They can switch back and forth between the debuggers to compare the VM state with the state of the user program and step through either language to observe the state changes in the other one. Nonetheless, this requires them to do context switches and manage two executions at once.

We present a debugging tool in which users define and select the language level that they focus on. Explicitly selecting debugged abstractions enables our user to reduce the cognitive overhead. They only see level-related stack-frames and scope items and may expand foreign level information on demand. This solution has the benefits, that (i) we deliver only the information needed and (ii) display them in a way that is desired for the current use-case.

In this paper, we present the concept of a debugger, that facilitates explicit debugging perspectives to ease the debugging process. Our contributions are:

- The concept of an interpreter-aware debugger that facilitates a merge-and-filter strategy and context-dependent callstack views
- A prototypical implementation of our concept in Squeak/Smalltalk based on Ohm/S



Halt:

```
QoppaWCons(Object)>>halt
[] in QoppaSexprSemantic>>haltPrimitive
QoppaSexprSemantic>>operate:on:in:
QoppaSexprSemantic>>eval:in:
[] in [] in QoppaSexprSemantic>>wrapPrim:countingArgs:
QoppaWCons(SequenceableCollection)>>collect:
[] in QoppaSexprSemantic>>wrapPrim:countingArgs:
QoppaSexprSemantic>>operate:on:in:
QoppaSexprSemantic>>eval:in:
QoppaSexprSemantic>>ListList:parenOpen:expressions:parenCl
QoppaSexprSemantic(OhmAttributes)>>value:
```

```
    (messages size = 1)
        ifTrue:  [ ^ self perform: messa          eElement
withArguments: ({ aNode } , (aNode child
        ifFalse: [ ^ self tryToUseSpecialAttributesFor: ruleName
asMessage: ruleMessageName on: aNode ].
```

Halt on a Cons(4)

```
halt
haltPrimitive
operate:on:in:
eval: (Q#halt 4) in:
wrapPrim:countingArgs:
collect:
wrapPrim:countingArgs:
operate:on:in:
eval: (Q#/ (Q#halt 4) 3 2) in:
ListList:parenOpen:expressions:parenClose:
List_list: (/ (halt 4) 3 2)
```

value:

```
    (messages size = 1)
        ifTrue:  [ ^ self perform: messages some
withArguments: ({ aNode } , (aNode children)). ]
        ifFalse: [ ^ self tryToUseSpecialAttributesFor: ruleName
asMessage: ruleMessageName on: aNode ].
```

Figure 2. Our debugger adds a level-selection step (b3) to the debugging process in addition to the stack frame list (a1, b1) and the source code view (a2, b2). Value:- and eval:in:-stack frames are characterized with help of call parameters (b1).

2. Dynamic Multi-level Debugging

Our key concept is to use levels of abstraction as interaction units to allow developers to see the system from the perspective they have in mind.

When speaking of levels, we refer to both *(i) host-language levels* and *(ii) guest-language levels*, as either form a mental unit that the user works within or explicitly wants to switch between. *Host-language levels* are levels of abstraction or working units, like packages, libraries or internal DSLs. *Guest-language* levels evolve when building an interpreter in the debugged language; they form a new level of interest as well. In our system, the levels are described in two perspectives: (1) By default, there is a grouping for each module (e.g. a package in Squeak Smalltalk), (2) developers may use extension points to add domain specific levels.

A level-aware debugger enriches the debugging experience of any type of applications that have multiple libraries or frameworks. The developer of a library can use his debugging experience to modify the presentation of his code in the debugger. Having done that, the developer and others do not need to apply (nor build) the knowledge on non-self-explanatory errors and frames anymore.

2.1 Merge-and-Filter the Call Stacks of Levels

A multi-level debugger *merges and filters* the call stacks of levels of abstraction. While conventional debuggers already show the unstructured call stacks where all levels are interleaved, they only provide minimal filtering facilities. We add the following mechanisms:

Map stack frames to their corresponding level. In our case, we use colored bullet points to enable programmers identify a level of interest based on the trace that they see.

Filter stack frames. A list of levels serves as key to that mapping and a user can select one or more to filter the stack frames such that only those of some particular levels are retained. This allows him order to inspect the behavior on this level only.

Step inside levels. Stepping should proceed the program until returning to a listed stack frame. Others frames are executed but not inspected as they are not of interest right now.

Unfold foreign levels. Developers should see when there are frames filtered away between two frames and be able to partially unfold this very fold only within the same level. This enables tracking down an error to lower levels.

2.2 Specialized Representation for Each Level

A multi-level debugger should represent program state specialized on different levels to be able to represent the programmers mental model for each level. We identified these two specializations:

Specialize stack list representation of stack frames. The stack trace is usually presented as a list and the method activations need to have a characteristic representation there. It is usual to show the name of the called method and its class. In log files, it is common sense to annotate entries with the source file and the line number. In frameworks and recursive programming, we observe the same code location appearing many times in the call chain. Thus, these representations should be enhanced to add distinction, e.g. by including arguments of a call.

Specialize inspection views of a stack frame. Projectional editors allow to view different representations of the same static code, e.g. a table or a diagram for a state machine. Often, it is possible to create a higher-level representation of raw code so that we can reason about it at the level at which the concept is usually represented.

2.3 Extensibility to Add Levels

Group frames to a level. A multi-level debugger should allow creating and modifying levels to grow with the system. When introducing a new level, the extension needs to determine if a given stack frame is contained in it (See 2.1), and needs to provide representations for the contained stack frames (See 2.2).

3. State of Implementation

Figure 2 shows the prototype of our debugger next to a traditional one. We have built a debugger that contains a level-selection-view (b3) in addition to the known stack-frame view (a1, b1) and a source code view (a2, b2).

3.1 Technical Background

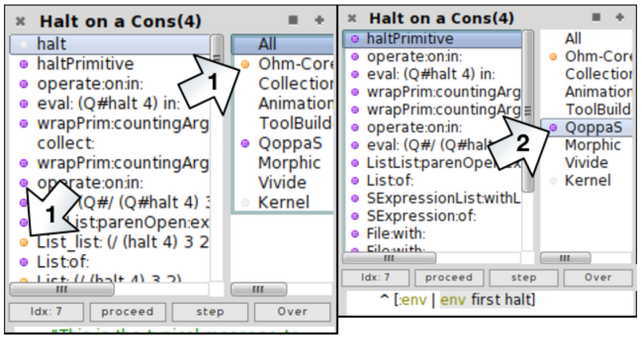
We created our system in Squeak Smalltalk[1] with the data exploration framework VIVIDE. We implemented an interpreter of a Scheme dialect without special forms (called Qoppa[2]) with help of the Ohm/S parser framework, aiming to specialize a debugger on *inter-language debugging*. The facilities that help to debug *within* the host-language resulted from our own needs during the development.

[1] `http://squeak.org/`, `https://github.com/hpi-swa/vivide`

[2] `http://mainisusuallyafunction.blogspot.de/2012/04/scheme-without-special-forms.html`, `https://github.com/hpi-swa/Ohm-S`

3.2 Debugging Walkthrough

Given a developer has built a Qoppa interpreter and found a bug: `(/ 4 3 2)` evaluates to the number `8/3` instead of `2/3`. He runs `(/ (halt 4) 3 2)` to stop the execution at the argument 4 so that he can track down the bug's origin.



```
halt
haltPrimitive
operate:on:in:
eval: (Q#halt 4) in:
wrapPrim:countingArg
collect:
wrapPrim:countingArg
operate:on:in:
eval: (Q#/ (Q#halt 4) 3
ListList:parenOpen:ex
List_list: (/ (halt 4) 3 2
List:of:
```

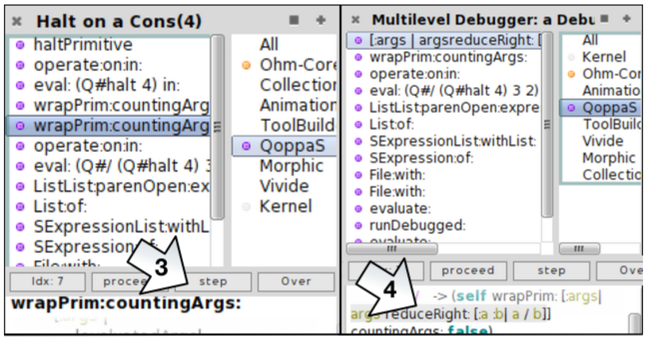
```
All
Ohm-Core
Collection
Animation
ToolBuild
QoppaS
Morphic
Vivide
Kernel
```

```
haltPrimitive
operate:on:in:
eval: (Q#halt 4) in:
wrapPrim:countingArg
wrapPrim:countingArg
operate:on:in:
eval: (Q#/ (Q#hal
ListList:parenOpe
List:of:
SExpressionList:withL
SExpression:of:
File:with:
```

```
^ [:env | env first halt]
```

Figure 3. The developer sees which frames belong together and to what level (1) and hides frames by selecting a level (2)

At first glance, he can see what frames belong to QoppaS by mapping the frame color to the level and that the correct program is being parsed by Ohm/S. (Figure 3, Step 1) The title, color, and representation of a frame is determined on a per-level base. The developer clicks on QoppaS to hide unrelated stack frames (Step 2). To inspect the execution of the `/` primitive, he clicks `operate:on:in:` frame of it, steps over the evaluation of the other arguments without stopping within the filtered `collect:`-call and steps into the implementation of the primitive. (Figure 4, Step 3) There he can find that the primitive runs a right-reduction on the arguments instead of the expected left-reduction. (Step 4)



```
haltPrimitive
operate:on:in:
eval: (Q#halt 4) in:
wrapPrim:countingArg
wrapPrim:countingArg
operate:on:in:
eval: (Q#/ (Q#halt 4) 3
ListList:parenOpen:ex
List:of:
SExpressionList:withL
SExpression
```

```
All
Ohm-Core
Collection
Animation
ToolBuild
QoppaS
Morphic
Vivide
Kernel
```

```
wrapPrim:countingArgs:
```

```
[:args | argsreduceRight: [
wrapPrim:countingArgs:
operate:on:in:
eval: (Q#/ (Q#halt 4) 3 2)
ListList:parenOpen:expre
List:of:
SExpressionList:withList:
SExpression:of:
File:with:
File:with:
evaluate:
runDebugged:
```

```
All
Kernel
Ohm-Cor
Animatio
QoppaS
ToolBuild
Vivide
Morphic
Collectio
```

```
-> (self wrapPrim: [:args |
args reduceRight: [:a :b | a / b]]
countingArgs: false)
```

Figure 4. The developer steps without leaving the level and finds the logic error

4. Related Work

Pavletic and Raza described an extension API for a hierarchical debugger that allows debugging of statically compiled DSLs [7]. While they do not describe the interactions with it, we approach it from the user experience side instead. Additionally, in contrast to compiling language workbenches statically, we focus on tools for dynamically interpreted languages instead.

Chis et. Al. described the idea of a customizable debugger and inspector in-depth [2, 3]. They applied their *Moldable* framework to a bytecode interpreter and identified step-by-step execution as a feasible implementation. We built on their rather technical analysis and envisioned interactions that would help building an interpreter from scratch.

The merging and filtering of runtime information is already applied in log file analysis. Online log file analysis services like papertrailapp.com allow the user to merge log files and filter particular entries to find correlations. Our system employs the same information-merge-and-filter concept to a systems runtime.

IDEs and frameworks provide means to remove library code from call stacks and to change the list representation of a stack frame (e.g. Visual Studio, Eclipse or Rails). Those mechanisms are designed for setting them once. However, they lack the dynamic task-based switching. Our system is more flexible to answer to the users varying needs.

5. Next Steps

The possibility to debug an interpreted language with an interpreter-level debugger suggests the question: *Is it necessary to implement a debugger for an interpreted language at all?* To investigate that, we will bundle the debugger together with the Ohm/S parser generator as an *interpreted-language workbench*, so interpreter developers get a guest-language debugger right from the beginning. Furthermore, we will build a *debugger primitive* in our interpreter to show that also interpreted languages can run interpreters using our base-level debugger.

We will investigate further in interpreters of languages with an execution model that differs from sequential execution and message passing, e.g. Prolog, APL, and State Machines. While static compilation is the focus of current DSL research, implementing an execution model that differs heavily from the host language is challenging with state-of-the-art DSL tools. In that case, building an interpreter with proper debugging support might be preferred over using a language workbench.

6. Summary

In this paper, we demonstrated the needs and benefits of a multilevel debugger. Our work-in-progress implementation promises a high value of call-stack-filtering as well as context-dependent representation in many software development environments. We will continue to work towards the use cases, requirements and implementation of an *interpreted language workbench*.

References

[1] Charles, P. et al. 2009. Accelerating the creation of customized, language-Specific IDEs in Eclipse. *ACM Sigplan Notices*. 44, 10 (2009), 191.

[2] Chiş, A. et al. 2015. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems and Structures*. 44, (Dec. 2015), 89–113.

[3] Chiş, A. et al. 2015. The moldable inspector. *2015 ACM - Onward! 2015* (New York, New York, USA, Oct. 2015), 44–60.

[4] Erdweg, S. et al. 2012. Language Composition Untangled. *LDTA '12*. (2012).

[5] Freudenberg, B. et al. 2014. SqueakJS A Modern and Practical Smalltalk that Runs in Any Browser. *DLS '14*. (2014), 57–66.

[6] Pavletic, D. et al. 2014. Extensible Debuggers for Extensible Languages. *Softwaretechnik-Trends*. 33, 2 (2014), 51–52.

[7] Pavletic, D. and Raza, S.A. 2015. Multi-Level Debugging for Extensible Languages. *Workshop Software-Reengineering und - Evolution*. 17 (2015), 21–23.

[8] Renggli, L. et al. 2010. Embedding Languages Without Breaking Tools. *ECOOP*. (2010), 380–404.

[9] Wu, H. et al. 2004. Debugging Domain-Specific Languages In Eclipse. *Eclipse Technology Exchange Poster*. (2004), 1–5.