# Optimizing R Language Execution via Aggressive Speculation

Lukas Stadler, **Adam Welc**, Christian Humer, Mick Jordan
Oracle Labs

ORACLE

# Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

# What is R?

- A programming language
  - Convenient tool for common statistical tasks
  - A DSL for statistics
  - A general-purpose language: ability to implement algorithms, analyses

- A data analysis workbench
  - Data exploration and manipulation
  - Graphics capabilities for visualizing data
  - Interactions with typesetting systems and web servers for data presentation

- A data science ecosystem
  - Over 11k open source packages for multiple purposes
  - Application areas: statistics, geoscience, bioinformatics, health sciences, machine learning, ...

ORACLE

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

function

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

function

function

ORACLE

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

function

function

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

function

function

- Functions can have side effects
  - Function re-definition
  - Search path alteration

ORACLE

# What is the challenge?

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

dynamic lookup

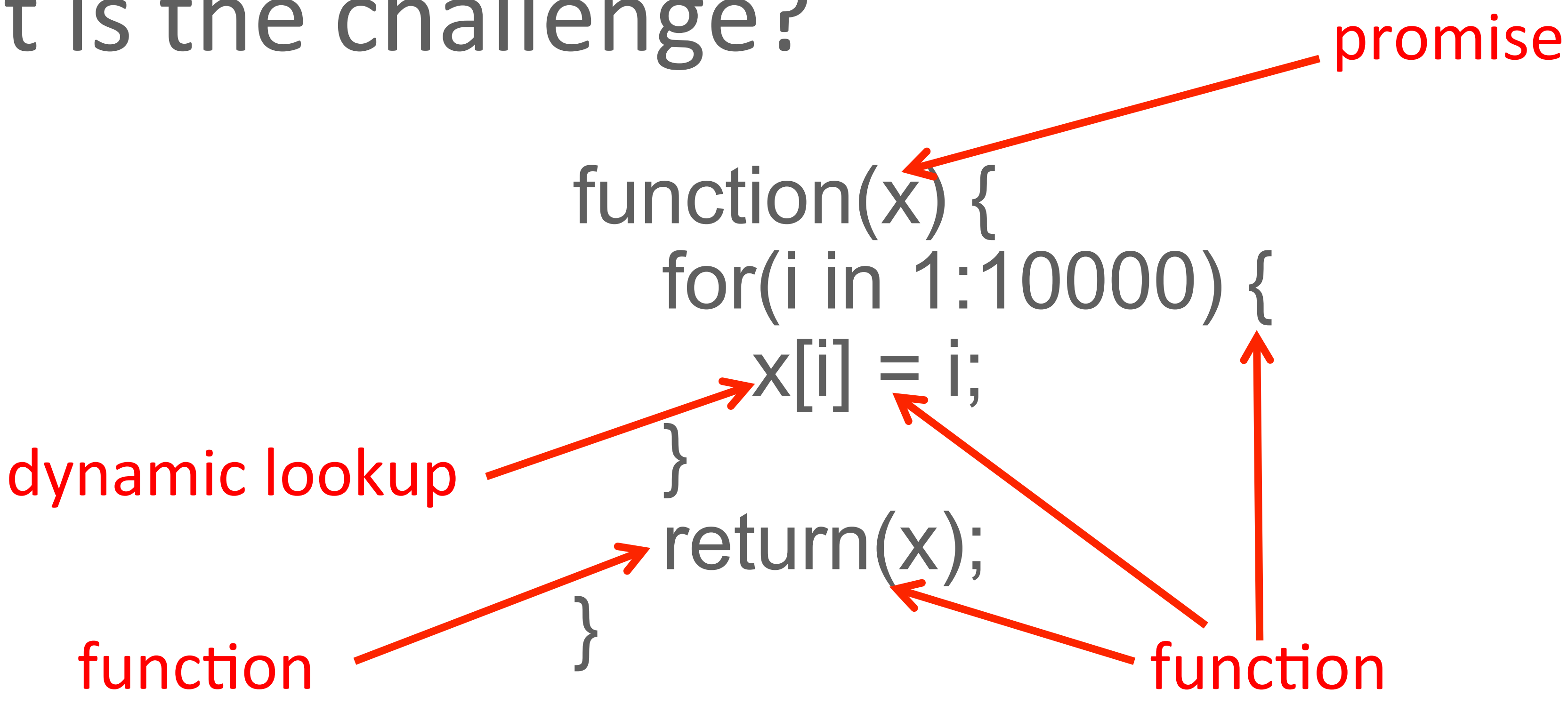
function

function

- Functions can have side effects
  - Function re-definition
  - Search path alteration

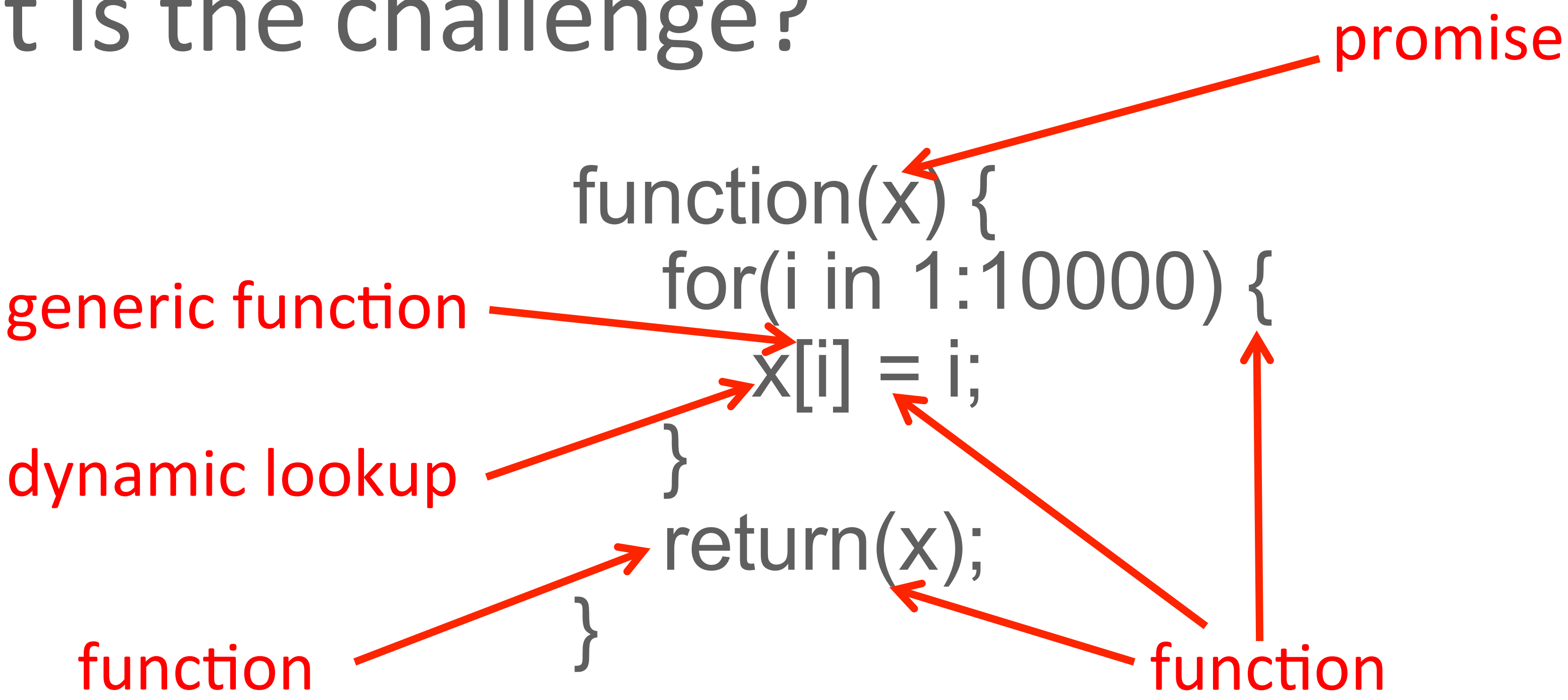
ORACLE

# What is the challenge?




- Functions can have side effects
  - Function re-definition
  - Search path alteration

# What is the challenge?




- Functions can have side effects
  - Function re-definition
  - Search path alteration

# What is the challenge?

promise

generic function

dynamic lookup

```
function(x) {
    for(i in 1:10000) {
        x[i] = i;
    }
    return(x);
}
```

function

function

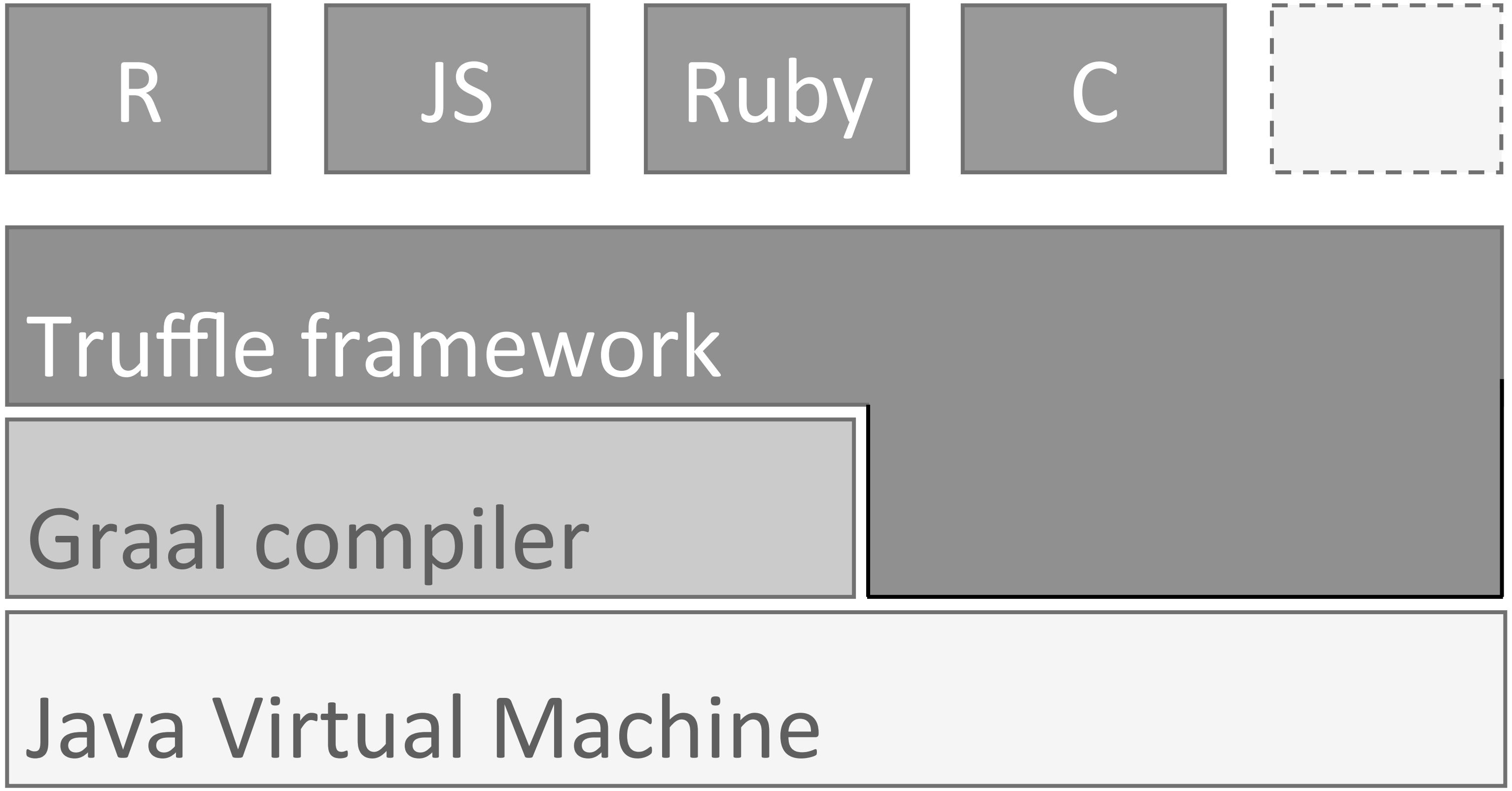
- Functions can have side effects
  - Function re-definition
  - Search path alteration
- Generic function’s dispatch depends on x’s metadata (attributes)

# What is FastR?

- An alternative R execution engine, developed under GPL v2 at Oracle Labs in collaboration with the academia
  - Started with Jan Vitek's group at Purdue
- Drop-in, fully compatible replacement for R's reference implementation GNU R
- Focused on improving performance of long-running R code
- Open-source: https://github.com/graalvm/fastr

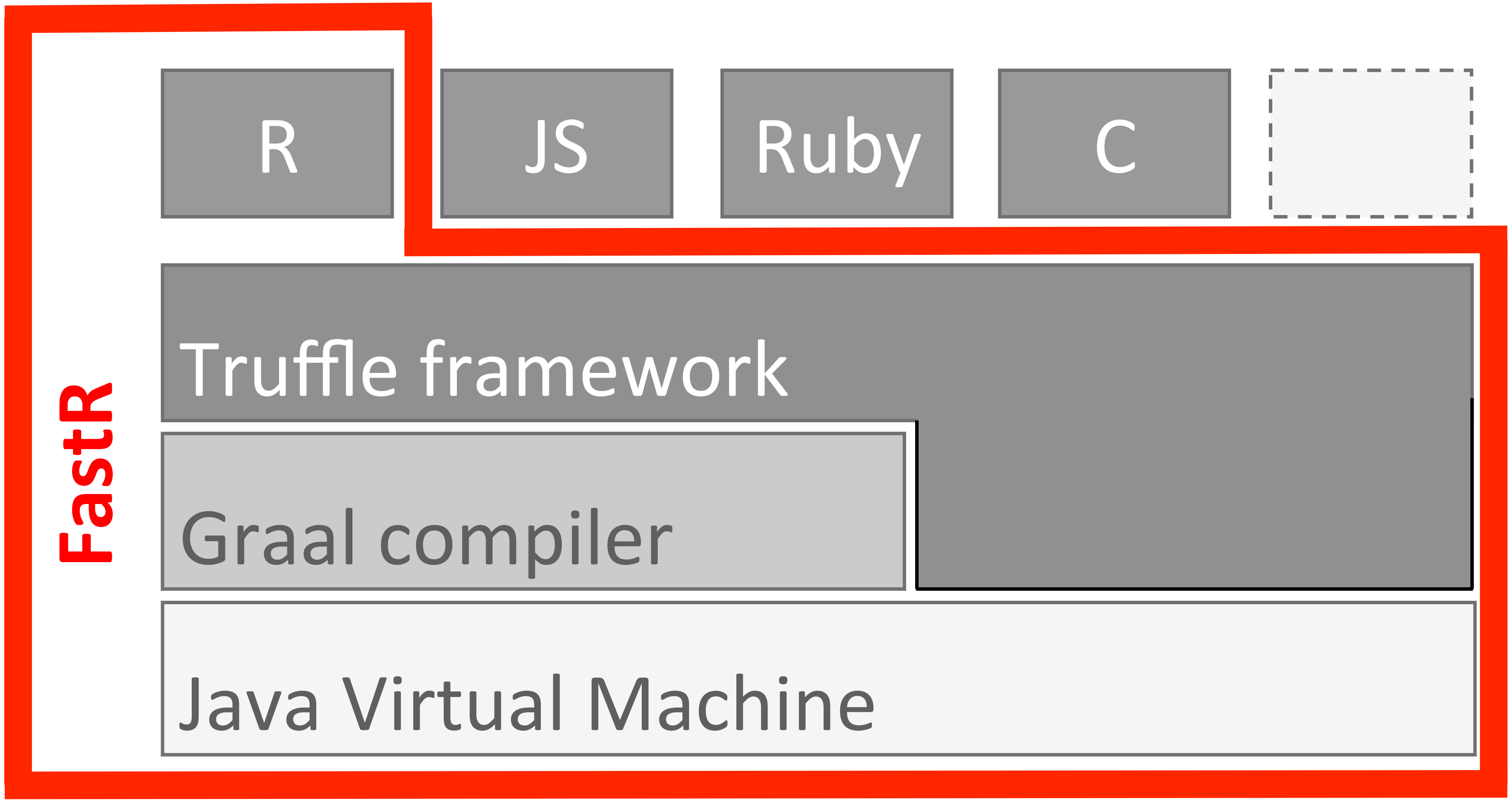
ORACLE

# System architecture




ORACLE

# System architecture

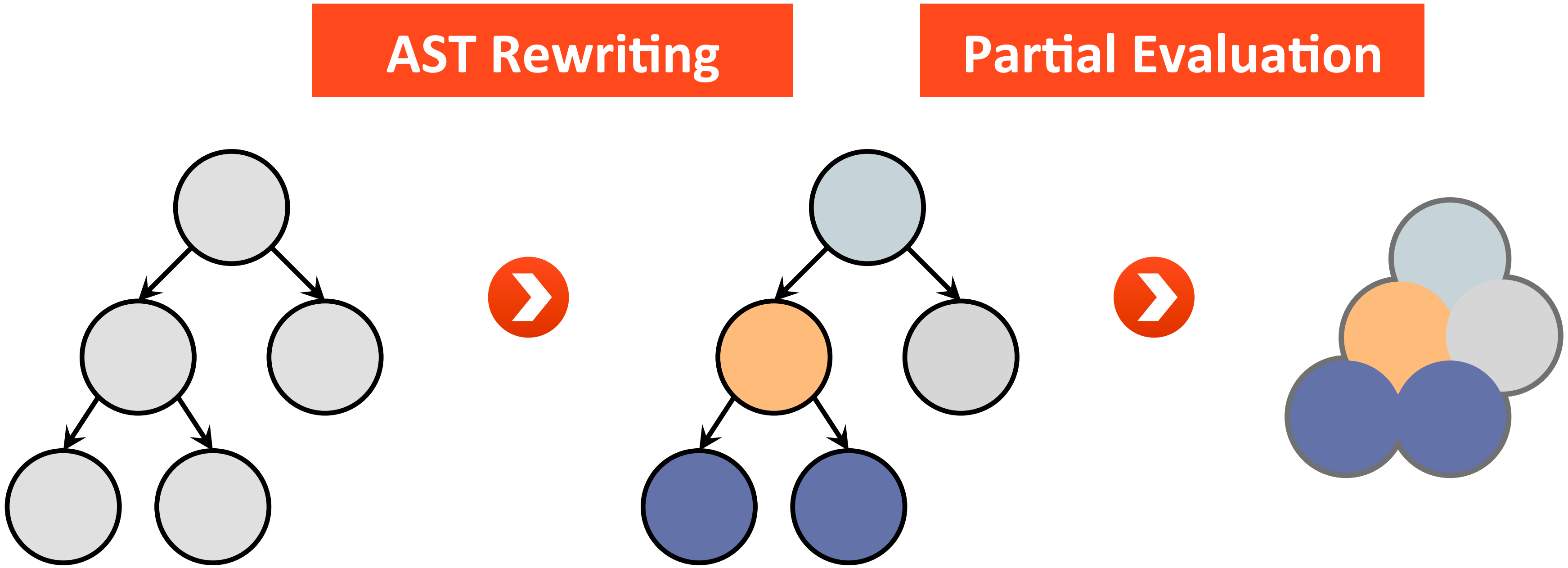


# Graal/Truffle technology stack

- Main components
  - Truffle framework to build Abstract Syntax Tree interpreters
  - Single Graal compiler to generate native code for all Truffle languages
- Competitive in peak performance to best-of-class of each language:
  - for Java (vs. HotSpot server compiler)
  - for dynamic languages (vs. V8)
  - for static languages (prototype, vs. GCC)
- Open source: https://github.com/graalvm

ORACLE

# From interpreted AST to native code




Interpreted AST (responsibility of the language creator)

Specialized AST

Compiled Code (guards for deoptimization)

# FastR – R as a Truffle language

- Superior performance without resorting to C and Fortran
  - Significant amounts of time are spent converting R to C code for performance
- Interoperability within the Graal/Truffle ecosystem
  - Transparent interop with JS, C, Ruby, …
- Research vehicle for data-heavy and parallel applications
  - Multi-tenancy, multi-thread execution of R applications, alternative internal data representations, etc.

ORACLE

# Optimizing R

- Three fundamental optimization techniques
  - Caching: inline caches for function calls, but also caching information for other operations (e.g. argument matching)
  - Assumptions: used to monitor low probability events – costly invalidations but inexpensive (with system support) to check
  - Specialization: divide implementation of an operation into smaller pieces and speculate that only a limited set of code paths will be taken
- These techniques permeate the entire implementation: symbol lookup, function calls, lazy evaluation, vector accesses, etc.

# Example: lazy evaluation

- R uses a *call-by-need* lazy argument evaluation strategy
  - Each argument is a *promise* (code snippet + evaluation environment)
  - Argument value is computed (promise is *forced*) as late as possible and only if needed
- Problems
  - Promise creation and indirect argument value access incur overhead
  - Environments (variable/value mappings) are virtualized (into native stack frames) – storing them requires materialization and is expensive
  - Each program point where promise can be forced becomes a call site

ORACLE

# Specialized promise implementation categories

- Eager promises – local variable used as parameter

```
x = 42; foo(x);         # extensible to include pure function calls as params
```

- Indirect promises – non-forced parameters passed to other calls

```
bar = function(x) { foo(x); }
```

- Default promises – arbitrarily complex code to be evaluated

```
foo(x + bar(y))
```

ORACLE

# Lazy vs. eager evaluation

global environment

# Lazy vs. eager evaluation

global environment

lazy

```
foo = function(a) {
  x <<- 7;
  print(a);
}
```

# Lazy vs. eager evaluation

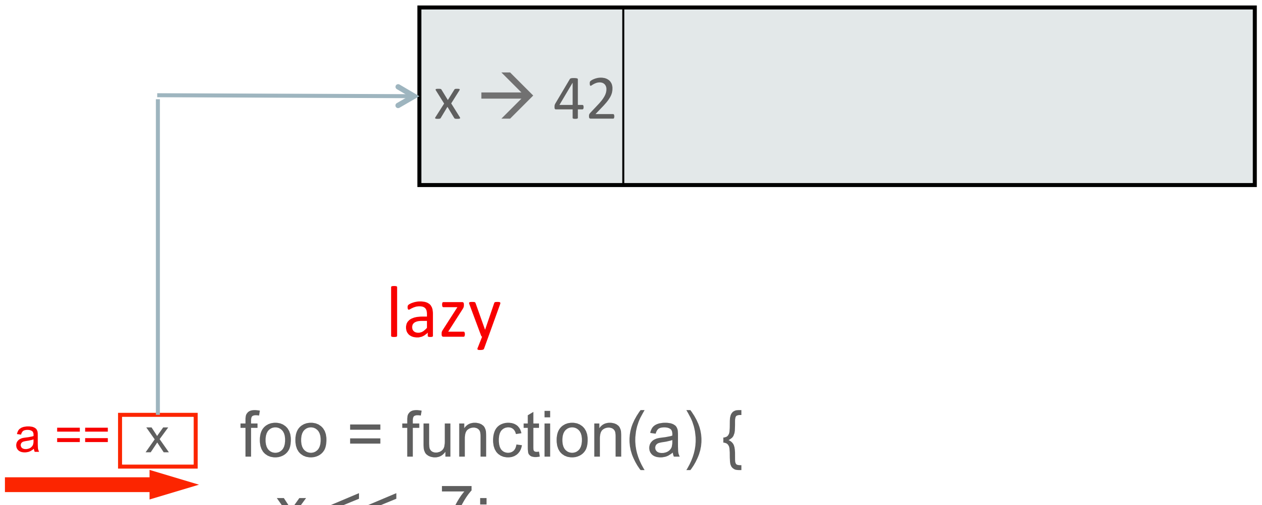

global environment

lazy

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
```

# Lazy vs. eager evaluation




global environment

lazy

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);
```

# Lazy vs. eager evaluation


global environment

lazy

a == x

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);
```

# Lazy vs. eager evaluation

x → 7

global environment

lazy

a == 7

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);
```

# Lazy vs. eager evaluation


global environment

lazy

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

# Lazy vs. eager evaluation

| x → 7 | |
|---|---|

global environment

**lazy**

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

**eager**

```
bar = function(b) {
  y <<- 7;
  print(b);
}
```

# Lazy vs. eager evaluation

| x → 7 | y → 42 | |
|---|---|---|

global environment

**lazy**

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

**eager**

```
bar = function(b) {
  y <<- 7;
  print(b);
}

y = 42;
```

# Lazy vs. eager evaluation

| x → 7 | y → 42 | |
|---|---|---|

global environment

lazy

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

eager

b == 42 →

```
bar = function(b) {
  y <<- 7;
  print(b);
}

y = 42;
bar(y);
```

# Lazy vs. eager evaluation

| | | |
|---|---|---|
| x → 7 | y → 7 | |

global environment

**lazy**

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

**eager**

b == 42 →

```
bar = function(b) {
  y <<- 7;
  print(b);
}

y = 42;
bar(y);
```

ORACLE

# Lazy vs. eager evaluation

| x → 7 | y → 7 | |
|---|---|---|

global environment

**lazy**

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);     # prints 7
```

**eager**

b == 42 →

```
bar = function(b) {
  y <<- 7;
  print(b);
}

y = 42;
bar(y);
```

ORACLE

# Lazy vs. eager evaluation

| x → 7 | y → 7 | |
|---|---|---|

global environment

**lazy**

```
foo = function(a) {
  x <<- 7;
  print(a);
}

x = 42;
foo(x);      # prints 7
```

**eager**

b == 42

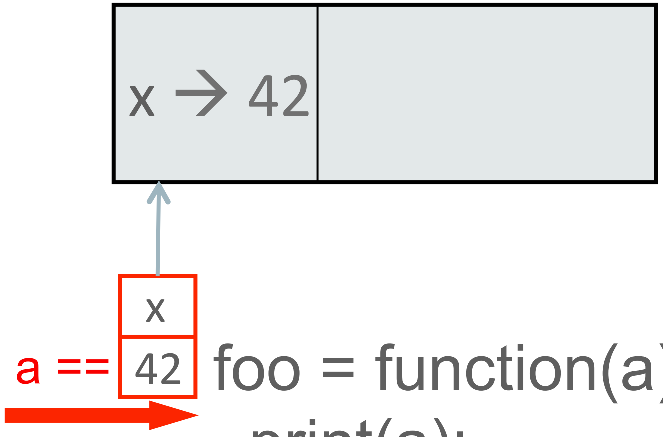
```
bar = function(b) {
  y <<- 7;
  print(b);
}

y = 42;
bar(y);      # prints 42
```

ORACLE

# Eager promises implementation

- Promise caches eager value

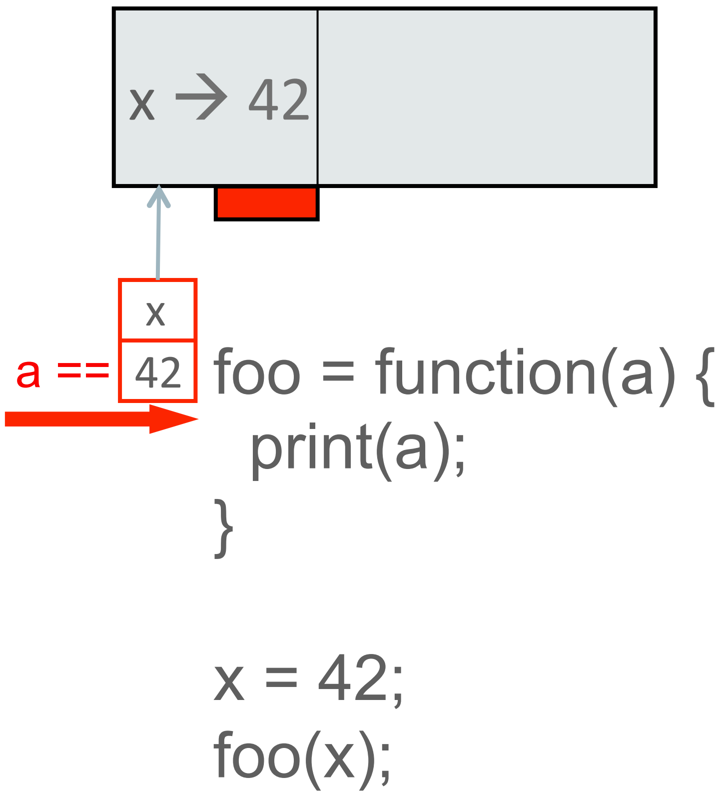



```
foo = function(a) {
  print(a);
}

x = 42;
foo(x);
```

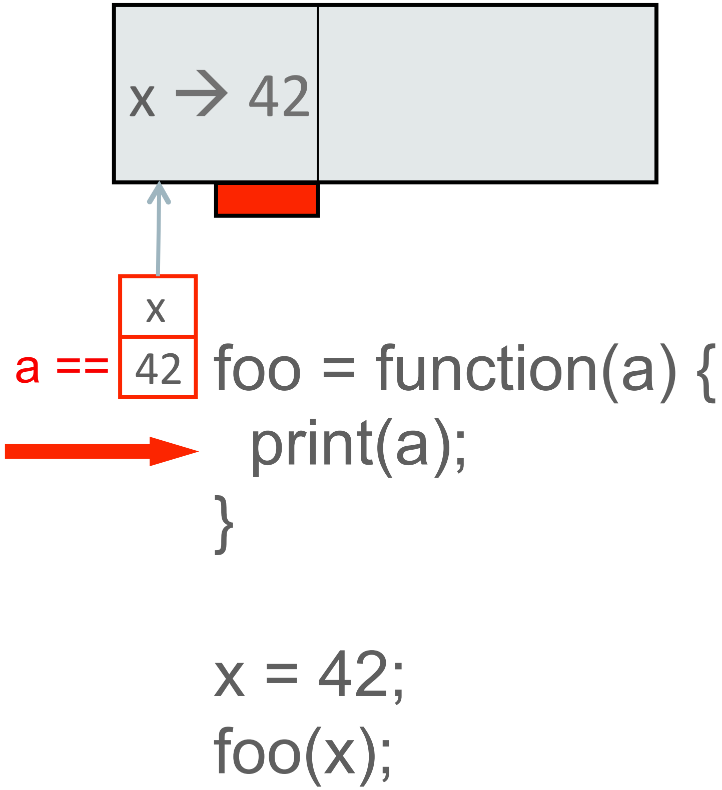
# Eager promises implementation

- Promise caches eager value
- Truffle assumption associated with an environment slot to monitor updates



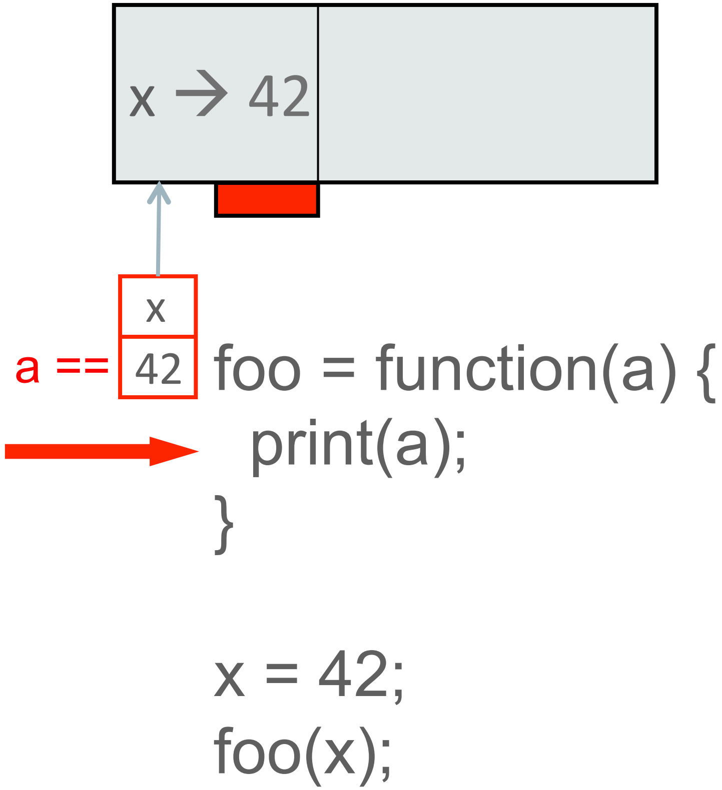
# Eager promises implementation

- Promise caches eager value
- Truffle assumption associated with an environment slot to monitor updates
  - Assumption checked before argument a is used for the first time
    - If valid – use cached value
    - If invalid – re-evaluate promise



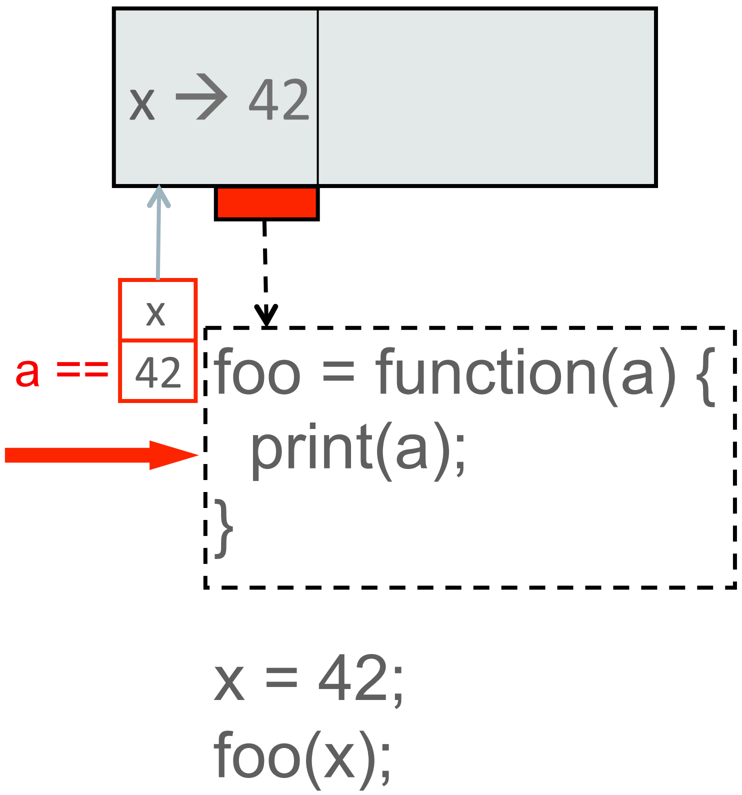
# Eager promises implementation

- Promise caches eager value
- Truffle assumption associated with an environment slot to monitor updates
  - Assumption checked before argument a is used for the first time
    - If valid – use cached value
    - If invalid – re-evaluate promise
  - No-cost assumption check in compiled code



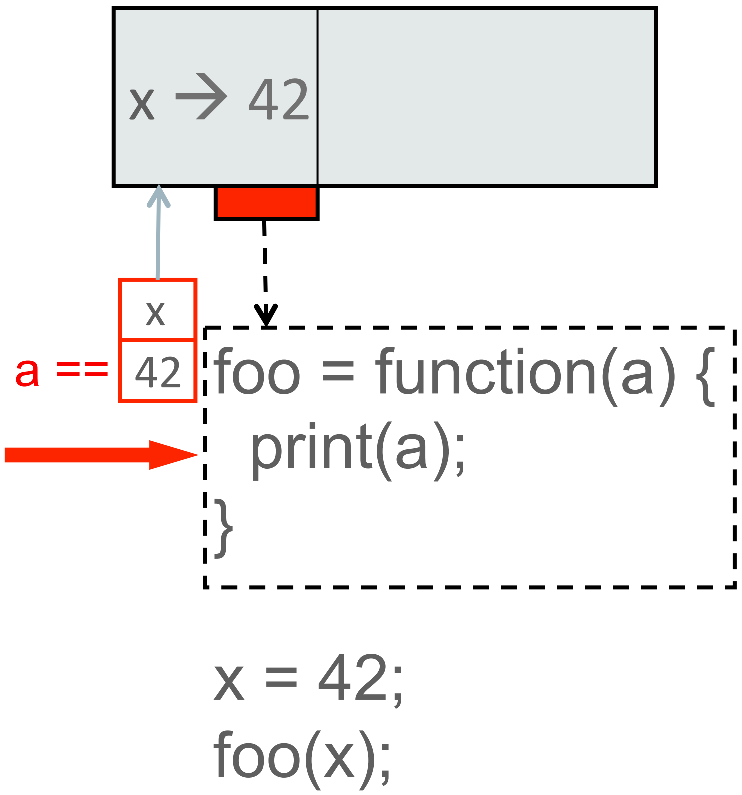
# Eager promises implementation

- Promise caches eager value
- Truffle assumption associated with an environment slot to monitor updates
  - Assumption checked before argument a is used for the first time
    - If valid – use cached value
    - If invalid – re-evaluate promise
  - No-cost assumption check in compiled code
    - Assumption “knows” the code to invalidate if needed



# Eager promises implementation

- Promise caches eager value
- Truffle assumption associated with an environment slot to monitor updates
  - Assumption checked before argument a is used for the first time
    - If valid – use cached value
    - If invalid – re-evaluate promise
  - No-cost assumption check in compiled code
    - Assumption “knows” the code to invalidate if needed
  - Compiler can unbox cached eager value



# One more problem…

- We don’t want a pointer to environment (to allow virtualization)

ORACLE

# One more problem…

- We don't want a pointer to environment (to allow virtualization)
- Fortunately, environments can be counted!

```
foo = function(a) {
  print(a);
}
bar = function(a) {
  foo(a);
}

x = 42;
bar(x)
```

# One more problem…

- We don’t want a pointer to environment (to allow virtualization)
- Fortunately, environments can be counted!

```
foo = function(a) {
  print(a);
}
bar = function(a) {
  foo(a);
}

x = 42;
bar(x)
```

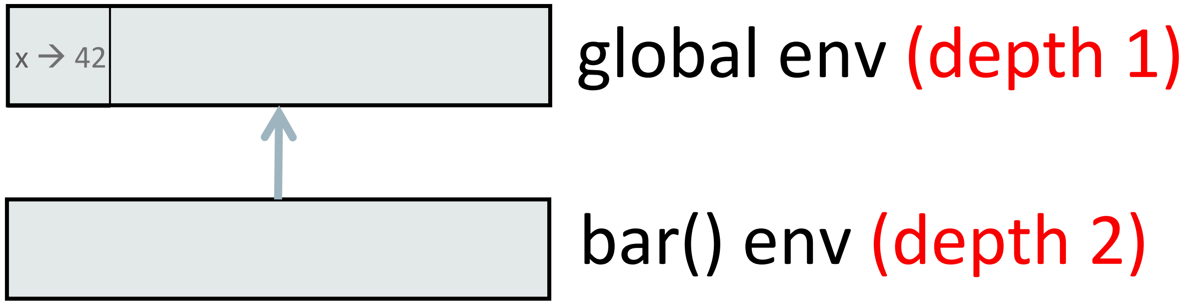

global env (depth 1)

# One more problem…

- We don’t want a pointer to environment (to allow virtualization)
- Fortunately, environments can be counted!

```
foo = function(a) {
  print(a);
}
bar = function(a) {
  foo(a);
}

x = 42;
bar(x)
```

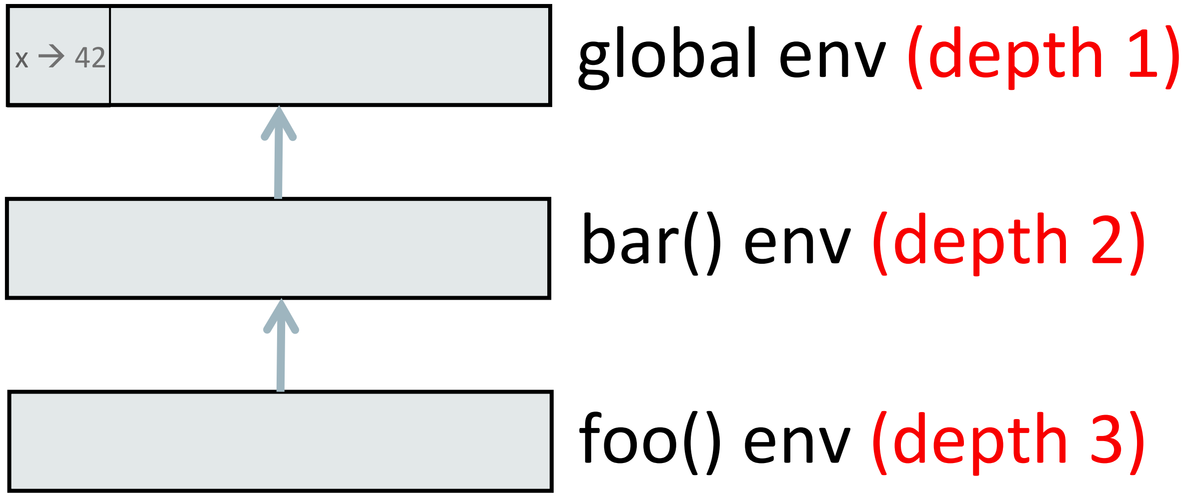


# One more problem…

- We don’t want a pointer to environment (to allow virtualization)
- Fortunately, environments can be counted!

```
foo = function(a) {
  print(a);
}
bar = function(a) {
  foo(a);
}

x = 42;
bar(x)
```

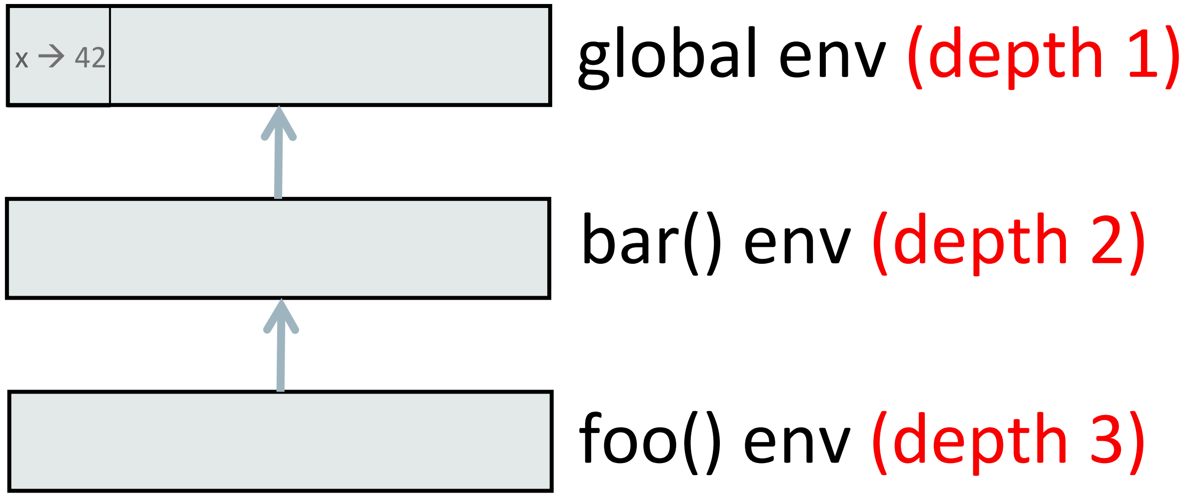


# One more problem…

- We don’t want a pointer to environment (to allow virtualization)
- Fortunately, environments can be counted!

```
foo = function(a) {
  print(a);
}
bar = function(a) {
  foo(a);
}

x = 42;
bar(x)
```




- Store environment depth with a promise

# Other promises

- Default promises
  - Environment stored with a promise
  - Inline caches used to reduce overhead of evaluating promises carrying the same code snippets
- Indirect promises
  - Technically – instance of eager promises (no costly meta data)
  - Practically – wrappers around other promise types
  - Evaluation cost the same as of the promise they are wrapping

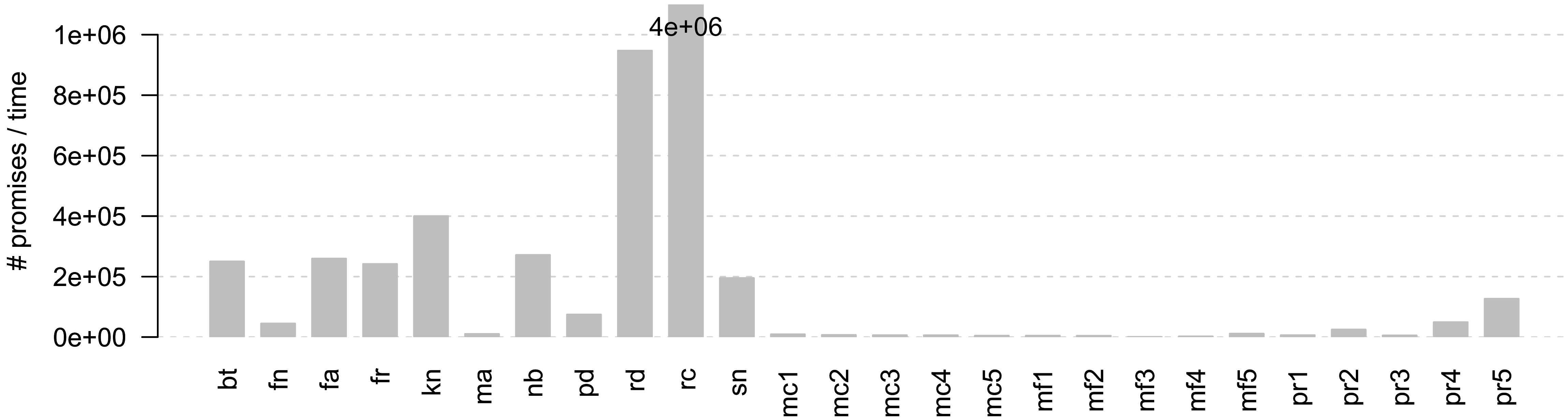
ORACLE

# Lazy evaluation optimization results

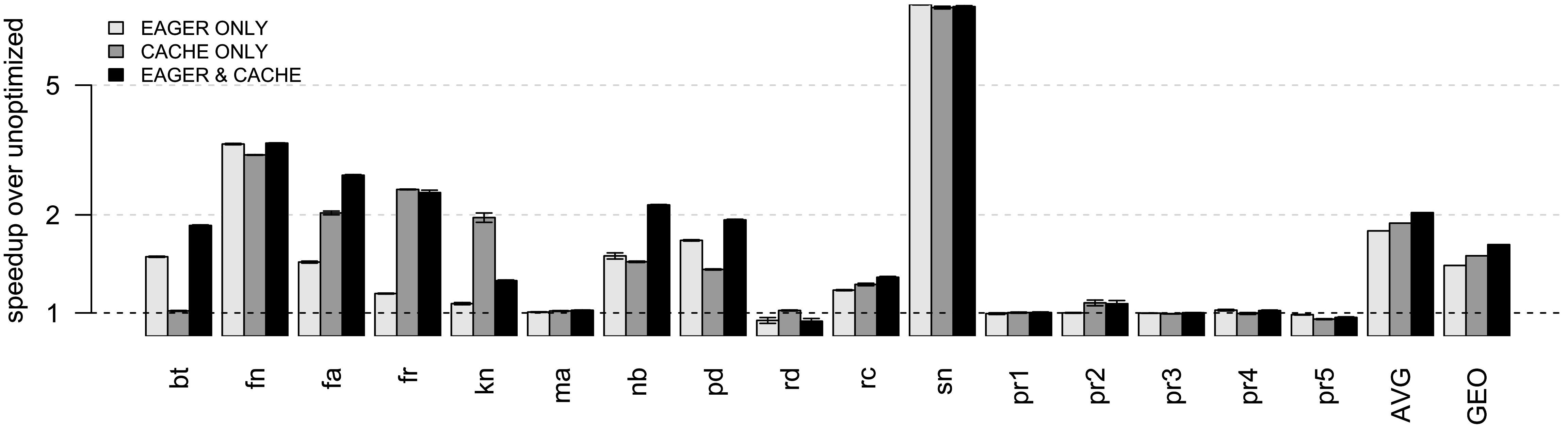
- Two benchmark suites
  - B25: matrix calculations + simple R computation tasks
  - Shootout: small applications consisting mostly of R code
- Estimated optimization potential measured in number of promises
- Three configurations to measure impact of the optimizations (peak performance plotted on logarithmic scale)
  - Eager promises optimization only
  - Caching for default promises only
  - Eager promises and caching combined

ORACLE

# Promise statistics



# Impact of lazy evaluation optimizations

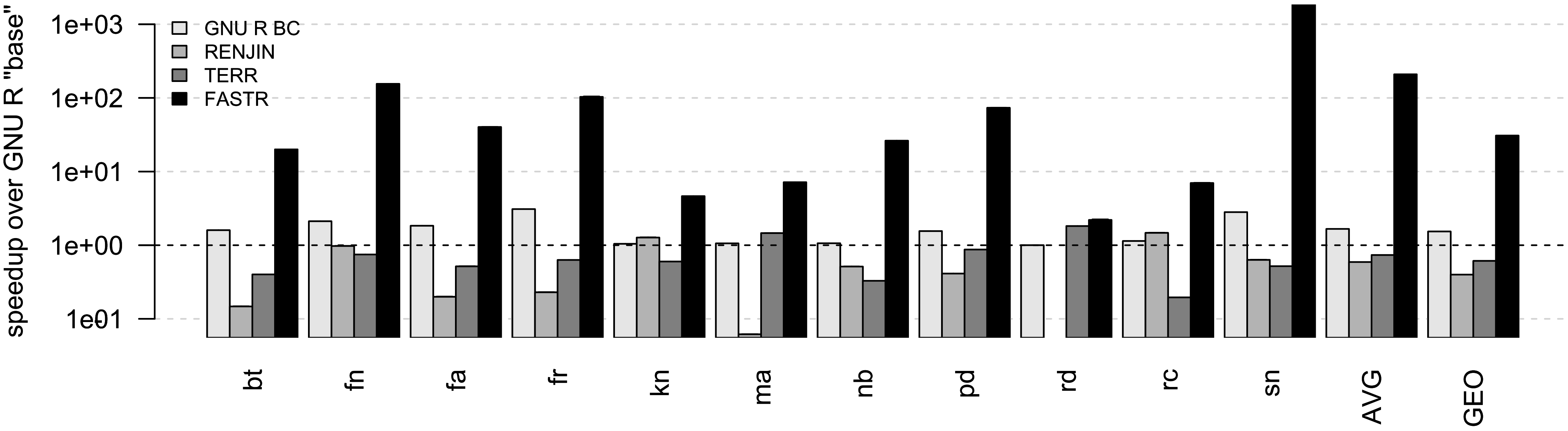


# Overall system performance

- Same two benchmark suites – b25 and shootout
- Five runtime configurations
  - GNU R “base” (default configuration)
  - GNU R “BC” (bytecode compiler)
  - Renjin (alternative R implementation from BeDataDriven)
  - TERR (alternative R implementation from TIBCO)
  - FastR
- Plotted peak performance on a logarithmic scale

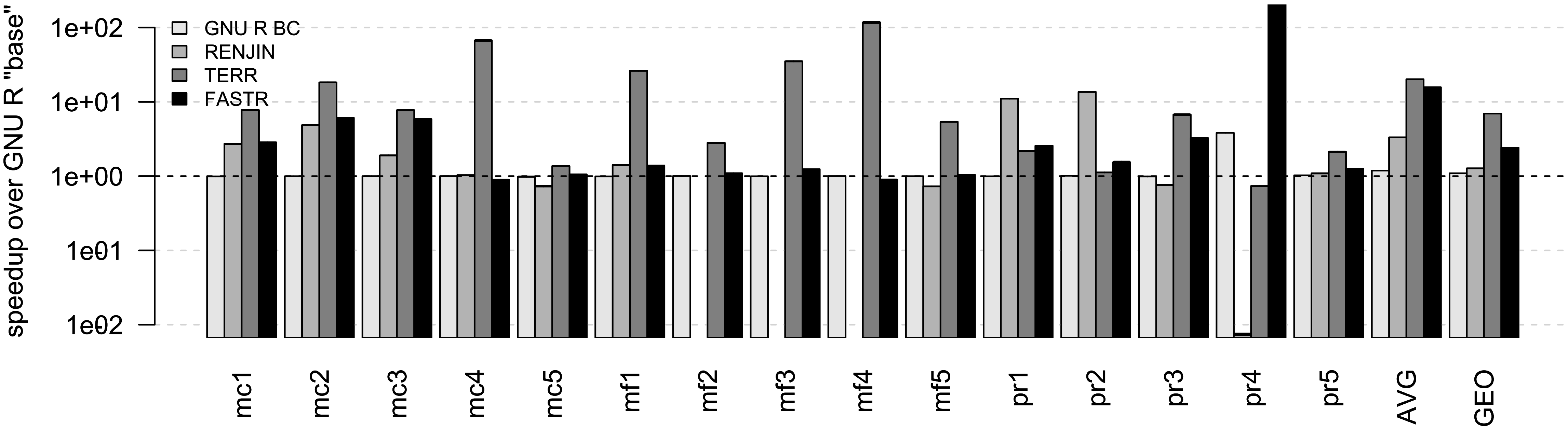
ORACLE

# Shootout benchmark suite




FastR’s average speedup: ~208.7 (geometric mean: ~30.8)

# B25 benchmark suite




- FastR's average speedup: ~15.7 (geometric mean: ~2.4)

# Project status

- Implemented all important language features, including lazy evaluation, calls to C/Fortran, S3 and S4 object models
- FastR can load over 2000 unmodified CRAN packages and run selected production applications in parallel
- Missing features include portions of native interface and selected builtins
- Bottom line – semantic compatibility is high but work ongoing on completeness and experimental features (e.g. autoparallelization)

# Acknowledgements

**Oracle**
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez

**Oracle (continued)**
Roland Schatz
Chris Seaton
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Lukas Stadler
Codrut Stancu
Jan Štola
Jaroslav Tulach
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

**Oracle Interns**
Brian Belleville
Miguel Garcia
Shams Imam
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
David Piorkowski
Gregor Richards
Robert Seilbeck
Rifat Shariyar

**Alumni**
Erik Eckstein
Michael Haupt
Christos Kotselidis
Hyunjin Lee
David Leibs
Chris Thalinger
Till Westmann

**JKU Linz**
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Huber
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

**University of Edinburgh**
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

**LaBRI**
Floréal Morandat

**University of California, Irvine**
Prof. Michael Franz
Gulfem Savrun Yeniceri
Wei Zhang

**Purdue University**
Prof. Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

**T. U. Dortmund**
Prof. Peter Marwedel
Helena Kotthaus
Ingo Korb

**University of California, Davis**
Prof. Duncan Temple Lang
Nicholas Ulle

**University of Lugano, Switzerland**
Prof. Walter Binder
Sun Haiyang
Yudi Zheng

ORACLE

ORACLE®