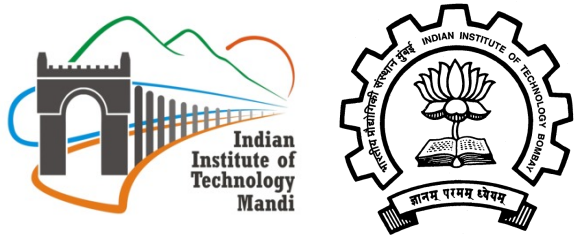
Debugging Dynamic Language Features in a Multi-Tier Virtual Machine

Anmolpreet Singh, Aayush Sharma, **Meetesh Kalpesh Mehta**, and Manas Thakur

IIT Mandi, IIT Bombay @ **VMIL '23**

October 23, 2023



CompL

Anmol

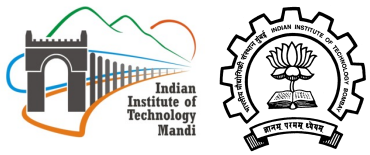
Aayush

Meetesh

Manas

Dynamic Languages are **great**!

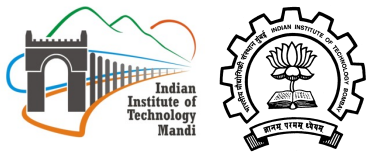
The **R** programming language, JS, lua...



Dynamic Languages are **great**!

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

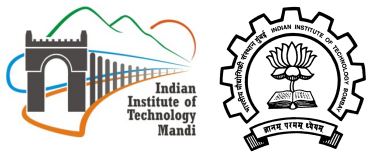
- Dynamic typing



Dynamic Languages are **great**!

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

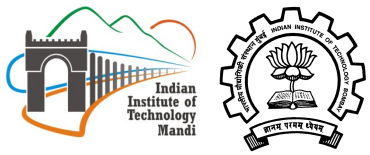
- Dynamic typing
- First-class functions/environments



Dynamic Languages are **great**!

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

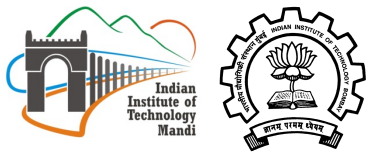
- Dynamic typing
- First-class functions/environments
- Lazy evaluation



Dynamic Languages are **great**!

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

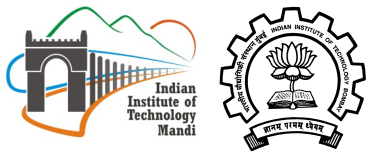
- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments



Dynamic Languages are **great**!

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

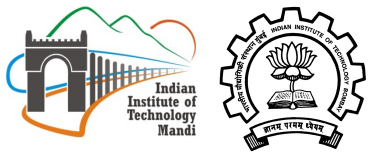
- Dynamic typing
- First-class functions/environments
- Lazy evaluation
- Runtime reification of environments
- Access to runtime stack



Dynamic Languages are **great**!

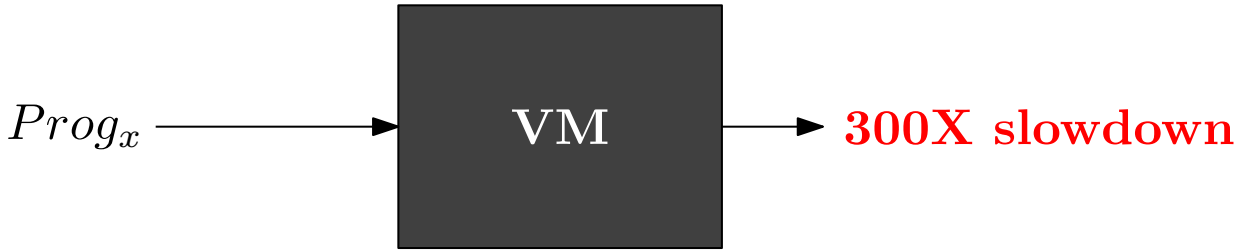
The **R** programming language, JS, lua...

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

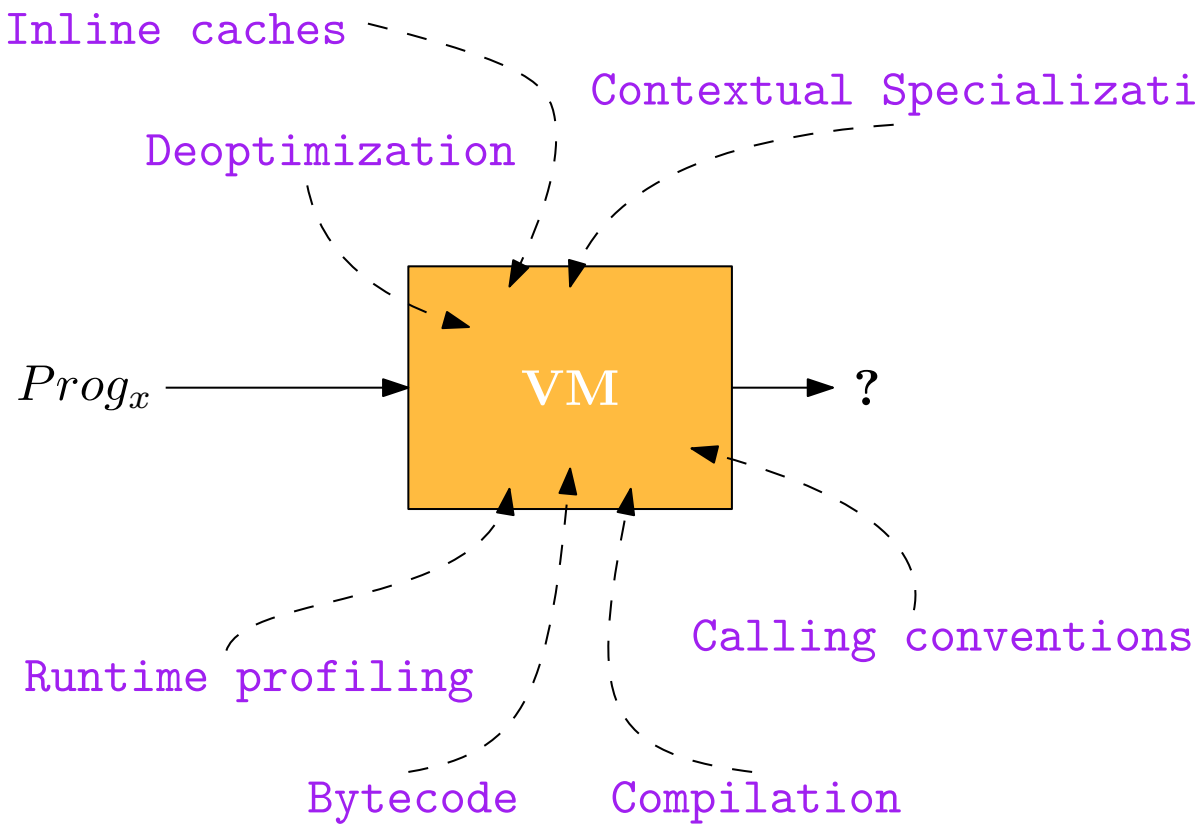


JITs under the hood

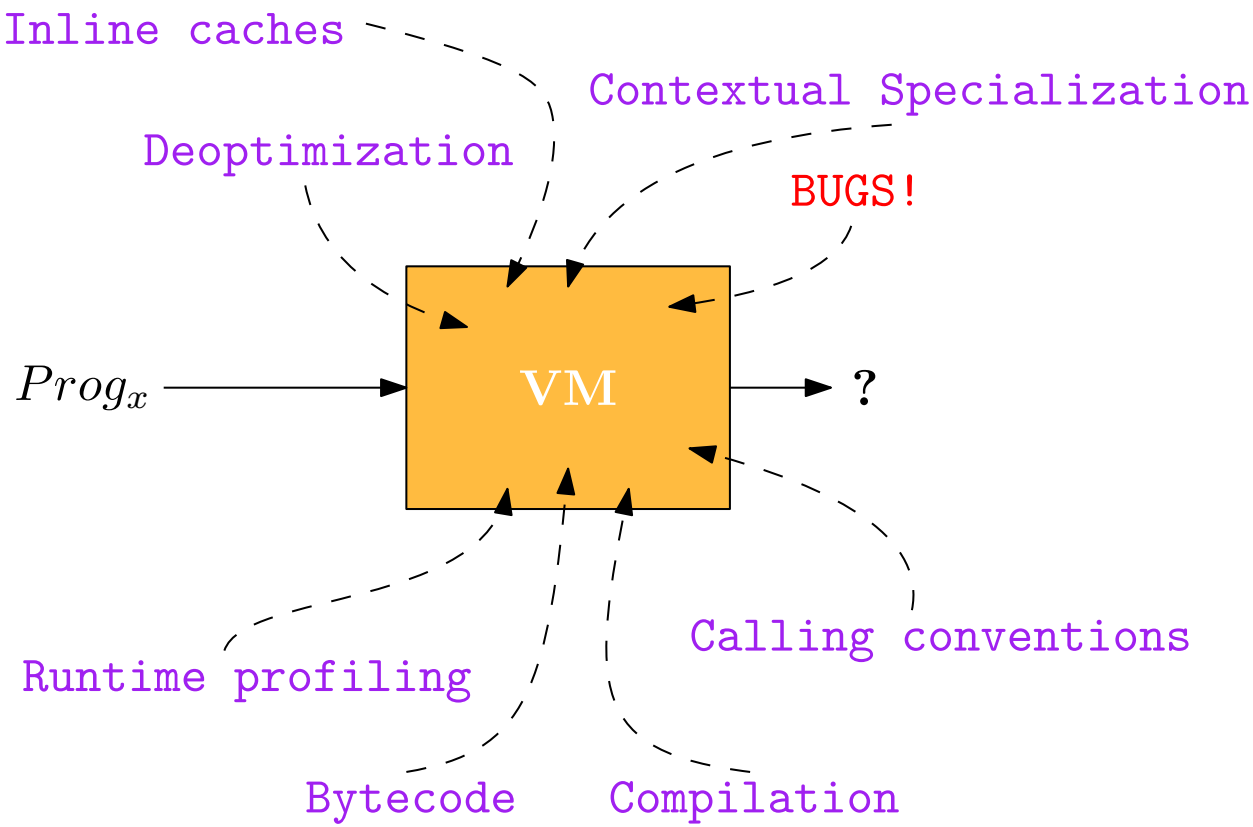
JITs under the hood



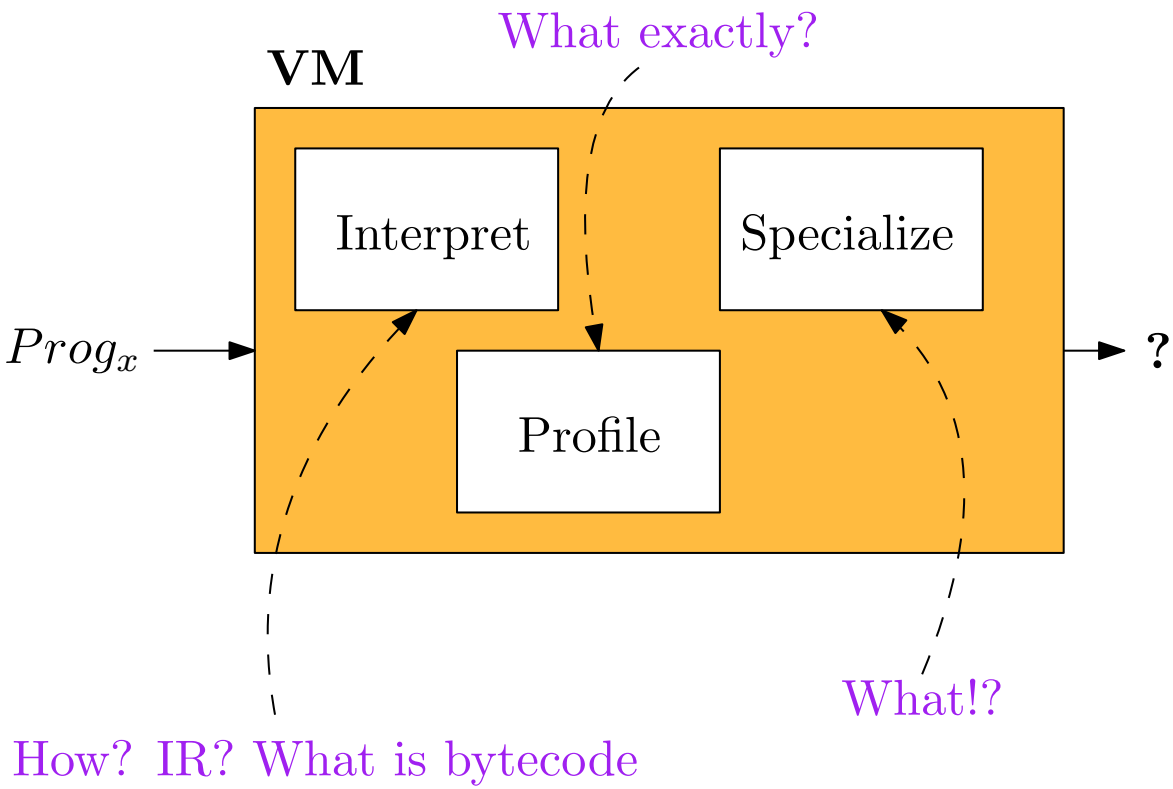
JITs under the hood



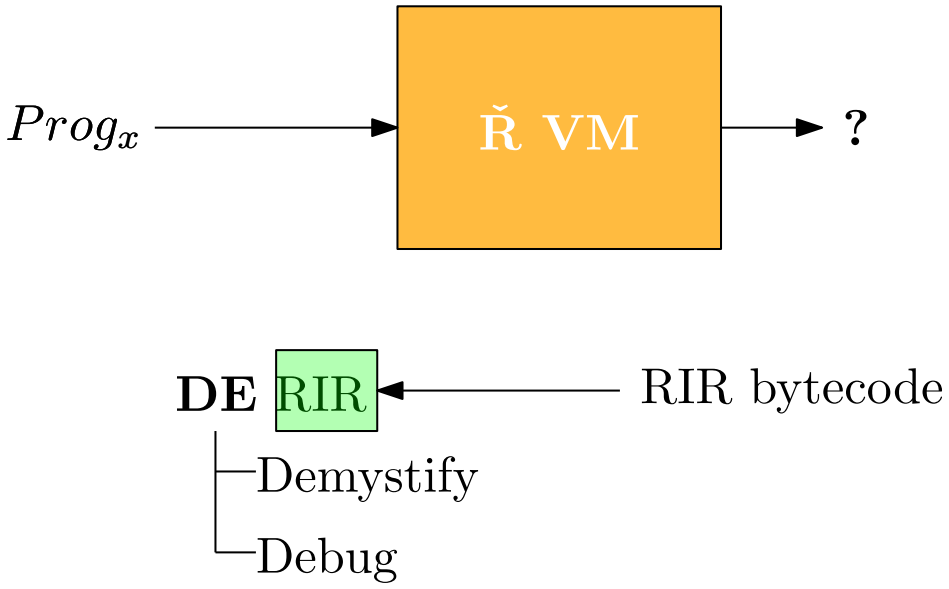
JITs under the hood



JITs under the hood



JITs under the hood

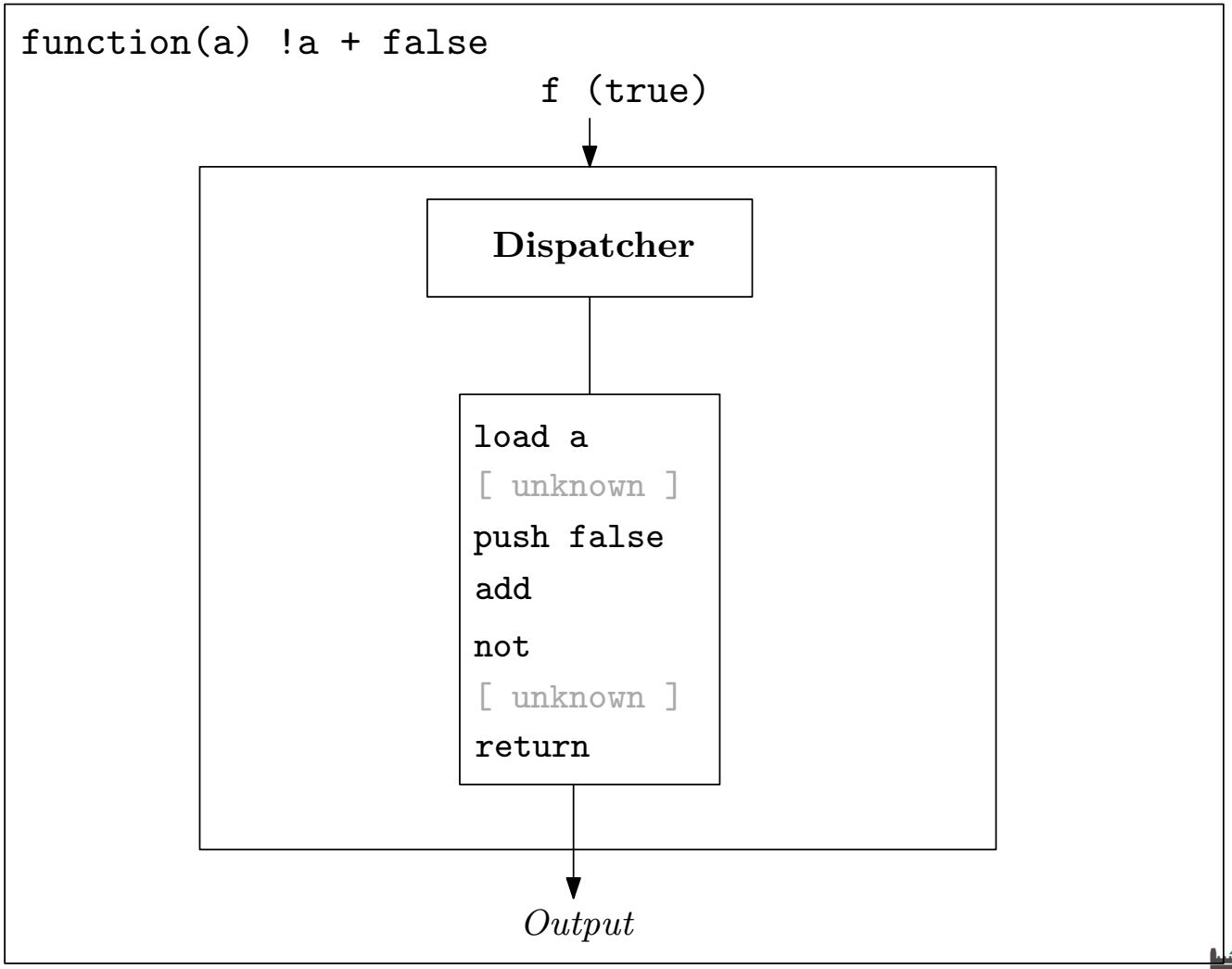


JITs under the hood

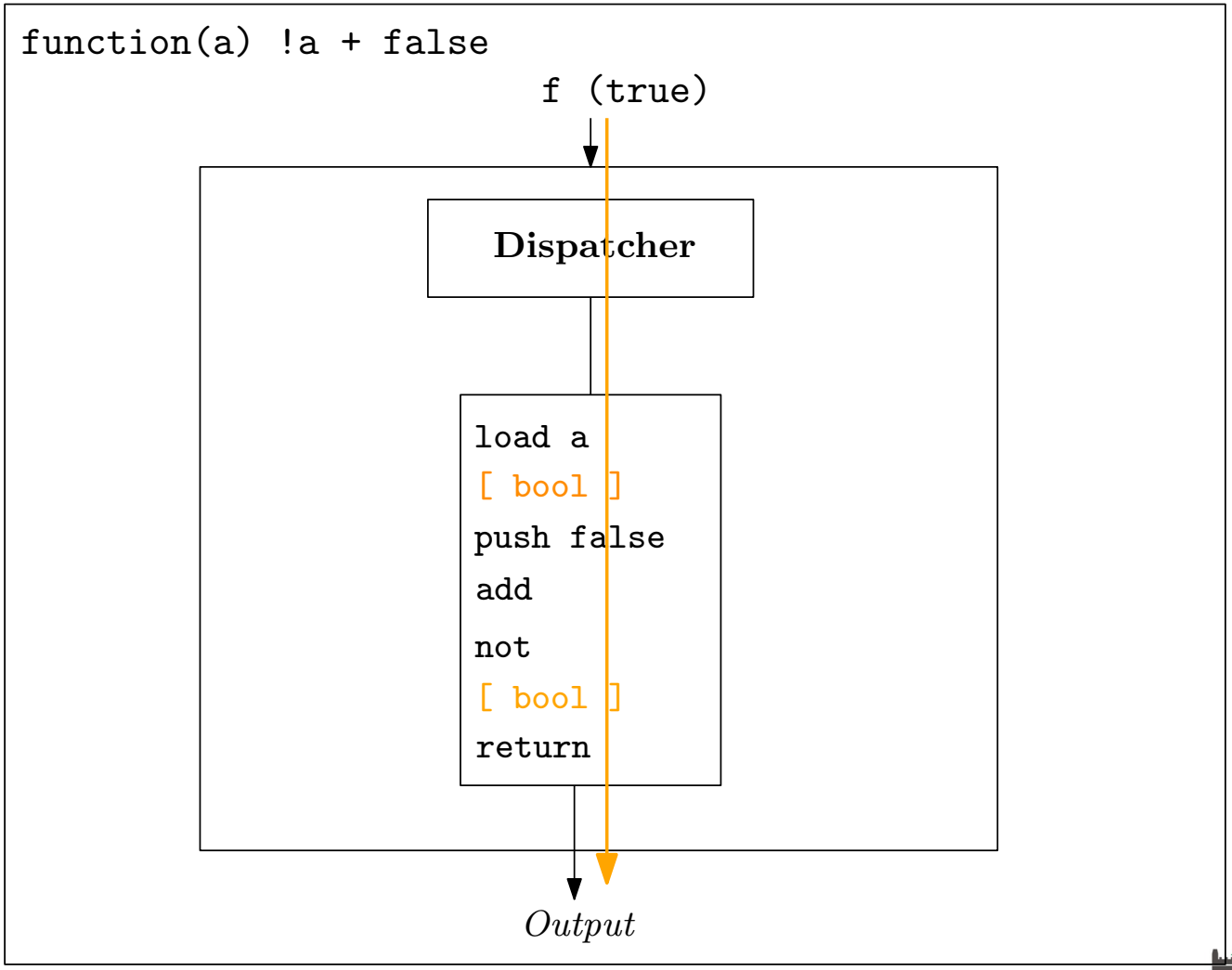
How is the user program interpreted?

```
# User Intention
expr = !a + a

# Unconditional true statement?

# JIT extension
!(a) + (a) # Wrong
!(a + a) # Right
```

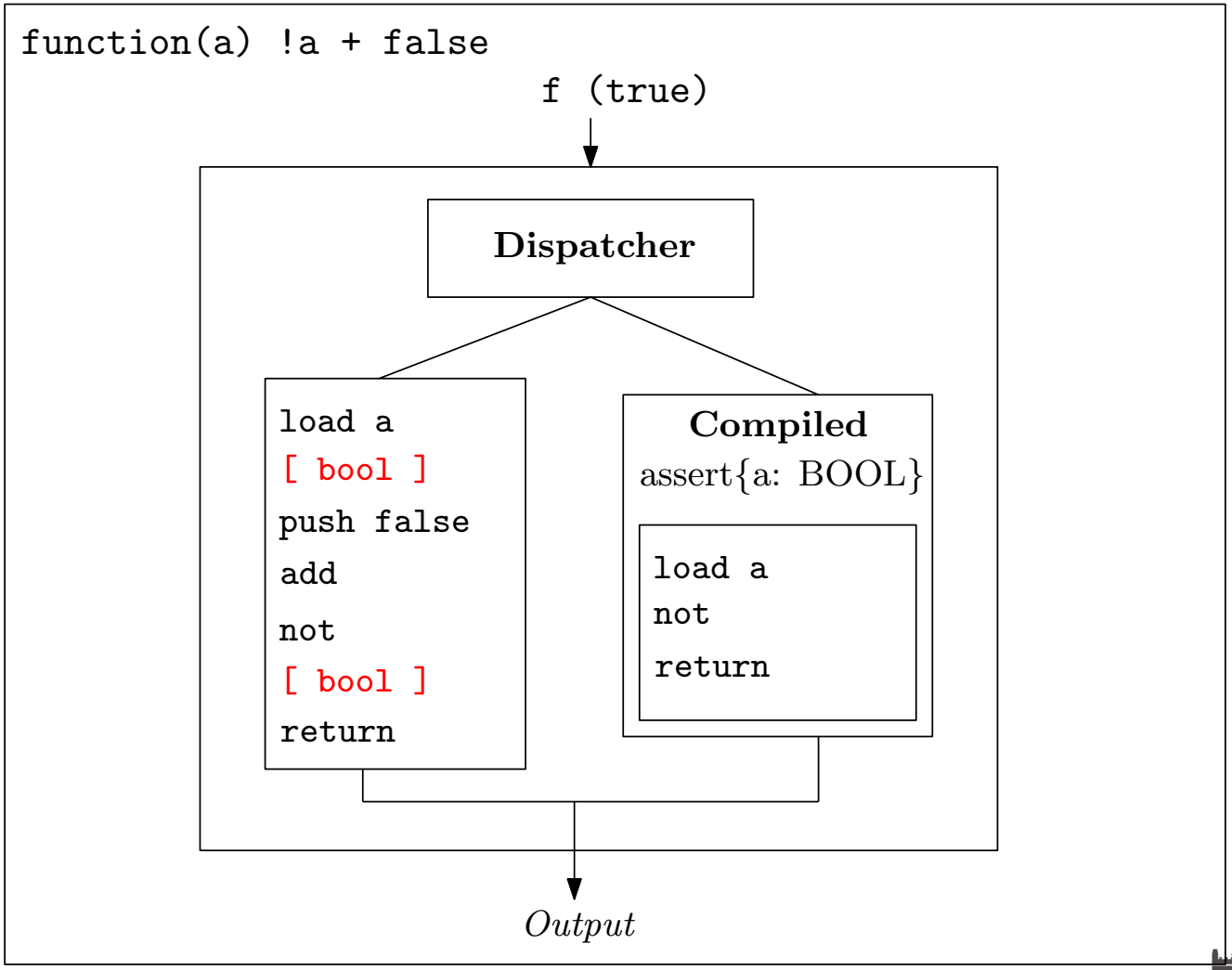
JITs under the hood



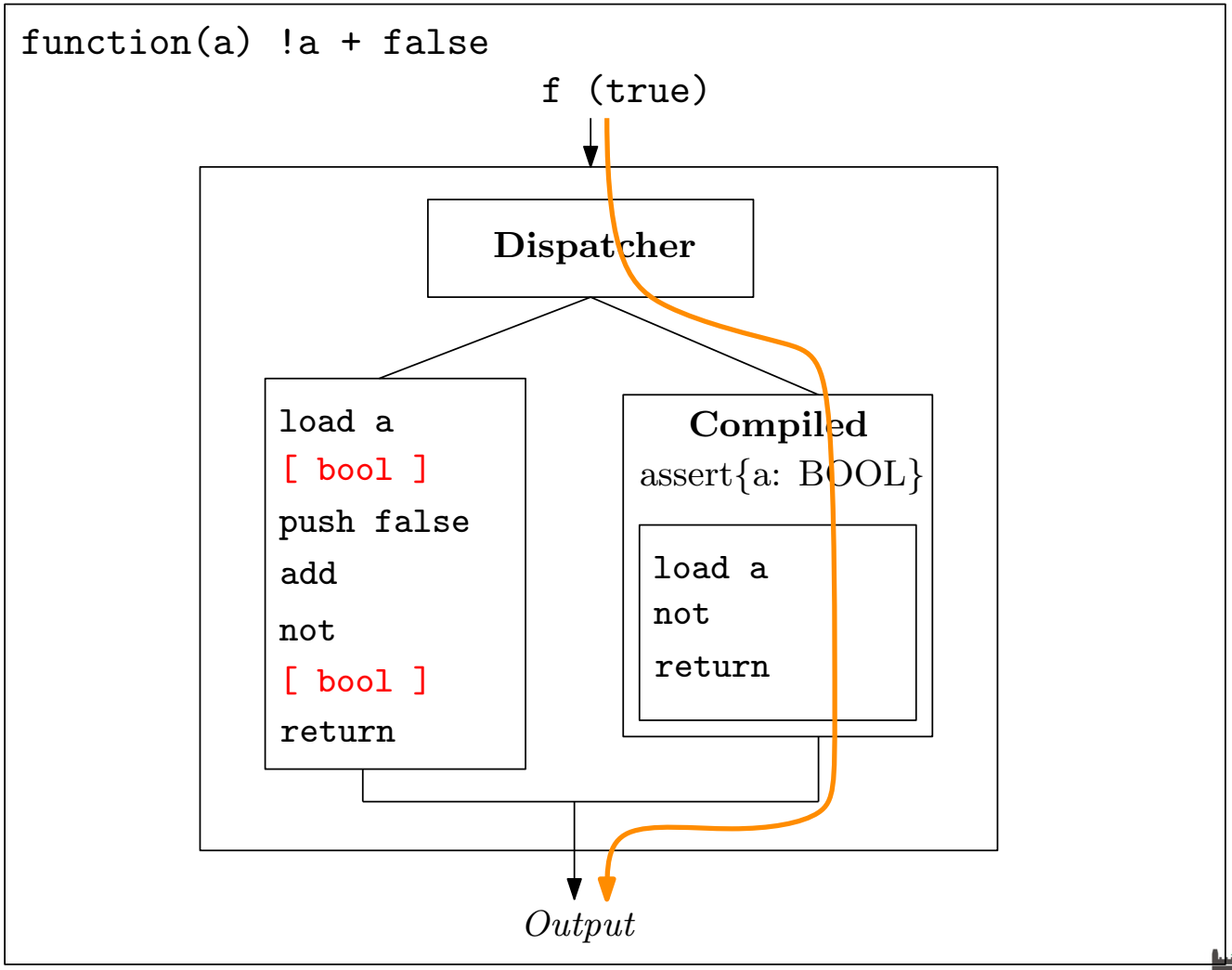
JITs under the hood



JITs under the hood



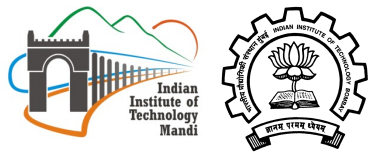
JITs under the hood



JITs under the hood

Strange control flows

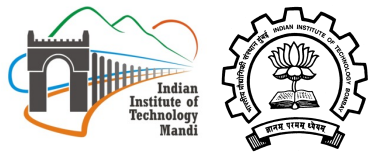
```
1  g <- function(b) { a <- b; print("g called"); }
2  f <- function(a) {
3     print("f called"); g(a);
4     print("f call end");
5  }
6  foo <- function() {
7     for (i in 1:20) {
8        if (i == 5) {
9           f(break)
10       }
11       print(i)
12    }
13 }
14 foo()
```



JITs under the hood

Strange control flows

```
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()
```



JITs under the hood

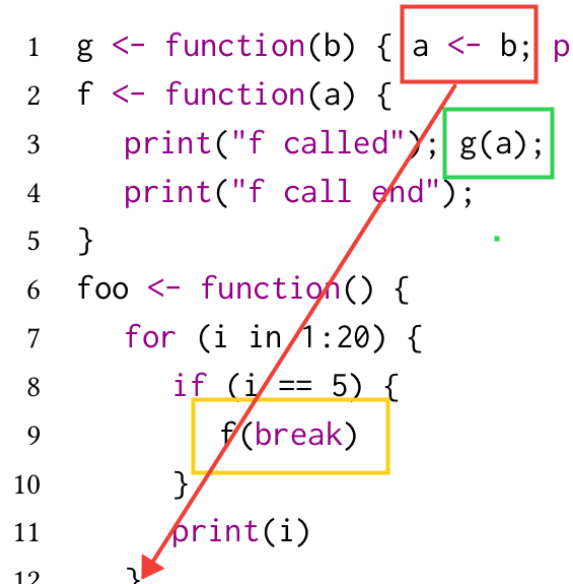
Strange control flows

```
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()
```

JITs under the hood

Strange control flows

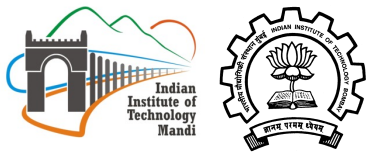
```
1  g <- function(b) { a <- b; print("g called"); }
2  f <- function(a) {
3     print("f called"); g(a);
4     print("f call end");
5  }
6  foo <- function() {
7     for (i in 1:20) {
8        if (i == 5) {
9           f(break)
10       }
11       print(i)
12    }
13 }
14 foo()
```



The $\check{\mathbf{R}}$ JIT compiler

Optimize for the most common cases.

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



The Ř JIT compiler

Optimize for the most common cases.

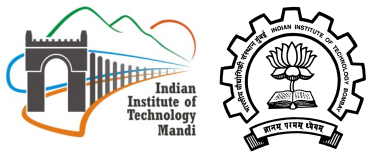
- Runtime profiling
  - Collect information about **types**, **call sites**, **branches**.
- Contextual Specialization [Flückiger et al. OOPSLA '20]

Contextual Dispatch for Function Specialization

OLIVIER FLÜCKIGER, Northeastern University, USA
GUIDO CHARI, ASAPP INC, Argentina
MING-HO YEE, Northeastern University, USA
JAN JEČMEN, Czech Technical University, Czechia
JAKOB HAIN, Northeastern University, USA
JAN VITEK, Northeastern University, USA and Czech Technical University, Czechia

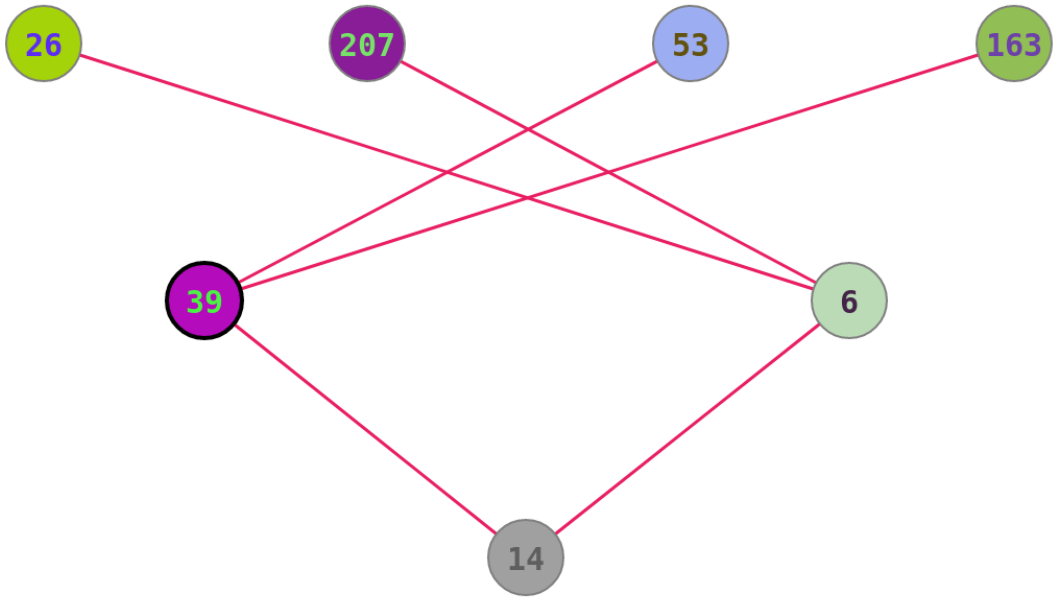
Regressions

Good performance in specialized functions, but...



Regressions

Good performance in specialized functions, but...



OOPSLA '23

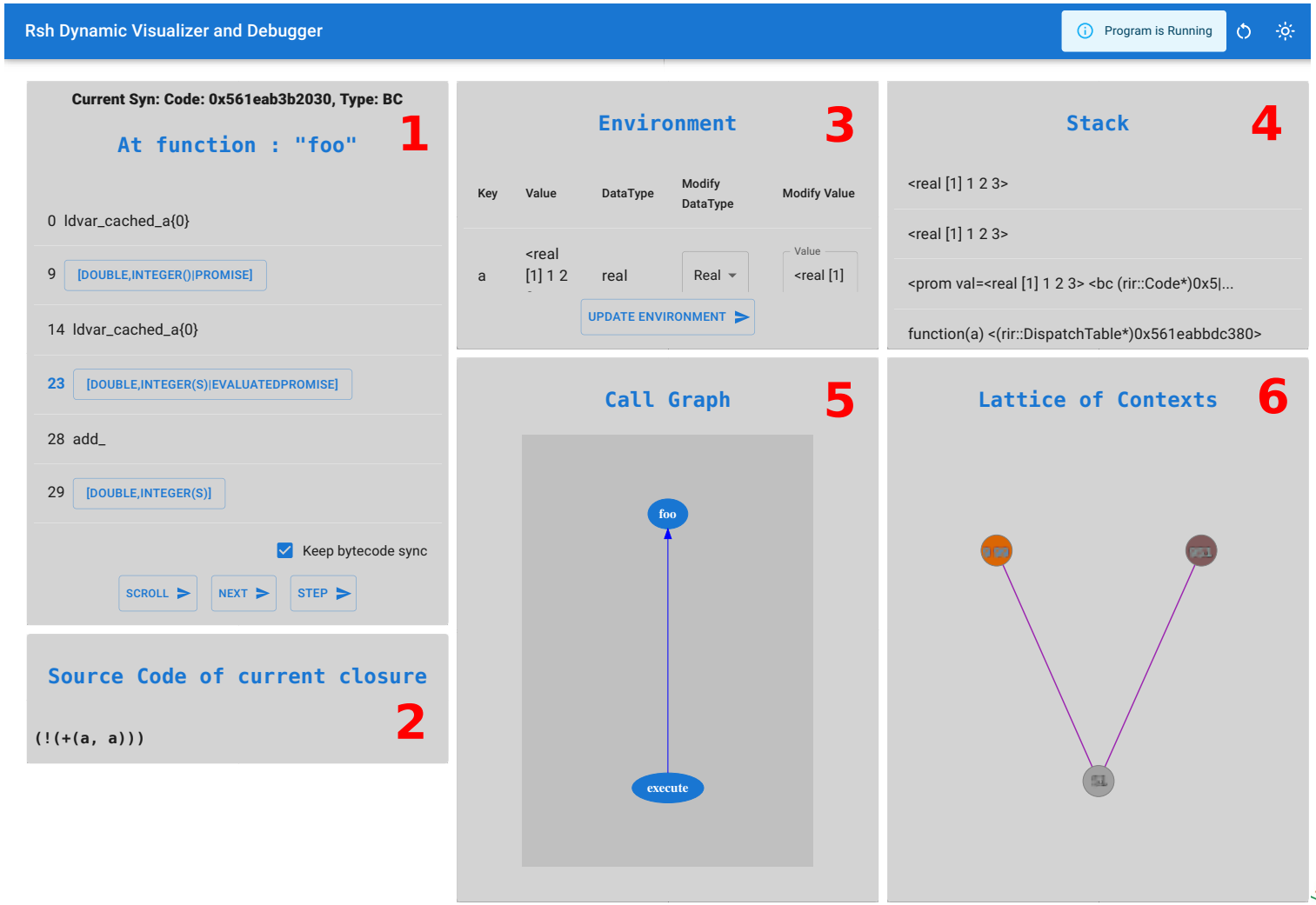
Reusing Just-in-Time Compiled Code

MEETESH KALPESH MEHTA, IIT Mandi, India
SEBASTIÁN KRYNSKI, Czech Technical University in Prague, Czechia
HUGO MUSSO GUALANDI, Czech Technical University in Prague, Czechia
MANAS THAKUR, IIT Bombay, India
JAN VITEK, Northeastern University, USA

Derir

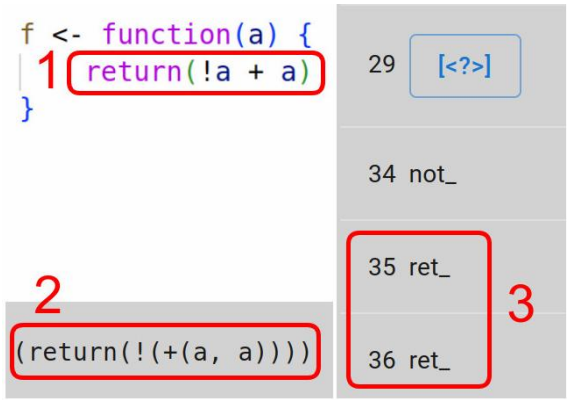
```
rir.viz("http://127.0.0.1:3011")
f <- function(a) { !a + a; }
f(1L)
f(c(1L,2L,3L))
f(1L)
f(FALSE)
```

Derir

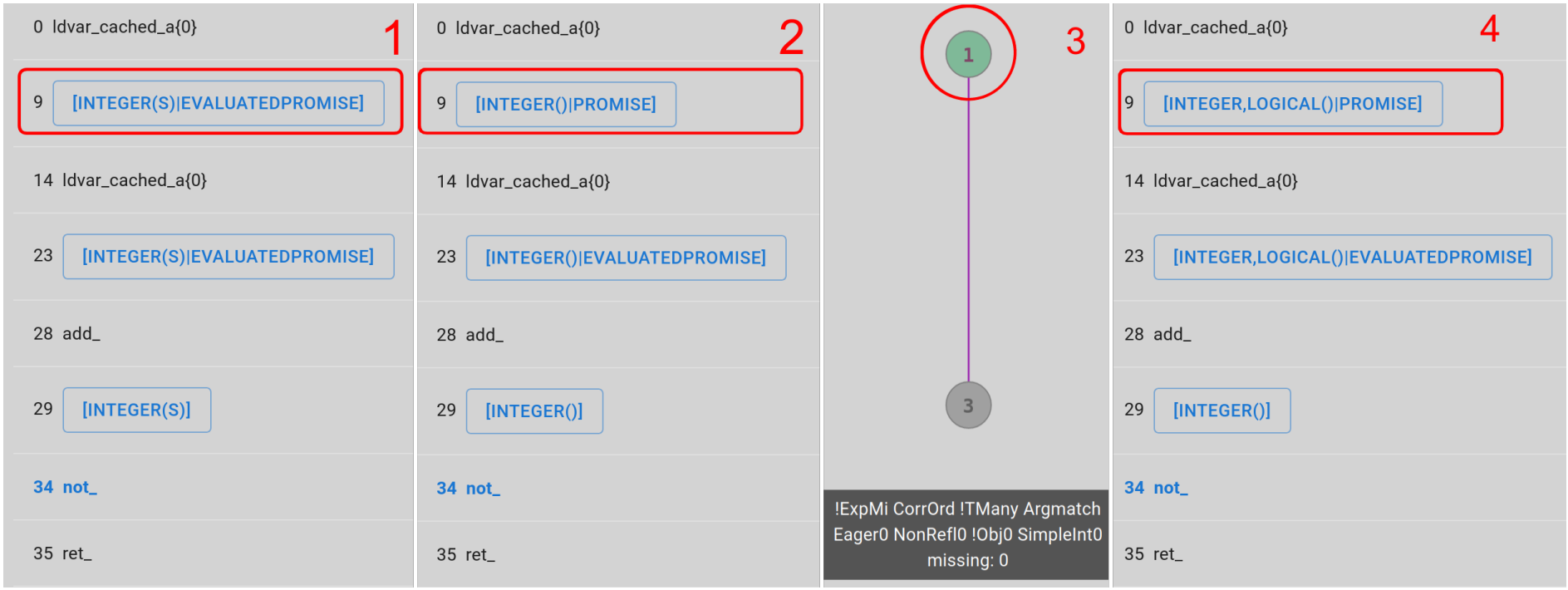

Rsh Dynamic Visualizer and Debugger
Program is Running
Current Syn: Code: 0x561eab3b2030, Type: BC
At function : "foo"
1
0 ldvar_cached_a{0}
9 [DOUBLE,INTEGER()|PROMISE]
14 ldvar_cached_a{0}
23 [DOUBLE,INTEGER(S)|EVALUATEDPROMISE]
28 add_
29 [DOUBLE,INTEGER(S)]
Keep bytecode sync
SCROLL
NEXT
STEP
Source Code of current closure
2
(!(+(a, a)))
Environment
3
Key
Value
DataType
Modify DataType
Modify Value
a
<real [1] 1 2
real
Real
Value
<real [1]
UPDATE ENVIRONMENT
Stack
4
<real [1] 1 2 3>
<real [1] 1 2 3>
<prom val=<real [1] 1 2 3> <bc (rir::Code*)0x5|...
function(a) <(rir::DispatchTable*)0x561eabbdc380>
Call Graph
5
foo
execute
Lattice of Contexts
6

Small bugs in the code emitter

Two returns?



The states


0 ldvar_cached_a{0}
1
9 [INTEGER(S)|EVALUATEDPROMISE]
14 ldvar_cached_a{0}
23 [INTEGER(S)|EVALUATEDPROMISE]
28 add_
29 [INTEGER(S)]
34 not_
35 ret_
0 ldvar_cached_a{0}
2
9 [INTEGER()|PROMISE]
14 ldvar_cached_a{0}
23 [INTEGER()|EVALUATEDPROMISE]
28 add_
29 [INTEGER()]
34 not_
35 ret_
1
3
!ExpMi CorrOrd !TMany Argmatch Eager0 NonRefl0 !Obj0 SimpleInt0 missing: 0
0 ldvar_cached_a{0}
4
9 [INTEGER,LOGICAL()|PROMISE]
14 ldvar_cached_a{0}
23 [INTEGER,LOGICAL()|EVALUATEDPROMISE]
28 add_
29 [INTEGER()]
34 not_
35 ret_

Feedback slots

Old:
00000000 00000000 00000000 0 0 0 0 11 01

New:
00000000 00000000 00000000 0 0 0 0 11 01

Field	CurrentValue	Modify
numTypes	**01**	1
stateBeforeLastForce	**11**	promise
Not Scalar	**0**	unset
Attribs	**0**	unset
Object	**0**	unset
Not Fast Vec Elt	**0**	unset
Seen 1	**00001110**	REALSXP

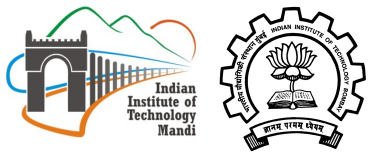
UPDATE

CLOSE

Conclusion

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.