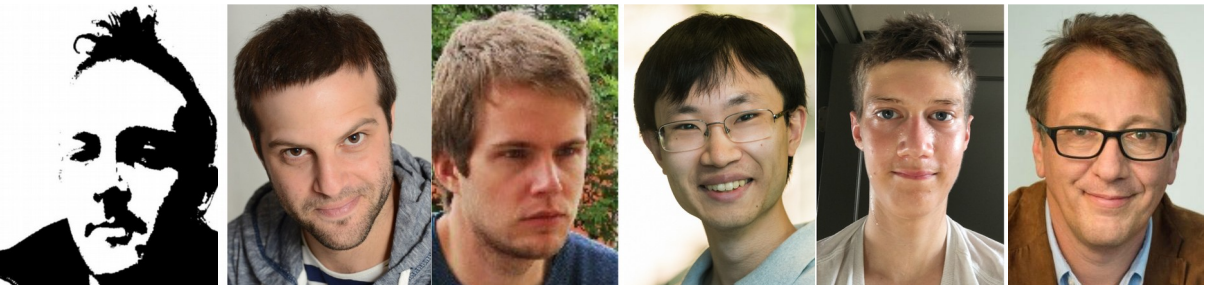
R Melts Brains

An IR for First-Class Environments and Lazy Effectful Promises

Olivier Flückiger, Guido Chari, Jan Ječmen,
Ming-Ho Yee, Jakob Hain, Jan Vitek



Northeastern University
Czech Technical University

R's mutable variable scopes heavily interfere with compiler optimizations

PIR – an IR to explicitly model, analyze and lower R variables

How to design a compiler for R

Scope in R

Variables contained in the environment...
...depends on the dynamics.

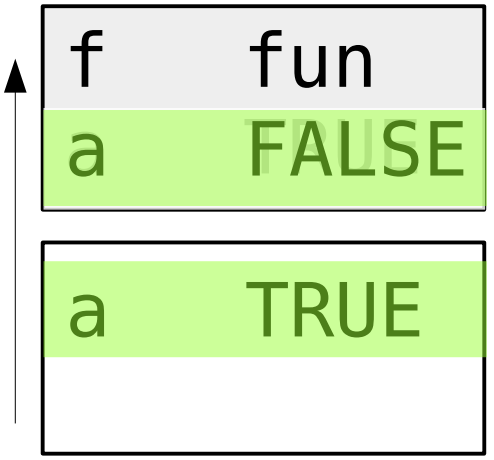
```
a <- ...

function() {
  if (a)
    b <- 1
  b
}
```

Closures capture environment...
...and can mutate it.

```
f <- function() {
  a <-  a
  a <<- FALSE
}
a <- TRUE
f()
a
```

environments



```
f <- function() {

}
a <- TRUE
f()
a
```

Callees...

...can mutate the environment.

```
f <- function() {

  e <- parent.frame()
  rm(`a`, envir=e)

}
g <- function() {
  a <- TRUE
  f()
  a   # → object `a` not found
}
```

Arguments...

...are promises.

...can mutate the environment.

```
f <- function(q) {
  e <<- environment()
  a <- TRUE
  q
  a
}
f(
  rm(`a`, envir=e)
)
```

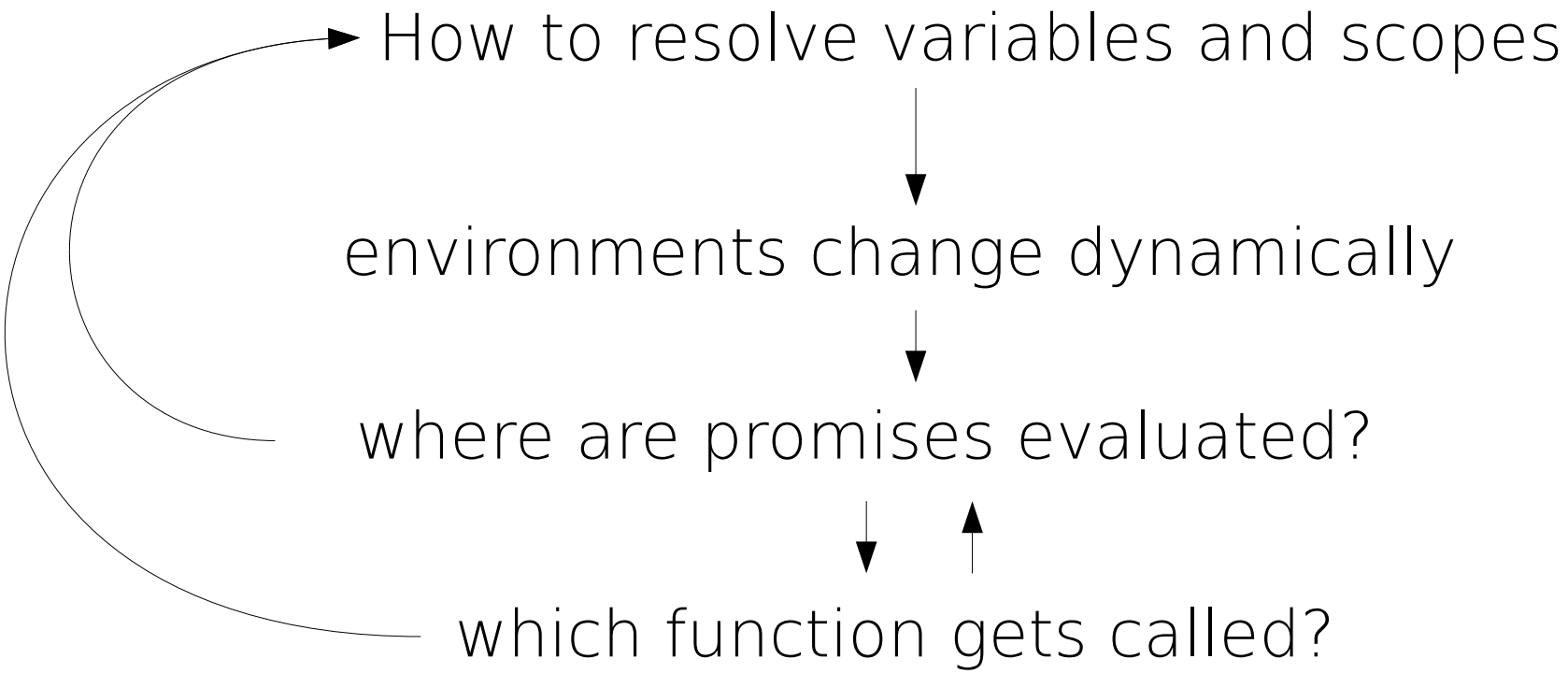
See “On the Design, Implementation and Use of Laziness in R” by Aviral Goel, Jan Vitek on Thu 14:45

Local variables...

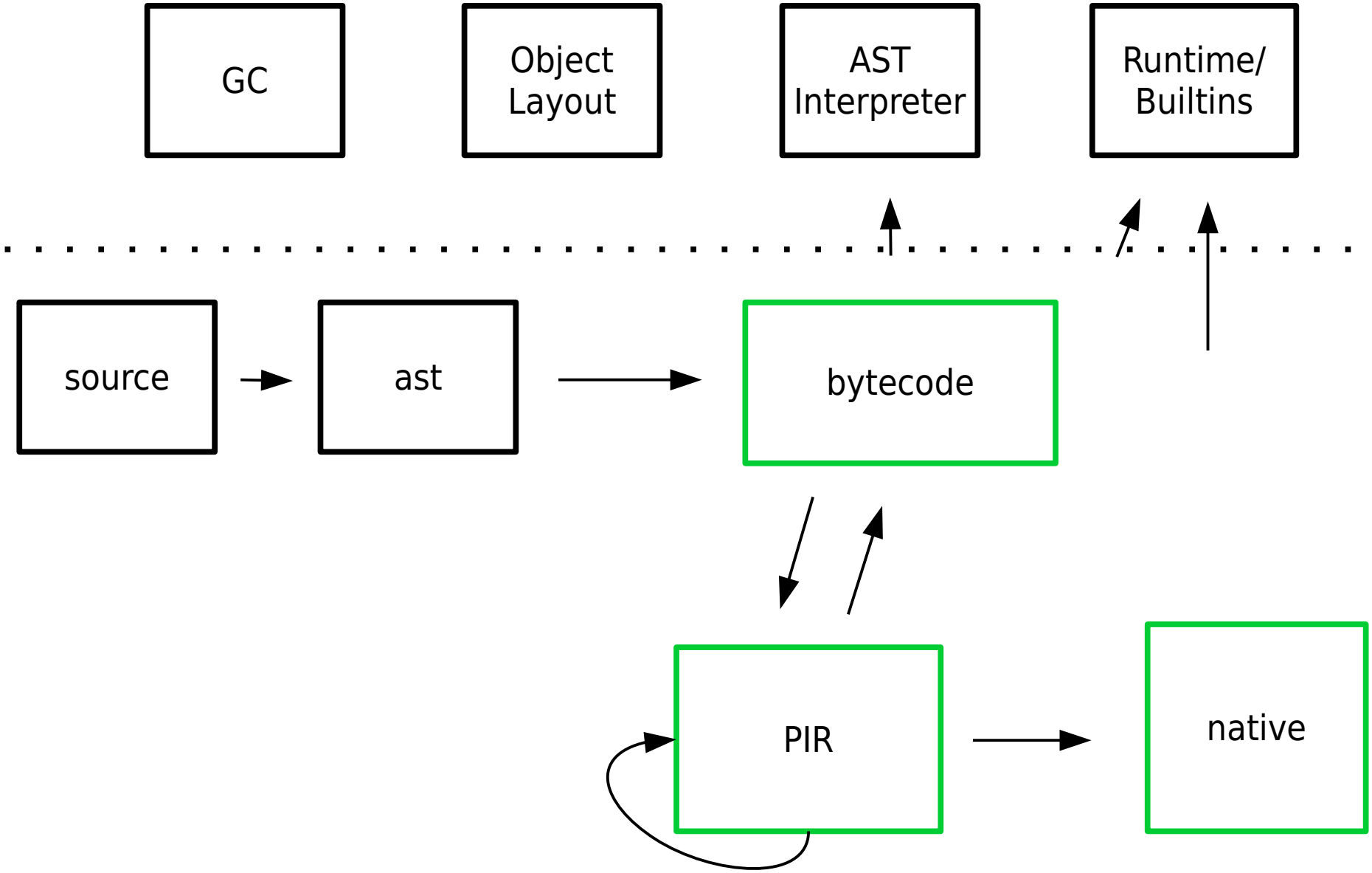
...stored in first-class environment

...accessible to callees

...accessible to promises


How to resolve variables and scopes
environments change dynamically
where are promises evaluated?
which function gets called?

An IR will Save Us


GC
Object Layout
AST Interpreter
Runtime/ Builtins
source
ast
bytecode
PIR
native

PIR design

SSA

Explicit environments and promises

more explicit/detailed than bytecode
higher-level than LLVM

Explicit Environments

$$
\begin{array}{lll}
\textit{instr} & ::= & \\
 & \mid & \texttt{MkEnv}\ ((x = a)^*\ : env) \quad \text{create env.} \\
 & \mid & \texttt{LdVar}\ (x,\ env) \qquad\qquad\ \ \text{load variable} \\
 & \mid & \texttt{StVar}\ (x,\ a,\ env) \qquad\ \ \text{store variable}
\end{array}
$$

Scope Resulution

Scope Resolution

```
function () {
  if (...) x <- 1
  else     x <- 2
  x
}
```

BB_0 : e1 = MkEnv (: G)
%2 = ...
Branch (%2, BB_1, BB_2)

BB_1 : %4 = LdConst [1] 1
StVar (x, %4, e1)
Branch BB_3

BB_2 : %7 = LdConst [1] 2
StVar (x, %7, e1)
Branch BB_3

BB_3 : %10 = LdVar (x, e1)
%11 = Force (%10) e1
Return (%11)

Scope Resolution : 1. Find Reaching Stores

```
function () {
  if (...) x <- 1
  else     x <- 2
  x
}
```

BB1
  x = %4
BB2
  x = %7
BB3
  x = %4 | %7

BB_0 : e1 = MkEnv (: G)
%2 = ...
Branch (%2, BB_1, BB_2)

BB_1 : %4 = LdConst [1] 1
StVar (x, %4, e1)
Branch BB_3

BB_2 : %7 = LdConst [1] 2
StVar (x, %7, e1)
Branch BB_3

BB_3 : %10 = LdVar (x, e1)
%11 = Force (%10) e1
Return (%11)

Scope Resolution : 2. Replace Loads

```
function () {
  if (...) x <- 1
  else     x <- 2
  x
}
```

BB1
  x = %4
BB2
  x = %7
BB3
  x = %4 | %7

BB_0 :
  e1 = MkEnv (: G)
  %2 = ...
  Branch (%2, BB_1, BB_2)

BB_1 :
  %4 = LdConst [1] 1
  StVar (x, %4, e1)
  Branch BB_3

BB_2 :
  %7 = LdConst [1] 2
  StVar (x, %7, e1)
  Branch BB_3

BB_3 :
  %10 = Phi (BB_1 : %4, BB_2 : %7)
  Return (%10)

This looks suspiciously easy

Scope Resolution : Stub Environments

```
function () {
  if (...){x <- 1;f()}
  else     x <- 2
  x
}
```

BB1
  x = *?*
BB2
  x = %7
BB3
  x = *?* | %7

BB_0 : e1 = (MkEnv)(: G)
  %2 = ...
  Branch (%2, BB_1, BB_2)

BB_1 : %4 = LdConst [1] 1
  StVar (x, %4, e1)
  Call
  Branch BB_3

BB_2 : %7 = LdConst [1] 2
  StVar (x, %7, e1)
  Branch BB_3

BB_3 : %10 = LdVar (x, e1)
  %11 = Force (%10) e1
  Return (%11)

R Environment

- ...a **hashmap**
- ...**bindings** can be changed **everywhere**

Stub Environment

- ...an **array**
- ...**values** can only be changed **locally**

Scope Resolution : Stub Environments

```
function () {
  if (...)
  else
  x
}
```

Static analysis if possible…
…speculation if needed

```
BB0 :  e1   = (MkEnv)( : G)
                                  BB1, BB2)
BB1 :  %4   =  LdConst [1] 1
               StVar (x, %4, e1)
               Call
               Branch BB3
BB2 :  %7   =  LdConst [1] 2
               StVar (x, %7, e1)
               Branch BB3
BB3 :  %10  =  LdVar (x, e1)
       %11  =  Force (%10) e1
               Return (%11)
```

→ deopt

```
BB1
   x = ?
BB2
   x = %7
BB3
   x = ?  | %7
```

Inlining of Closures and Promises

Explicit Environments

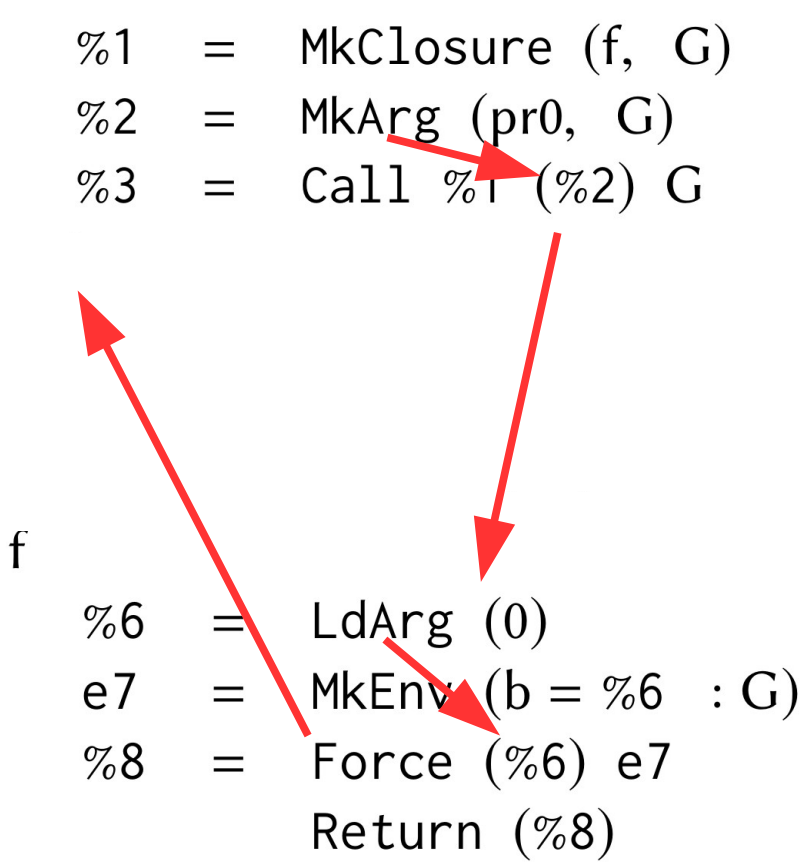
$$
\begin{array}{lll}
\textit{instr} & ::= & \\
 & \mid \texttt{MkEnv}\ ((x = a)^*\ : env) & \text{create env.} \\
 & \mid \texttt{LdVar}\ (x,\ env) & \text{load variable} \\
 & \mid \texttt{StVar}\ (x,\ a,\ env) & \text{store variable} \\
 & \mid \texttt{MkClosure}\ (id,\ env) & \text{create closure}
\end{array}
$$

Explicit Promises

$$
\begin{array}{lll}
\textit{instr} & ::= & \\
 & \mid \ \texttt{MkArg}\ (\textit{id},\ \textit{env}) & \text{create promise} \\
 & \mid \ \texttt{Force}\ (a)\ \textit{env} & \text{force promise} \\
 & \mid \ \texttt{MkEnv}\ ((x = a)^*\ :\ \textit{env}) & \text{create env.} \\
 & \mid \ \texttt{LdVar}\ (x,\ \textit{env}) & \text{load variable} \\
 & \mid \ \texttt{StVar}\ (x,\ a,\ \textit{env}) & \text{store variable} \\
 & \mid \ \texttt{MkClosure}\ (\textit{id},\ \textit{env}) & \text{create closure}
\end{array}
$$

Promise Inlining

```
f <- function(b) b
f(x)
```



Promise Inlining : 1. Normal Inlining

```
f <- function(b) b
f(x)
```

```
%1    =   MkClosure (f, G)
%2    =   MkArg (pr0, G)
# inlinee
e7    =   MkEnv (b = %2 : G)
%8    =   Force (%2) e7
```

inlinee retains environment

```
pr0
   %4    =   LdVar (x, G)
   %5    =   Force (%4) G
             Return (%5)
```

```
f <- function(b) {
  print(ls())     # -> "b"
  b
}
```

```
g (0)
v (b = %6 : G)
e (%6) e7
rn (%8)
```

Promise Inlining : 2. Promise Inlining

```
f <- function(b) b
f(x)
```

```
%2   =   MkArg (pr0, G)
# inlinee
e7   =   MkEnv (b = %2 : G)
%8   =   Force (%2) e7
```

pr0

```
%4   =   LdVar (x, G)
%5   =   Force (%4) G
         Return (%5)
```

dominating force instruction

Promise Inlining : 2. Promise Inlining

```
f <- function(b) b
f(x)
```

```
%2   =   MkArg (pr0, G)
# inlinee
e6   =   MkEnv (b = %2 : G)
# inlined promise
%4   =   LdVar (x, G)
%5   =   Force (%4) e6
```

Assume `x` does not do reflection

| `Force` (*a*) *env* — force promise

env – dependency, used in case of reflection

Promise Inlining : 2. Promise Inlining

```
f <- function(b) b
f(x)
```

```
%4   =   LdVar (x, G)
%5   =   Force (%4) e0
```

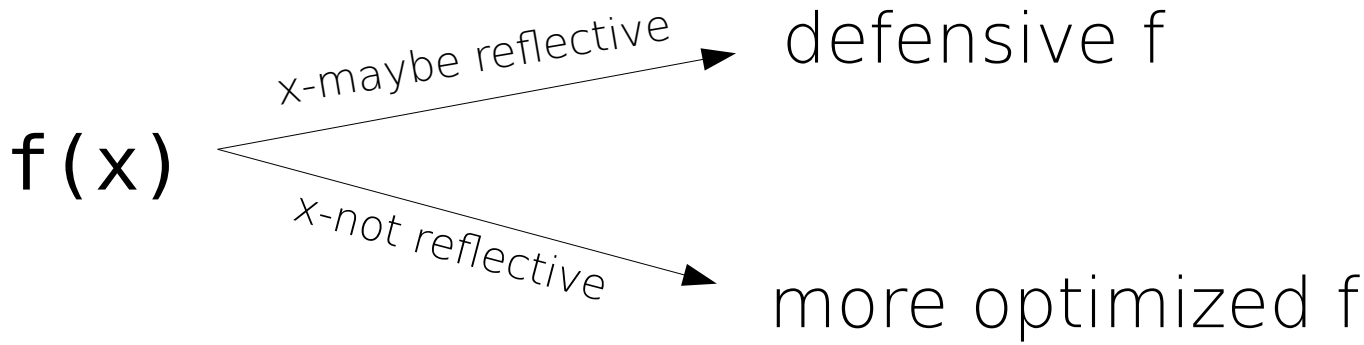
Assume `x` does not do reflection

Promise Inlining : 3. Is the promise safe?

Inter-procedural analysis

Stub Environment

Version Dispatch



When possible, statically...

...lower R variables to PIR registers

...inline promises

...elide environments

Results

(12%-65% of closures,
28%-87% of invocations)
...sometimes it is possible to statically resolve R scopes

...disabling either scope resolution or promise inlining significantly defeats the other optimizations

Ř Melts Brains

https://github.com/reactorlabs/rir

R is a bizarre language.

Static reasoning for some R programs is possible.

Explicitly model what is hard to reason.

Give us your slow R scripts!

https://o1o.ch/about/brains

https://arxiv.org/abs/1907.05118