

Debugging Dynamic Language Features in a Multi-Tier Virtual Machine

Smoke and Mirror

Meetesh K Mehta

Advisor: Dr Manas Thakur

IIT Bombay

June 2, 2023



Dynamic Languages are **great!**

The **R** programming language, JS, lua...

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments
- Lazy evaluation

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments
- Access to runtime stack

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments
- Access to runtime stack
- Eval

Dynamic Languages are **great!**

The **R** programming language, JS, lua...

- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments
- Access to runtime stack
- Eval

Slow and **complex** runtimes

The problem with optimizing dynamic languages

```
1  # Parent Scope
2  x <- 100
3  √ f <- function(a) {
4  |     a;
5  |     # Function Scope
6  |     print(x);
7  | }
8
9
10
11
12
13
14
15
16
17
```

The problem with optimizing dynamic languages

```
1  # Parent Scope
2  x <- 100
3  √ f <- function(a) {
4    a;
5    # Function Scope
6    print(x);
7  }
8
9  f(10)
10
11 # Output: 100
12
13
14
15
16
17
```

The diagram illustrates the variable resolution process. It shows a function `f` with a local scope containing `a=10` and a global scope containing `x=100`. A green arrow points from the `x` in the function's `print(x)` statement to the `x=100` in the global scope. Another green arrow points from the `x=100` to the output `100`. The function `f` is labeled in red, and the global scope is labeled `global` in red.

The problem with optimizing dynamic languages

```
1 # Parent Scope
2 x <- 100
3 v f <- function(a) {
4     a;
5     # Function Scope
6     print(x);
7 }
8
9
10
11
12
13 badIdea <- function() { assign("x", 11, sys.frame(-1)); 1; }
14 f(badIdea())
15
16 # Output: 11
17
```

The diagram illustrates the environment stack during the execution of the provided R code. It consists of three frames:

- global** (bottom frame): Contains the variable `x = 100`.
- f** (middle frame): Contains the variable `x = 11` and the variable `a = badIdea()`. A blue arrow points from the `a;` line in the function code to this assignment. A green arrow points from the `x = 11` assignment to the `print(x);` line in the function code.
- badIdea** (top frame): This frame is currently empty.

The code below the diagram shows the execution of `badIdea()` which assigns the value 11 to the variable `x` in the parent frame (the `f` frame), and then calls `f(badIdea())`, which prints the value of `x` (11).

The problem with optimizing dynamic languages

```
1 # Parent Scope
2 x <- 100
3 v f <- function(a) {
4     a;
5     # Function Scope
6     print(x);
7 }
8
9
10
11
12
13 badIdea <- function() { assign("x", 11, sys.frame(-1)); 1; }
14 f(badIdea())
15
16 # Output: 11
17
```

The diagram illustrates the environment structure for the function `f`. It is represented as a box divided into two horizontal sections. The top section is labeled `f` (Function Scope) and contains the assignment `a=10`. The bottom section is labeled `global` (Global Scope) and contains the assignment `x=100`. A green circle highlights the `a=11` assignment in the function scope. A green arrow points from the `x` in the `print(x)` statement (line 6) to the `a=11` assignment, indicating that the function finds the variable in its own environment. Another green arrow points from the `badIdea()` call (line 14) to the `a=11` assignment, showing that the function environment is created with `a=11` as its parent environment.

The \checkmark JIT compiler

Optimize for the most common cases.

- Runtime profiling
 - Collect information about **types**, **call sites**, **branches**.

The \checkmark JIT compiler

Optimize for the most common cases.

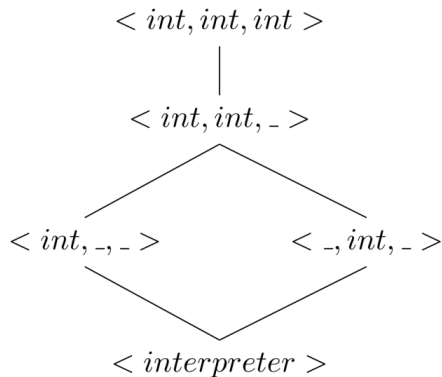
- Runtime profiling
 - Collect information about **types, call sites, branches.**
- Contextual Specialization
 - Optimize for **classes of behaviours.**

Contextual Dispatch

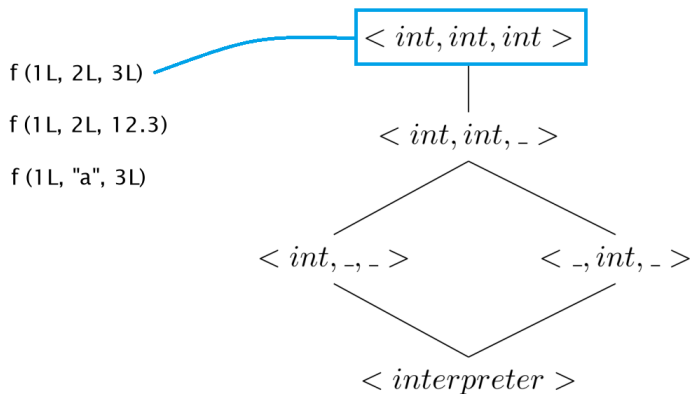
f (1L, 2L, 3L)

f (1L, 2L, 12.3)

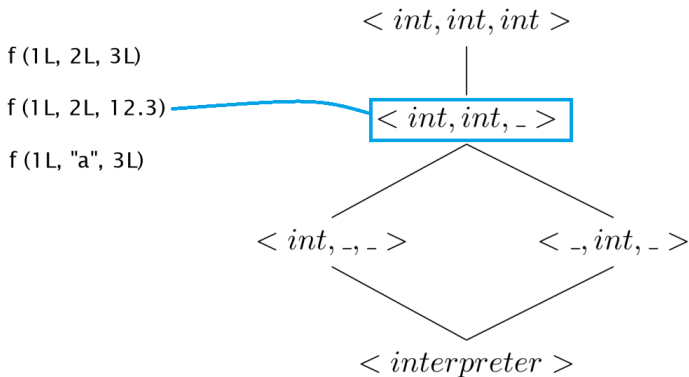
f (1L, "a", 3L)



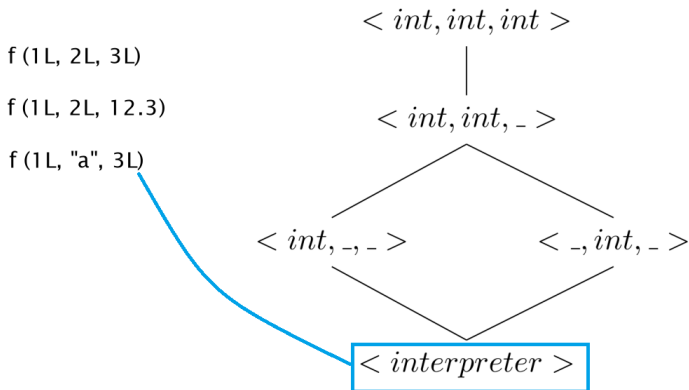
Contextual Dispatch



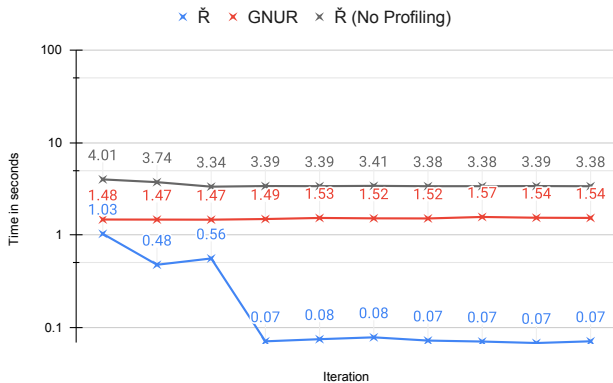
Contextual Dispatch



Contextual Dispatch



mandelbrot (shootout)



The regressions

reg-s4.R

- Normal GNUR: $\tilde{0.35}$ sec

The regressions

reg-s4.R

- Normal GNUR: $\tilde{0}.35$ sec
- \check{R} : **$\tilde{2}36$ sec**

The regressions

reg-s4.R

- Normal GNUR: *0.35 sec*
- R̃: **236 sec**

About the program.

- 11 lines of R code.

The regressions

reg-s4.R

- Normal GNUR: *0.35 sec*
- R: **236 sec**

About the program.

- 11 lines of R code.
- Nothing complex.

The regressions

reg-s4.R

- Normal GNUR: *0.35 sec*
- R̃: **236 sec**

About the program.

- 11 lines of R code.
- Nothing complex.
- Creates one class object.

The regressions

reg-s4.R

- Normal GNUR: *0.35 sec*
- R: **236 sec**

About the program.

- 11 lines of R code.
- Nothing complex.
- Creates one class object.

Let the debugging begin.

- **Data Collection:** What tools to use? GDB, Valgrind, rr?

The regressions

reg-s4.R

- Normal GNUR: $\tilde{0}.35$ sec
- R̃: **236** sec

About the program.

- 11 lines of R code.
- Nothing complex.
- Creates one class object.

Let the debugging begin.

- **Data Collection:** What tools to use? GDB, Valgrind, rr?
- **Hypothesis Testing:** Realistic insights, **solid evidence**.

The regressions

reg-s4.R

- Normal GNUR: *0.35 sec*
- R: **236 sec**

About the program.

- 11 lines of R code.
- Nothing complex.
- Creates one class object.

Let the debugging begin.

- **Data Collection:** What tools to use? GDB, Valgrind, rr?
- **Hypothesis Testing:** Realistic insights, **solid evidence**.
- **Insights:** Real research begins **here**.

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB
- Memory Leaks: Valgrind

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB
- Memory Leaks: Valgrind
- Wrong results: Compiler pass logs, rr

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB
- Memory Leaks: Valgrind
- Wrong results: Compiler pass logs, rr

The quick and dirty solutions are the best!

- **Trace visualizer:** R̃-viz (2̃ weeks)

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB
- Memory Leaks: Valgrind
- Wrong results: Compiler pass logs, rr

The quick and dirty solutions are the best!

- **Trace visualizer:** R-viz (2 weeks)
- **Runtime debuggers:** Rsh Dynamic Visualizer and Debugger (BTech MTP @ IITMandi)

Usually the hammer works

Existing state of the art: fast and reliable

- Segfaults: GDB
- Memory Leaks: Valgrind
- Wrong results: Compiler pass logs, rr

The quick and dirty solutions are the best!

- **Trace visualizer:** R-viz (2 weeks)
- **Runtime debuggers:** Rsh Dynamic Visualizer and Debugger (BTech MTP @ IITMandi)
- **Event querying:** General Event Query Engine (2 days)

A little bit of **React**, **C++**, **JS** and some **sockets**.

Query: What creates so many contexts; are they even useful?

Hypothesis: Contextual dispatch is a bad idea for the real world!

Query: What creates so many contexts; are they even useful?

Hypothesis: Contextual dispatch is a bad idea for the real world!

Ř-viz demo: <https://compl-research.github.io/r-viz/>

Query: What creates so many contexts; are they even useful?

Hypothesis: Contextual dispatch is a bad idea for the real world!

Ř-viz demo: <https://compl-research.github.io/r-viz/>

Reality: Number of contexts are usually manageable, but there are too many compilations!

Query: What creates so many contexts; are they even useful?

Hypothesis: Contextual dispatch is a bad idea for the real world!

Ř-viz demo: <https://compl-research.github.io/r-viz/>

Reality: Number of contexts are usually manageable, but there are too many compilations!

Insight: Compilation heuristic warrants deeper investigation.

Hypothesis: All compiled functions are equally responsible for peak performance.

Insight: Not always, sometimes inlining is dominant and most compilations are deprecated quickly.

Hypothesis: All compiled functions are equally responsible for peak performance.

Insight: Not always, sometimes inlining is dominant and most compilations are deprecated quickly.

Hypothesis: Compilation heuristics are buggy, leading to unnecessary compilations.

Insight: Yes, there is evidence to prove that.

Data Collection

Hypothesis: All compiled functions are equally responsible for peak performance.

Insight: Not always, sometimes inlining is dominant and most compilations are deprecated quickly.

Hypothesis: Compilation heuristics are buggy, leading to unnecessary compilations.

Insight: Yes, there is evidence to prove that.

Hypothesis: Contextual compilation sounds good in theory, but in reality, only a generic context becomes dominant.

Insight: No, in most cases the call-box shows that most contexts are used equally throughout the runtime.

Runtime Debugger

Goal: Learn about the **internal working** of the system.

Team: Ayush Sharma and Anmolpreet Singh (**just-in-graduation-time**).

Rsh Dynamic Visualizer and Debugger

Program is Running

Current Syn. Code: 0x55e267cb910. Type: BC

At function : "f"

```
0 idfun_g
5 [4,<1>,vaik0.0x55e2704379e8(closure)]
22 nk_promise_0
27 call_1
44 [DOUBLE(S)]
49 sivar_cached_a[0]
58 kvar_cached_a[0]
```

Keep bytecode sync

SCROLL ▶ NEXT ▶ STEP ▶ UPDATE TYPES ▶

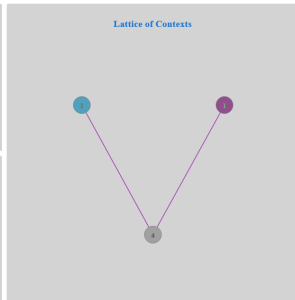
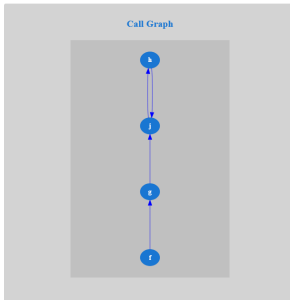
Source Code of current closure

```
(=(a, g(a)), =(res, +(a, b)), res)
```

Environment

Key	Value	DataType	Modify DataType	Modify Value
b	2L	int	Integer ▾	Value: 2L

UPDATE ENVIRONMENT ▶



Stack

```
4096
<prom val=2L 2L >
<prom val=1L 1L >
```

General Event Query Engine

Goal: A simple and fast query for runtime events.

General Event Query Engine

Goal: A simple and fast query for runtime events.

Motivation: Most things look right, but there is something suspicious.

General Event Query Engine

Goal: A simple and fast query for runtime events.

Motivation: Most things look right, but there is something suspicious.

GEQE: <https://meetesh06.github.io/General-Event-Query-Engine/>

General Event Query Engine

Browse... log-ser-6.json

REFRESH COLORS



Timeline contains 11522 events



Selected Event

▼ root: {} 9 keys

```
timestamp: 1680006137454545200
eventType: "dispatchTrying"
time: 0
pid: 490425
host: "NS base: 10599897219958198292"
hostOffset: "0"
inferred: "IExpMl,CorrOrd,ITMany,Argmatch,Eager2,NonRef0,NonRef1,NonRef2,IObj0,IObj1,IObj2,SimpleIn0,SimpleIn2 miss: 1"
context: "IExpMl,CorrOrd,ITMany,Argmatch,Eager2,NonRef1,NonRef2,IObj1,IObj2,SimpleIn2 miss: 1"
vtab: "0x55a4425024f0"
```

Query

Cheatsheet

Compilation time: $\$sum[timeline[eventType=="pirCompilation"].time]$

Number of X events: $\$sum[timeline[eventType=="X"]]$

Number of X events for where M=V: $\$sum[timeline[eventType=="X" and M="V"]]$

Give list of dispatch events that happened for before this compilation: $timeline[timestamp < TIMESTAMP and eventType=="dispatch" and hast=="HAS T"]$

See all dispatch events leading up to a compilation: $sort[timeline[timestamp <= 1679782893927526000 and eventType=="I2check" and hast=="NS.base:13807296501490167129"], function($i, $r) { $i.timestamp > $r.timestamp }]$

$sort[timeline[timestamp >= 1679782923452649000 and eventType=="I2check" and vtab="0x55a4425024f0"], function($i, $r) { $i.timestamp > $r.timestamp }]$

$timeline[vtab="0x55a4425024f0"]$

Show Auxiliary Timeline

EXECUTE



Timeline contains 212 events



Selected Event

The good

- Specialized tools are really useful in gaining meaningful insights.
- Mature frameworks like React are great way to write reusable code.

Conclusion

The good

- Specialized tools are really useful in gaining meaningful insights.
- Mature frameworks like React are great way to write reusable code.

The bad

- There are no one-size-fits-all solution for these problems.
- Things move fast and documentation is hard.

Conclusion

The good

- Specialized tools are really useful in gaining meaningful insights.
- Mature frameworks like React are great way to write reusable code.

The bad

- There are no one-size-fits-all solution for these problems.
- Things move fast and documentation is hard.

Road ahead

- These tools can be quickly retrofit to other use cases.

Dynamic languages are **great!**

Higher Order Functions

```
quickSort(data, predicate);
```

```
data = [  
  {  
    [k]: [v],  
    date: "5th Jan 2022"  
  },  
  {  
    [k]: [v],  
    date: "1st Jan 2022"  
  },  
]
```

```
p = (a, b) => a.date > b.date;  
g = (a) => a % 22;  
p = (a, b) => g(a) > g(b);
```

```
data = [1121, 2321, 12333, ...otherData]
```

Dynamic languages are **great!**

Dynamic Typing

```
f <- function(a, b) {  
  if (isS4(a, b)) {  
    add.s4(a.rep, b.rep);  
  } else if (isS3(a, b)){  
    add.s3(a, b)  
  } else {  
    add.default(a,b)  
  }  
}
```

$f: \square \times \square \rightarrow \square$

Dynamic Signature

Dynamic languages are **great!**

Parse and Eval

```
function parseAndEval(str) {  
  return Function(`${str}`)()  
}  
  
parseAndEval(getUserInput())  
// Output: 18446744073709552000
```

String

User Prompt

```
> a=2;  
  b=64;  
  return a**b;  
  
>
```

★ One line Interpreter