Debugging Dynamic Language Features in a Multi-tier Virtual Machine

Anmolpreet Singh
b19070@students.iitmandi.ac.in
IIT Mandi
India

Aayush Sharma
b19229@students.iitmandi.ac.in
IIT Mandi
India

Meetesh Kalpesh Mehta
meeteshmehta4@gmail.com
IIT Bombay
India

Manas Thakur
manas@cse.iitb.ac.in
IIT Bombay
India

Abstract

Multi-tiered virtual-machine (VM) environments with Just-In-Time (JIT) compilers are essential for optimizing dynamic language program performance, but comprehending and debugging them is challenging. In this paper, we introduce Derir; a novel tool for tackling this issue in the context of Ř, a JIT compiler for R. Derir demystifies Ř, catering to both beginners and experts. It allows users to inspect the system's runtime state, make modifications, and visualize contextual specializations. With a user-friendly interface and visualization features, Derir empowers developers to explore, experiment, and gain insights into the inner workings of a specializing JIT system. We evaluate the effectiveness and usability of our tool through real-world use cases, demonstrating its benefits in learning as well as debugging scenarios. We believe that our tool holds promise for enhancing the understanding and debugging of complex VMs.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Dynamic analysis.*

Keywords: Virtual Machines, Dynamic Languages.

ACM Reference Format:
Anmolpreet Singh, Aayush Sharma, Meetesh Kalpesh Mehta, and Manas Thakur. 2023. Debugging Dynamic Language Features in a Multi-tier Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623507.3623549

VMIL '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0401-7/23/10…$15.00
https://doi.org/10.1145/3623507.3623549

1 Introduction

R is a powerful dynamically typed programming language known for its unique features, including first-class mutable environments, call-by-need evaluation, and first-class closures. However, though its language capabilities such as polymorphism, late binding, reflection, promises, lazy evaluation, and randomized order/overflow of arguments contribute significantly towards expressive power, they also lead to a complex and hard to debug runtime.

In presence of dynamic features such as the ones listed above, it is nearly impossible to impart performance through static compilation. Hence, typical runtimes for dynamic languages, including R, employ a JIT compiler to optimize code for performance. The JIT specializes the program to a particular execution instance, based on profile-guided assumptions. These assumptions could be used to create versions specific to the types of arguments supplied to a procedure, which could be selected and dispatched to based on actual arguments supplied therein. Further, in order to handle wrong assumptions at future call sites, many such runtimes employ multiple tiers: a lower "safe" tier can be used in case of failures of higher tier assumptions. Over the years, many alternative implementations have been developed to improve the performance of R [9, 11, 16, 17].

Introduction of a VM environment and complex JIT optimizations on a language is often daunting and a bug-prone task. Prima facie, a VM is a complex system, making the discovery and thorough investigation behind buggy cases hard. This often means that, even though most programs work completely fine, there may be cases where some unexpected behaviour is observed. The people who observe such behaviours are not often the maintainers of such a VM and people who actually work on the VM may not fall under the use case that triggers the bug. This highlights a major problem: the behaviour of typical multi-tier VMs presents itself as a black box to its users. Further, as we demonstrate in this paper with several examples, some of these behaviours are hard to comprehend and debug even for the VM developers.

To summarize, for end users, a VM is a magic box where they submit their programs and obtain the desired output

coupled with enhanced performance. On the other hand, for developers, a VM represents a complex set of technologies that can be further enhanced to improve performance and functionality. In this paper, we present Derir,[1] a visualization-cum-debugging tool aimed at bridging the gap between these two perspectives. Derir is designed for individuals with some programming knowledge who struggle to gain intuition behind the workings of a complex VM. It serves as both a learning resource and a visualization tool for features such as contextual dispatch [7], call-graph generation, as well as modification of various aspects of the runtime state. This combination of capabilities enables beginners to learn about VM internals while providing developers with a means to debug complex scenarios.

As our test bed we chose the Ř [6], a JIT-based VM that can achieve several times faster execution than standard GNU-R. Ř specializes R functions by compiling them using features such as speculation, type feedback, and contextual dispatch. The VM itself uses multiple intermediate representations for its functioning and performs a wide range of optimizations both in terms of compilation and dispatching. Thus, the practical objective of our work is to provide a peek into the Ř VM and its inner workings, while providing a simple and intuitive interface that anyone can use and extend for even other programming languages and virtual machines.

2 Background and Preliminaries

Developed in the early 1990s, R has gained popularity among statisticians, data scientists, and researchers due to its extensive range of statistical and graphical techniques, as well as its flexibility and extensibility through packages. R's rich set of dynamic language features make it possible to use it under various paradigms and target diverse workloads.

During execution, R code is first converted into an abstract syntax tree (AST), which is then evaluated recursively; notably, this can be a very slow process [17]. To speed up R's performance, GNU introduced a bytecode interpreter. This bytecode interpreter compiles the ASTs into bytecode sequences, which can be interpreted much faster. This is the current implementation that the standard GNUR implementation provides to its users. However, this implementation still leaves scope for performance improvement for a dynamic language, and in recent years, there have been a number of projects that have further improved the performance of R using various techniques [9, 11, 16]. One such project is Ř [6], which is a virtual machine (VM) that JIT compiles R code on the fly and has been shown to achieve significant performance improvements over GNUR for a series of real-world programs.

[1]Pronounced as *de-rear*.

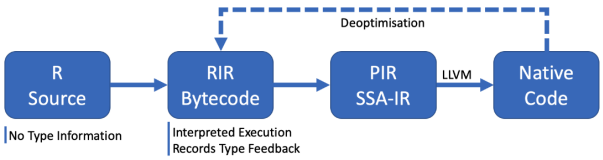

Figure 1. Graphical representation of the Ř compilation pipeline.

2.1 The Ř VM

The Ř VM (shown in Figure 1), like many other VMs [9], translates the source code into an intermediate representation called rir, which is a bytecode that can be interpreted by the Ř VM. During bytecode execution, the runtime performs profiling and collects useful contextual cues for the various called functions. The contextual cues are stored inside inline caches in the bytecode itself and are instrumental to generating optimized code. When code regions become hot (i.e. deemed suitable for compilation), they are further optimized in the pipeline and compiled into native code. This process involves converting rir into pir, which is a register based static single assignment IR used for performing high-level optimizations. The final optimized pir code is then translated into LLVM IR, which is used to perform low-level optimizations. The final code is then compiled into a executable system native binary.

Like most VMs, Ř also optimizes non-code-related aspects of the R implementation. For instance, in R, each function call generates a new evaluation environment initiated by the caller and passed to the callee. [2] In contrast, when Ř compiled code needs to be evaluated, the runtime delegates the creation of the evaluation environment to the callee; enabling it to forgo environment creation when it's not needed. Additionally, Ř also supports stub environments (faster implementations of hashmaps that can be materialized as R environments on demand), implementing fast cases for common operations, and so on.

2.2 Contextual Dispatch

Unlike many traditional JIT systems, Ř is capable of compiling and dispatching to multiple versions of functions dynamically during runtime using contextual dispatch [7]. Contextual dispatch allows Ř to maintain various specialized (optimized) versions of a single function under different contexts. A context is computed dynamically during runtime using the arguments provided to a function at the call site. The contexts represent a set of predicates that must be true

[2]An environment in R stores bindings from symbols to values, along with a pointer to the parent environment.

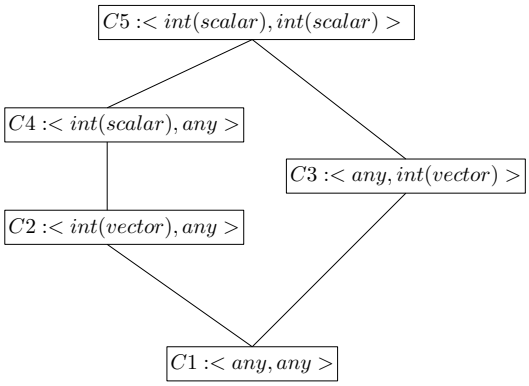

Figure 2. Lattice of contexts in Ř.

for a compiled function to be compatible for use at a given call site. [3]

Figure 2 shows an example context lattice created by Ř during runtime. The baseline context is $C1$ which does not assume anything about the arguments and serves as a fallback if no specialized version exists; this is the largest context as it allows for all valid invocations to be executed. Context $C2$ is said to be smaller than $C1$ because it asserts the type of the first argument to be an integer vector. $C4$ asserts even stronger property on the first argument, asserting that the type must be a vector of size one i.e. a scalar.[4]

In implementation, the set of predicates include features such as the type and the shape of the parameters, the number of arguments provided, information about missing arguments, order of arguments, as well as their types. Every time a function is invoked, the dispatcher picks the version that is closest to the calling context.

3 Motivation

One of the primary goals of a JIT-based VM is to provide its users with an environment that executes code faster than plain interpretation. This involves tasks such as profiling of the runtime environment, speculating on collected information, and establishing fallback mechanisms when things go wrong. Further, programs in R can modify the stack frames at runtime (allowing its users to write dynamically scoped code), runtime reification of environments means the reaching stores to a variable cannot be determined statically, side effects stemming from lazily forced promises can create complex control flows leading to hard to debug code, and many more. This presents two interesting challenges. Firstly, the progression of system state in presence of dynamic language features is difficult to comprehend statically. Secondly, what a VM actually does behind the scenes that leads to real-world

[3]In Ř, contexts are a set of preconditions that decide if a piece of compiled code is sound to execute.

[4]In R, scalars are treated as vectors of size one.

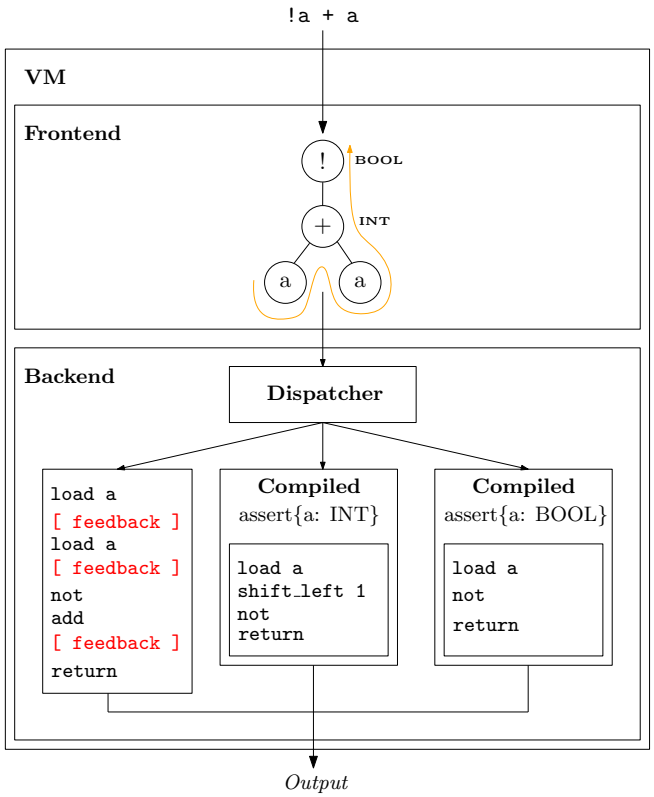

Figure 3. An example to show an interesting scenario while parsing expressions in R.

performance benefits, is completely hidden from the user. Let us examine the problems in more detail.

I. Consider Figure 3, where the input to the VM is the expression `!a + a`. In boolean algebra, this expression represents an unconditional true and it can generally be expected that this code must return unconditional true values when provided with any boolean. This, however, is not true in case of R (where evaluation of plus precedes the negation operation). When provided with boolean values, it acts as a logical negation (not gate) and returns an unconditional false for all non-zero integers. This behavior is a result of the way R parses expressions; it is not an ambiguous term in R but a behaviour that is designed to work a certain way. The information about the specifics of how such expressions are evaluated by R is hidden away from the users. In the figure we can clearly see the formation of the AST and the order in which operations are going to be performed. Being able to see some part of the internals can provide users with not only the implementation specifics but also as a learning tool to understand how the language actually works. Interestingly, the story does not end there.

Upon examining the VM's backend, specifically a dispatcher that is responsible for selecting versions of code most suited for a use case, it is easy to determine what this

```
1  g <- function(b) { a <- b; print("g called"); }
2  f <- function(a) {
3     print("f called"); g(a);
4     print("f call end");
5  }
6  foo <- function() {
7     for (i in 1:20) {
8        if (i == 5) {
9           f(break)
10       }
11       print(i)
12    }
13 }
14 foo()
```

Figure 4. An example to show complex control flows in R.

code does. JIT compilers in VMs also employ various analyses and optimizations to reduce trivial work. In this case, the JIT compiles two optimized code versions: when the type of `a` is an integer, the final result can be calculated quickly by just left shifting the value and negating it; and the other optimized case in which the type is a boolean, simply negating the value is enough. Both these versions get rid of load operations that involve expensive lookup into the environment and also do less computation, thus saving execution time. Being able to see these two versions and how they are generated can provide developers useful insights about when the underlying JIT can optimize code more, and how.

II. Determining the behaviour of programs in presence of dynamic language features can obscure understanding and debugging, even in seemingly simple code. For example, in the code snippet in Figure 4, function `foo()` iterates over the variable `i` from one to twenty and prints its value at each interation. If the condition on line 8 is satisfied, the function `f` is invoked, which prints "f called", followed by an invocation of `g`. Inside `g`, the argument passed to it is assigned to a local variable `a` followed by printing "g called". After this call completes, "f call end" is printed and the control returns back to the for loop.

We would expect the previously discussed code to print "1 2 3 4 f called g called f call end 5 6...", but that is not what happens. The output stops at "1 2 3 4 f called"; if we follow the execution and examine the program state at each line, we observe that the `break` statement is executed when the expression `a <- b` is evaluated (despite arguments being lazy in R, the assignment operation always ends up forcing the argument). This behavior involves two key factors: lazy evaluation and premature collapsing of the stack frame of the function `f`. As a consequence of the `break` statement's execution, the stack frames of both `g` and `f` collapse, and control directly exits the for loop.

By examining such scenarios, we find that having some mechanism to clearly visualize the runtime state of a VM can serve as a learning aid for its users while also providing a handy tool for the experienced developers to quickly experiment with the system. Derir focuses on the following aspects of a VM:

1. Clearly presenting the internal state of the system.
2. Allowing users to modify and experiment with the runtime state.
3. Basic set of tools such as the ability to step into and step over interpreted code.
4. Visual representation of the contextual dispatch system.

The aim of Derir is to enhance the understanding and debugging capabilities of users, facilitating both learning and development in the context of complex VMs.

4 System Model and Functionalities

The implementation of R is heavily inspired by the LISP programming language. Everything in R, either a function or an object, is stored as an SEXP (S-Expression). This is implemented as a C struct, where a tag field determines the type of the object. Directly interacting with these objects in C++, though possible, makes the experience more cumbersome. To build Derir, we handle and encode these SEXP objects into JSON and transmit them over sockets between the backend and the frontend. The runtime is instrumented at various program points and allows the user to view and modify the program state at these points.

Figure 5 shows the flowchart of Derir. The application uses an event loop to exchange data between frontend, middleware and backend. The data is requested by the frontend through the middleware and upon receiving the request, the backend sends the data back to the frontend as a JSON string which is then parsed by the frontend; this parsing is done with a JavaScript program utilizing the React Framework.

The execution pauses at the current offset of the bytecode and waits for the user input before proceeding to the next instruction in the bytecode. During this step, the backend sends a SYN request that contains information about the program counter and the code object being executed. The frontend, upon receiving the request can choose to request the data that it does not have. It emits multiple data requests with different tags; these tags include bytecode, feedback slot information, source code, runtime environment, runtime stack and contextual dispatcher information. The frontend decides to cache things like the source code and the bytecode so that it does not need to be fetched again in future. Whenever the user presses the next/step button, the execution continues until the next program counter or next step, respectively. The debugger also allows the user to manipulate the environment as well as type feedback. The update request is sent to the backend where we have written a custom class to manipulate the current environment bindings, and update the flags of the type feedback as received from the user. In order to ensure consistency between the different tiers, the

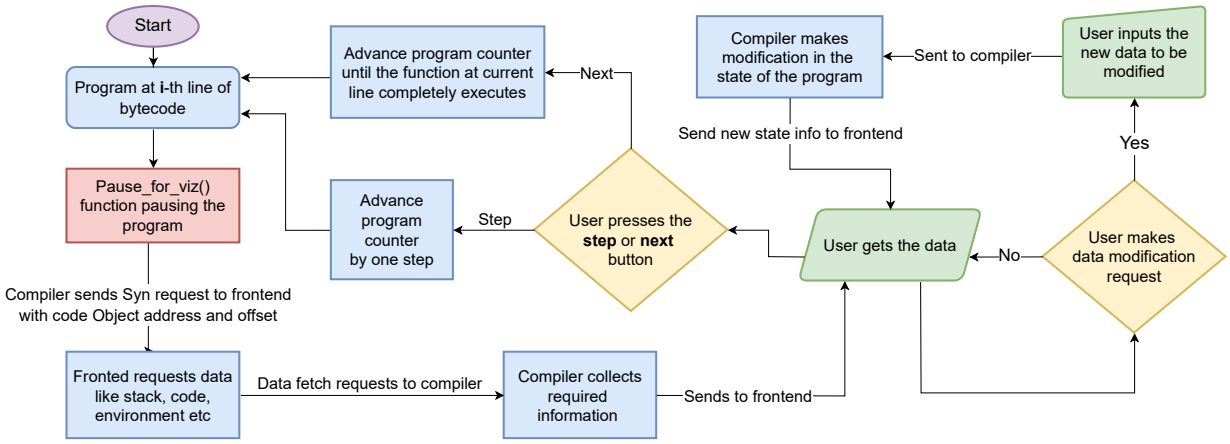

Figure 5. Figure showing the flow of information through Derir.

frontend keeps track of the sending/receiving of data so that no requests are made in between the transfers. The data is updated each time the user does a step or next.

As can be seen, the above model allowed us to create a debugging and visualization framework that is not only simple to work with, but as later shown in Section 5, is also extensible to new features by just adding more information as part of the saved JSON objects.

Figure 6 shows a part of Derir in action. The GUI has been written in React, which allows us to create generic components that can be reused for future implementations. The different components, annotated one through six, are laid out in a grid and allow the user to simply drag and drop to customize the layout. The right end of the top bar shows the execution state of the program and the buttons next to it are used for restarting the connection and triggering dark mode, respectively. When execution starts, a synchronize packet is sent between the backend and the GUI; this is handled though an intermediate middleware server that only serves as a forwarding server in our implementation. The various components of Derir are detailed as follows.

4.1 Bytecode Stepper

The text on the top part of the component displays the current synchronization status, which details the memory address of the code being executed and the type of object (bytecode or native code). If the frontend has not previously executed the function it requests the bytecode from the backend and caches the result to avoid requesting the same data again.

The center portion of this component displays the bytecode being executed and highlights the current program counter (PC) offset in blue. The values shown in the blue outlined button represent inline caches/feedback slots used for profiling in Ř. When clicked, these boxes open a popup that can be used to modify the state of the system feedback at that point (discussed in the following sections).

The bottom portion is used for navigation and stepping through the bytecode. For large bytecode sequences the user may scroll though the code and lose track of the current instruction, hence the scroll button allows scrolling the current instruction into the view. The next button is used for stepping over instructions and the step button is used to step into instructions. The checkbox "Keep Bytecode Sync" can be selected to immediately update the feedback slots when they are updated; this may be disabled if the user is not concerned with the profiling information, thus improving performance.

4.1.1 Type Feedback.

The rir bytecode has some empty slots reserved for type feedback. Type feedback records information about function calls and variables. For variables, it records all the types of bindings it has stored in the past. Derir allows the user to view/modify the state of the type feedback of the system using the GUI. The compiler uses this information during profiling and then uses this information to make decisions about optimizations. We have allowed the user to modify the type feedback to play with the optimizations or avoid deoptimizations. Figure 7 shows the type-feedback slot modification panel in Derir.

4.2 Source Code

This component simply displays R's interpretation of the source code. When source code is parsed, the resultant ASTs follow a certain order of operations. The source code displayed in this component during interpretation makes the exact order of evaluation of expressions explicit.

4.3 Environment

This component contains all the bindings between symbols and their values in the current evaluation environment. The user has an option to modify the bindings from the frontend using the dropdown and the text boxes. Currently the debugger/visualizer is able to handle conversion to four datatypes:

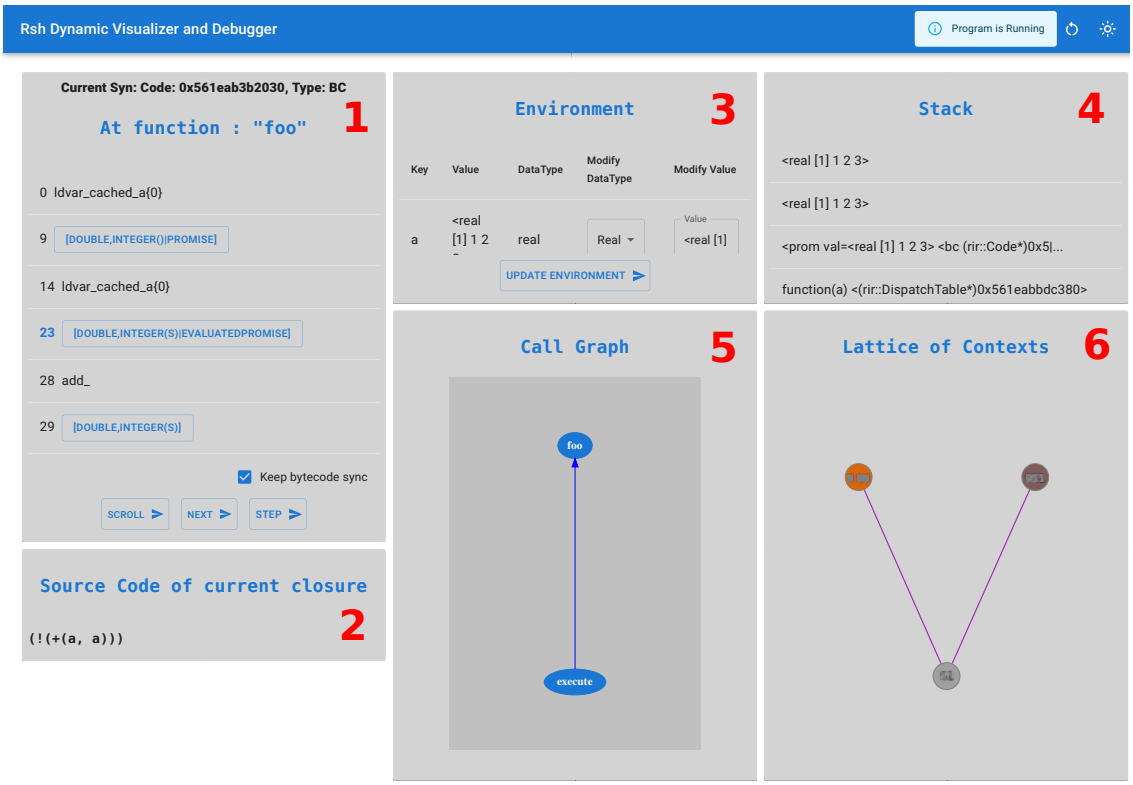

Figure 6. Figure showing the different components of Derir.

integer, real, logical and string; we hope to extend support to arbitrary assignments using `eval` in the future.

4.4 Stack

After each step, the backend sends the runtime stack to the frontend. Our frontend prints the entire stack frame (note that the stack grows from bottom-to-top). If the stack frame consists of `INTSXP`, `LGLSXP` or `REALSXP`, then we print their actual values, otherwise we print the default value as it is received.

4.5 Callgraph

The callgraph displays the order in which functions get called. This is done by building a DOT file (format used by GraphViz [5]) at the backend. This DOT file is sent to the frontend as a string, which is then converted to a graph using the graph-viz library [14] of npm.

4.6 Contexts

Each time a function call is made, a `doCall` method is invoked in Ř. This method takes care of all the function calls, including the deoptimzations/recompilations, and has access to the dispatch table of the called closure. The dispatch table contains all the versions of the called functions compiled under different assumptions. We keep track of all the contexts under which the given closure was compiled and the number of times each compilation was invoked. In order to build the contextual lattice, we needed to compare the contexts (which follow a partial order). Note that a function specialization can be invoked only if its context is greater than or equal to the current context. Thus, the contexts that are comparable are connected to each other in the lattice.

As there is a partial order between the contexts of a function, we construct the context lattice from the array of contexts at the frontend. Firstly, we build an inclusion array to include all the contexts that can be a child of a given context or its assumptions are a subset of the context's assumptions. Then we generate level data by starting from leaves and deleting links to their immediate parents iteratively until there are no nodes left. The baseline context is the parent of all contexts. Then we generate edges by linking all the nodes in the previous level that contain the contexts of the nodes to the nodes in the next level. Finally, we have a lattice displayed in the frontend with a ToolTip displaying the context of a particular node, and a number in the node represents the number of runs with specialized function for that context.

5 Case Studies

Derir aims to be useful in two ways. First, it is designed to make learning about the virtual machine easy for beginners. It presents visualization of complex compiler concepts in

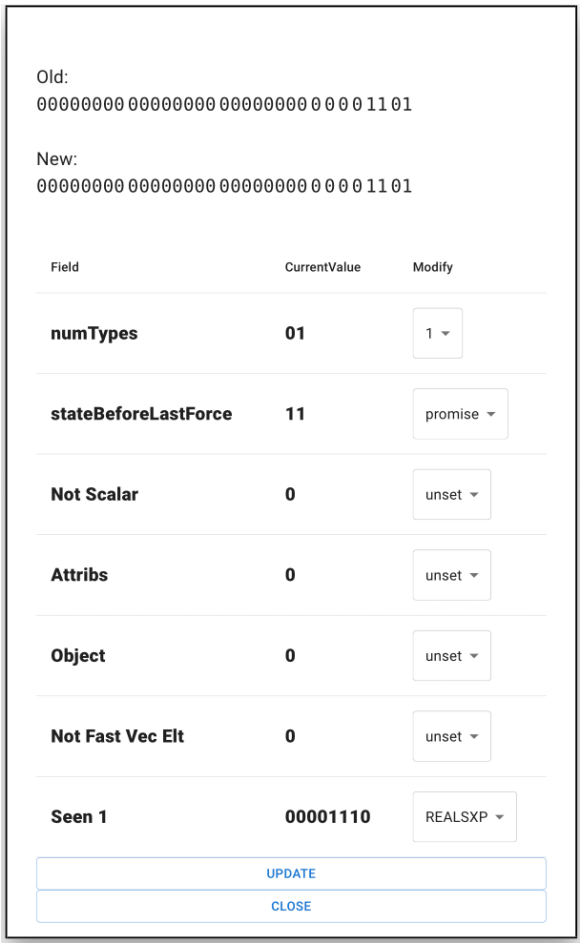

Figure 7. Figure showing the type-feedback modification panel of Derir.

a simple and understandable manner. Second, it provides advanced features for experienced developers to debug the system effectively. By offering both simplicity for learners and depth for experts, Derir becomes valuable to users at all levels, and helps them work with as well as for a dynamic language as sophisticated as R.

5.1 As a Learning Tool

As a learning tool, it is interesting to see the kinds of small intricacies and optimizations that happen in the runtime. This involves looking at how different components of the JIT work in tandem to collect profiling data and optimize for specific cases.

Consider the code snippet in Figure 8. Line 1 is used to connect to the frontend visualizer. Function `f` is then called using different call-site contexts (line 3- 6); integer, vector of integers, integer and a boolean respectively. It may seem inconsequential, but behind the scenes the VM performs a bunch of different tasks. Figure 9 shows the impact of these different calling contexts on the state of the system.

```
1 rir.viz("http://127.0.0.1:3011")
2 f <- function(a) { !a + a; }
3 f(1L)
4 f(c(1L,2L,3L))
5 f(1L)
6 f(FALSE)
```

Figure 8. Code snippet in R where function `f` is called under different calling contexts.

Under the first calling context (line 3), notice that the feedback slot (shown in state one of Figure 9) holds `integer(s) | evaluatedPromise` in its feedback for loading the argument `a`. This is highly irregular if one thinks about it, because as per R semantics all arguments must be passed as promises and be evaluated only when they are forced but in case of R, the feedback says the promise was already evaluated before; when did this happen?. The underlying reason is that Ř decides to optimize call sites by eagerly evaluating promises that are safe (in this case, a promise holding a scalar integer is free of any side effects implying it is safe to be passed eagerly instead of lazily).

Under the second calling context (line 4), a vector of integers is passed to the function. In Figure 9 state two, we can see that Ř cannot eagerly evaluate the argument hence the feedback is more generalized. Notice that the 's' indicating scalar also gets removed. The new feedback `integer() | promise` is modified in place. This shows the generalization of feedback as time progresses; which means that code compiled later in the runtime is likely to be less specialized.

Under the third calling context (line 5), a previously seen context (the one created at line 3) is called again. In Figure 9 state three, the runtime decides to JIT compile a version of `f` that is specialized to handle integer inputs (the new version is the one with green color and an integer inside showing the number of calls to this version, which is 1 in this case) This is depicted as a new node in the context lattice, representing a specialized version of `f`.

Under the fourth calling context (line 5), a new argument type is seen (a logical value). In Figure 9 state four, we see that the previously compiled version of `f` was not dispatched; instead, the runtime dispatches to the interpreter version of `f`. Also notice that the feedback is further generalized to handle both integers and logical values, seen as `integer,logical() | promise`.

Derir makes understanding this evolution of contexts across call sites of interpreted as well as compiled functions extremely simple, and helps users visualize interesting stages their code goes through, in the underlying multi-tiered VM.

5.2 As a Debugging Tool

Compiler developers often have a set of test cases and benchmarks that they run to verify the system. These test cases are

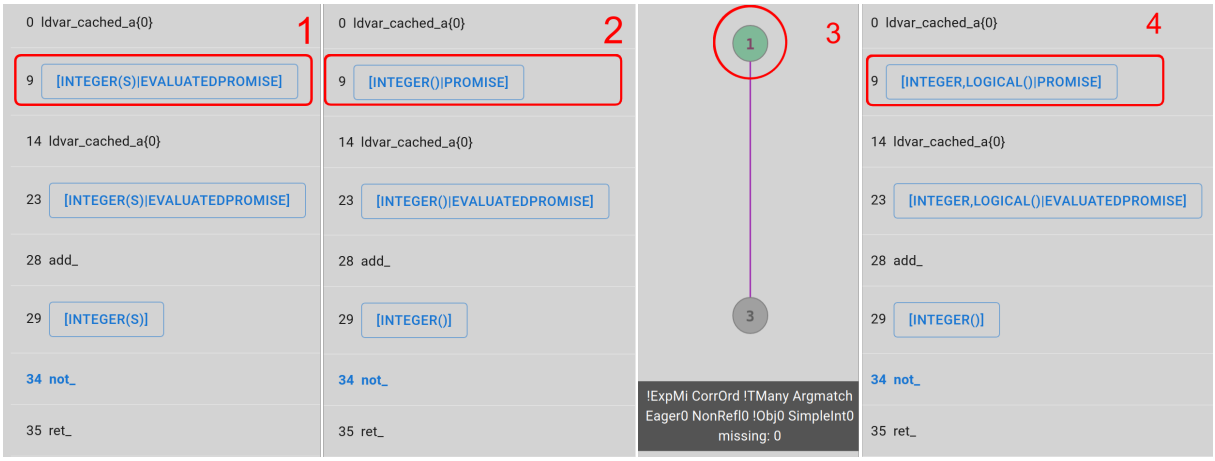

Figure 9. Figure showing the progression of states for f under different call-site contexts.

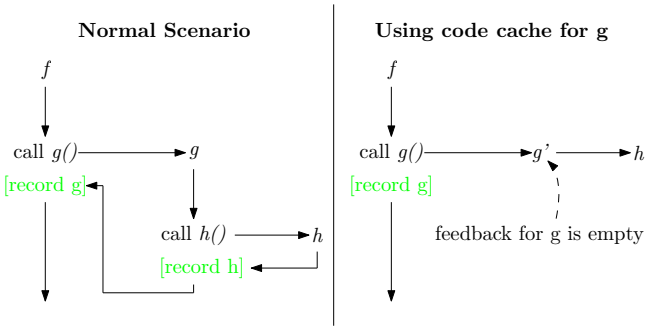

Figure 10. Figure showing the new state reached when using a code cache.

seldom functionally complete; as new functionality and optimizations are added, there is always a possibility that new bugs get added to the system. Sometimes when major revisions happen to the underlying language implementation, it is possible that the intermediate languages that worked fine before now generate erroneous code for specific use cases. On the other hand, sometimes even non-functional changes (such as introduction of a code cache to promote code reuse under lazy evaluation) result in behavioural changes that can reveal hidden semantic problems that were never triggered before. We found few such interesting cases using Derir.

5.2.1 Debugging and reproducing bugs. Ř JIT compiler is known to produce performant code, but this often comes at a substantial cost in the form of compilation time. We decided to implement a code cache mechanism that would save code to disk and reuse it in future executions for faster warmups. This change seemingly should be error free, provided that the implementation takes care of handling all the references correctly. However, in our implementation, we found a bug that would happen when a code cache was used; essentially, we observed that during code reuse, the system could reach a previously unreachable state in terms of type-feedback collection.

Figure 10 shows an abstracted out scenario that gets introduced when using a code cache. Here, f is a function that calls g (a generic function that decides the final function to call based on the class of the arguments), which in turn calls h, the final target. In Ř, a prominent pass responsible for performant code is the inlining pass. Under normal execution, when f calls g, it records the information about which function was called at the call site. Similarly, when g calls h, it records the information about h as before. When the JIT decides to compile f, g gets inlined into the body of f (in this example, both g and h are monomorphic call targets at their respective call sites).

When we introduce a code cache, first call from f to g ends up actually calling g' (a precompiled version of g from the code cache). Under normal scenario, the first call would always result in interpretation of g. If g is never interpreted, no feedback gets collected and all the slots are therefore empty. Now when the JIT decides to compile f an interesting runtime state is reached. The following, previously impossible condition, is possible now: "f inlines g as it is a monomorphic call target from f, but all type feedback inside g is empty, suggesting that the body of g was never called previously".

What happens above is that the body of g gets inlined into f as before. However, after the inlining pass, another pass that is responsible for removing previously unexecuted branches comes into picture. This pass removes part of the inlined code that calls h and replaces it with a dead call (an unconditional deoptimization point). We found that when this deoptimization actually occured, control transfers back to interpretation of g, which then calls h. The called function h then throws an unexpected error indicating that no arguments were supplied to it. This was because generics has a much different calling convention that other functions in R, if some specific flags and pointers in the calling context

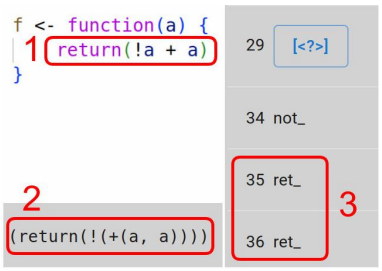

Figure 11. Figure showing the generation of redundant return statements in rir bytecode.

are not set correctly the called function may fail to find the supplied arguments. In Ř, the environments were not set correctly for some inlined generics; leading to JIT compiled code sometimes failing to execute correctly when a code cache was introduced.

Though this is a very specific case, the main problem is the difficulty in debugging and creating small reproducible examples. Using Derir, it is possible to simply clear the feedback slots of g and easily reproduce the problem, while having a clear picture of the runtime state.

5.2.2 Redundancy in the runtime. In Figure 11, we see the generated bytecode for function `f` (annotation one). This function evaluates the expression `!a + a` (as discussed previously in Section 3). When running the function `f` using the visualizer we notice something strange. There were two return instructions in the generated bytecode (annotation three); the second return is unreachable. The second return was added as a fallback in case no return statement was present; as all R functions implicitly return the last evaluated expression. This indicates that there are cases where the rir bytecode generation could be further improved. Such code generation will not lead to any crashes or any unexpected behaviour, it simply takes more memory than it should. It still interesting to see that such a thing can go unnoticed, as developers seldom actually look into these parts of the runtime until things start to break.

5.2.3 Redundancy in specialization. Sometimes a function may get specialized again and again, but it may not be useful for it to be even compiled. Imagine compiling a piece of code under tens of different contexts when all it does is simply pass all the incoming arguments to an external function. For instance, consider the function `match` shown in Figure 12. In this case, we compile seven different contexts for `match` despite the fact that its specialization is unlikely to provide any further benefit; see Figure 13. While looking at functions using the logging capabilities of Derir, we found this to be the case for many inbuilt functions. These functions are very generic and take many conditional arguments as input and they simply pass it forward to a precompiled piece of C code for execution. Using this observation, we

```
1 match <- function (x, table, nomatch = NA_integer_,
      incomparables = NULL) {
2   .Internal(match(x, table, nomatch, incomparables))
3 }
```

Figure 12. Code snippet to explain a common regression scenario for contextual specialization.

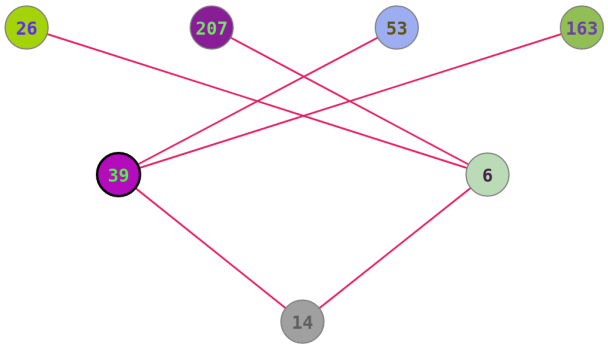

Figure 13. Figure showing the contextual dispatch lattice for function `match`. This illustrates a case where redundant contextual specializations may be created in Ř even when they yield no performance benefit.

can create heuristics that restrict such functions from being overspecialized.

The above case studies and example scenarios just showed the benefits a powerful visualization and debugging tool like ours could help impart to a powerful and complex runtime; we believe the use cases will grow as we continue using Derir for Ř and as such tools get developed for and adapted to more number of dynamic language runtimes.

6 Discussion

6.1 Extensibility to New Features

Derir uses JSON as the only format for data transfer between the frontend and the backend. The frontend is free to request any new data from the backend using a data request. For example, currently it requests stack data, code, and more (as discussed in Section 4), which can be easily extended by adding more data handling classes to the backend. In the backend, a special callback is invoked when a data request is received from the frontend. This callback can be easily customized with a custom class to suit user needs. The class can simply modify the state of the system and send back the updated state, or set predicates to control debugger-flow. As an instance, a module can be added to pause the execution only when a specific criteria is met or to visualize more aspects of the runtime; e.g. pausing during nth invocation of

a function, sending argument information for all invocations of a function, pausing execution when a compilation takes more than a threshold time, and so on.

6.2 Limitations

In the current implementation of Derir we have not implemented features like hot-code replacement or direct modifications to the generated bytecode. One of our main focus was to limit the user interface options to a minimum, allowing beginners to get an understanding of the underlying system without having to worry about various options/configurations. Though this may appear to hamper the debugging capability for more advanced users, such features can be easily implemented and integrated as need arises. One limitation of Derir is that it does not integrate with the garbage collector. We believe that future integration of this feature would improve Derir by allowing users to see allocation and deallocation of objects in memory and also understand how the underlying garbage collector works.

7 Related Work

Understanding and debugging multi-tiered virtual-machine (VM) environments with Just-In-Time (JIT) compilers has been a subject of interest in the field of dynamic languages optimization. Several research efforts have been made to address the challenges associated with comprehending and debugging the internal workings of such systems. Tools like JSExplain [2] present reference interpreters that closely follow language specifications and produce execution traces. These traces offer insight into the behavior of JIT-compiled programs and aid in identifying and resolving bugs. Würthinger et al. [19] discuss the challenges of debugging meta-circular VMs and the need for tools that provide access to both platform-independent and platform-specific details. Luxton-Reilly et al. [10] present Ladebug, an online tool designed to scaffold the learning of debugging skills for novice programmers, encouraging them to investigate runtime states. Derir complements this approach by providing users with a deeper understanding of a real-world specializing JIT system, Ř, and its optimizations, empowering them with insights into the VM's runtime state and the ability to make modifications. Wuerthinger [18] implemented a visualizer for the ideal IR, which is a graph based IR used when compiling Java bytecodes into machine code. A frontend visulaizer created by Miller [12] for the YJIT [3] allows for visualization of dynamic runtime specialization features of the JIT. Similar to Ř, YJIT supports runtime specialization based on profiling, but unlike the contextual dispatch system of Ř YJIT employs a lazy basic block versioning scheme to specialize code at runtime. An article by Bolz-Tereick [1] talks about the different ways PyPy uses GraphViz [5]; even in this work we make use of GraphViz for visualization of different runtime states. The widely popular python code execution tool by Guo [8] has over the years been extended to languages such as Javascript, C, C++ and Java. The tool is popular among beginners, who can use the tool for debugging and understanding code execution.

Visualization has proven to be a powerful approach for understanding complex systems, including VMs. Previous works [4, 13, 15] have introduced visualization techniques that help developers gain insights into the behavior of these systems. These visualizations often show the execution flow, profiling information, and the impact of different optimizations on program performance. However, while visualization capabilities have been discussed in previous works, Derir aims to provide a novel and user-friendly interface specifically for understanding and visualizing the contextual specializations in a JIT-based VM.

8 Conclusion

The onus of imparting performance to programs written in modern dynamic languages is often on a managed virtual-machine that uses JIT compilation to generate specialized machine code. Such systems almost always perform speculation and hence are coupled with fallback options in the form of an interpreter or a baseline compiler. Notwithstanding the advantages, it is not easy for compiler designers to understand these complex multi-tiered systems: the stages the code travels through, the various intermediate forms, the optimizations that depend heavily on live profile, and so on.

In this paper, we introduced a visualizer-cum-debugger called Derir for an optimizing VM runtime, for the R programming language. Derir facilitates visualization of the runtime state and allows changing values as well as feedback information to comprehend their effects. Specifically, we support stepping through the intermediate language rir of the Ř JIT compiler, updating interesting data structures related to its optimization passes, and illustrating the contextual lattice and the runtime call-graph. We also allow tweaking the runtime environment and the type-feedback profile while updating all these components. We also highlight interesting use cases of the tool, both as a user and as a developer, and show how they could be used to understand language features as well as identify compiler bugs or scopes for improvement. Finally, Derir is open source and designed modularly using latest technologies to support extensibility.

In future, we believe our tool would help promulgate the idea of "helping compiler developers" and consequently, pull in more experimenters, engineers and researchers alike to explore the nuances of wonderful tiered runtimes, for dynamic languages including R and beyond.

References

[1] Carl Friedrich Bolz-Tereick. 2021. Some way that PyPy uses Graphviz. https://github.com/jimmyhmiller/PlayGround.

[2] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings*

of the The Web Conference 2018 (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 691–699. https://doi.org/10.1145/3184558.3185969

[3] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: A Basic Block Versioning JIT Compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3486606.3486781

[4] Marcus Ciolkowski, Simon Faber, and Sebastian von Mammen. 2017. 3-D Visualization of Dynamic Runtime Structures. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement* (Gothenburg, Sweden) *(IWSM Mensura '17)*. Association for Computing Machinery, New York, NY, USA, 189–198. https://doi.org/10.1145/3143434.3143435

[5] Peter Eades. 2023. Graphviz. https://graphviz.org/

[6] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/3359619.3359744

[7] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428288

[8] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[9] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. *SIGPLAN Not.* 49, 7 (mar 2014), 89–102. https://doi.org/10.1145/2674025.2576205

[10] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: An Online Tool to Help Novice Programmers Improve Their Debugging Skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 159–164. https://doi.org/10.1145/3197091.3197098

[11] Microsoft R Open. 2015. Microsoft R Open. https://github.com/microsoft/microsoft-r-open

[12] Jimmy Miller. 2022. YJIT visualizer frontend. https://github.com/jimmyhmiller/PlayGround.

[13] Oracle Corporation. 2023. Java VisualVM. Retrieved 2023-07-24 from https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/

[14] Dominic Parfitt. 2022. graphviz-react. https://github.com/DomParfitt/graphviz-react

[15] Jeremy Singer and Chris Kirkham. 2006. Visualized Adaptive Runtime Subsystems. In *Proceedings of the 2006 ACM Symposium on Software Visualization* (Brighton, United Kingdom) *(SoftVis '06)*. Association for Computing Machinery, New York, NY, USA, 195–196. https://doi.org/10.1145/1148493.1148541

[16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. *SIGPLAN Not.* 52, 2 (nov 2016), 84–95. https://doi.org/10.1145/3093334.2989236

[17] Luke Tierney. 2019. A Byte Code Compiler for R. http://www.stat.uiowa.edu/~luke/R/compiler/compiler.pdf

[18] Thomas Wuerthinger. 2007. Ideal Graph visualizer. https://ssw.jku.at/General/Staff/TW/igv.html.

[19] Thomas Würthinger, Michael L. Van De Vanter, and Doug Simon. 2010. Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture. In *Perspectives of Systems Informatics*, Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–412. https://doi.org/10.1007/978-3-642-11486-1_34

Received 2023-07-23; accepted 2023-08-28