Compiling R and other Adversarial Languages

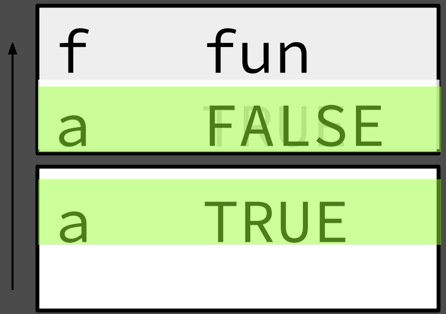
Olivier Flückiger — MathWorks — 6/23/22

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

Fun with Environments

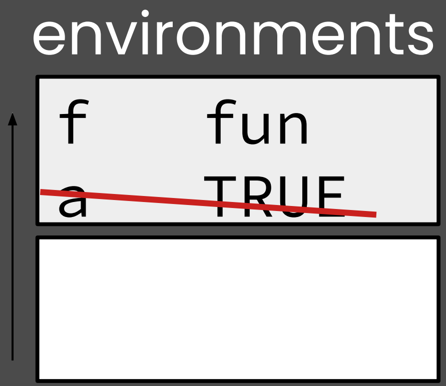
```
f <- function() {
    a <-   a
    a <<-  FALSE

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

environments



Fun with Environments

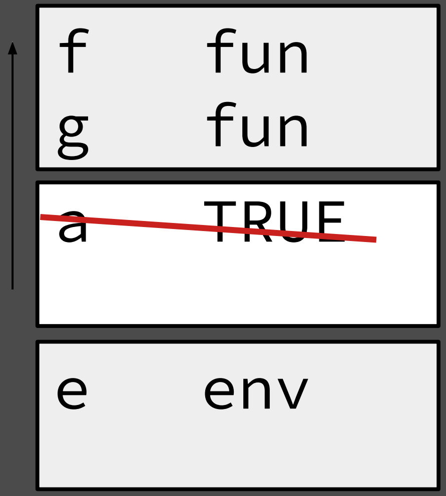
```
f <- function() {
    e <- parent.env()
    rm(`a`, envir=e)

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



Fun with Environments

```r
f <- function() {
    e <- sys.frame(-1)
    rm(`a`, envir=e)
}
g <- function() {
    a <- TRUE
    f()
    a    # → object `a` not found
}
```

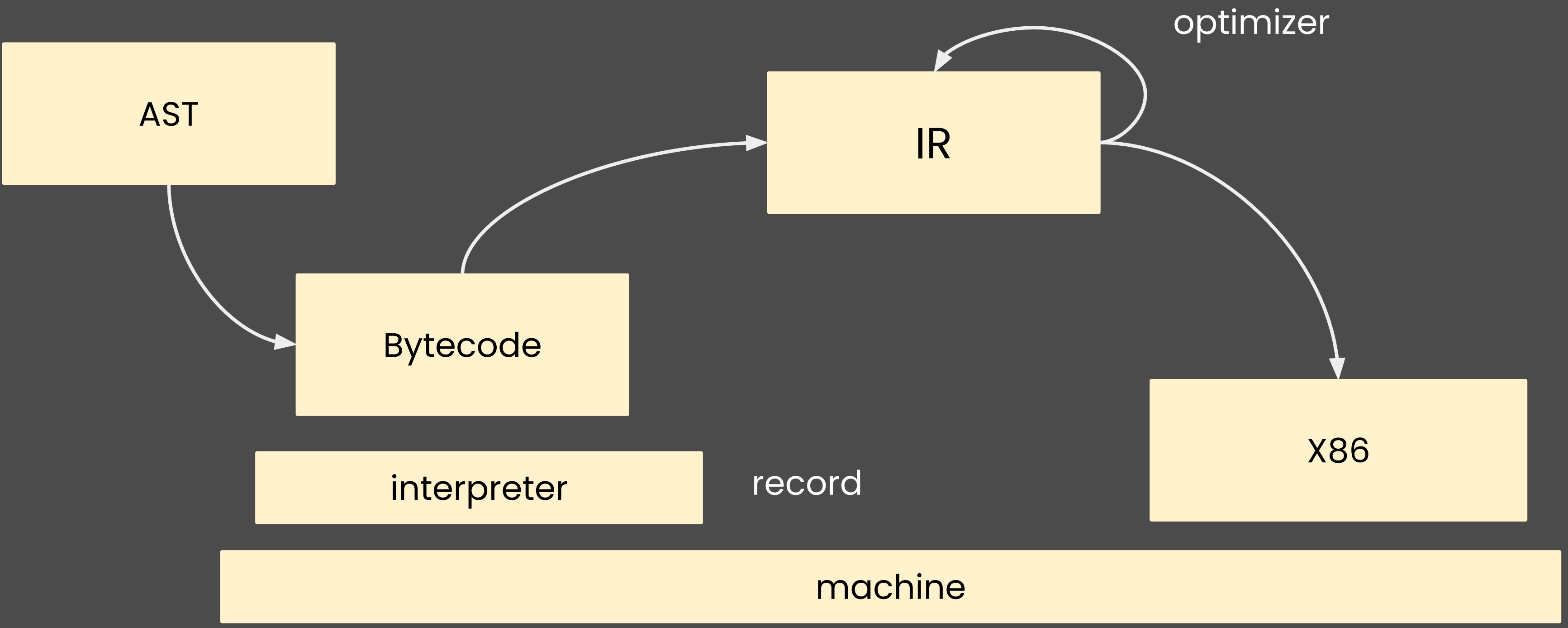
environments



Fun with Environments

```
f <- function() {
    e <- sys.frame(-1)
    rm(`a`, envir=e)
}
g <- function(x) {
    a <- TRUE
    x
    a   # → object `a` not found
}
g(f())
```

Ř Architecture



IR design

- SSA
- **Explicit** environments and promises
- more explicit/detailed than bytecode
  higher-level than LLVM

[DLS'19] R Melts Brains, Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek

Explicit Environments

instr	::=		
	|	`MkEnv` $((x = a)^* \; : env)$	create env.
	|	`LdVar` $(x, \; env)$	load variable
	|	`StVar` $(x, \; a, \; env)$	store variable

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. Analysis

```
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: 1. Analysis, 2. Load Elision

```
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

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

foo()

lol, no…

How to compile a dynamic language?

```
factor <- function(x)
  x * factor
```

```
factor ← 2
...

scale(1)
scale(c(1,2))
scale(1+2)
```

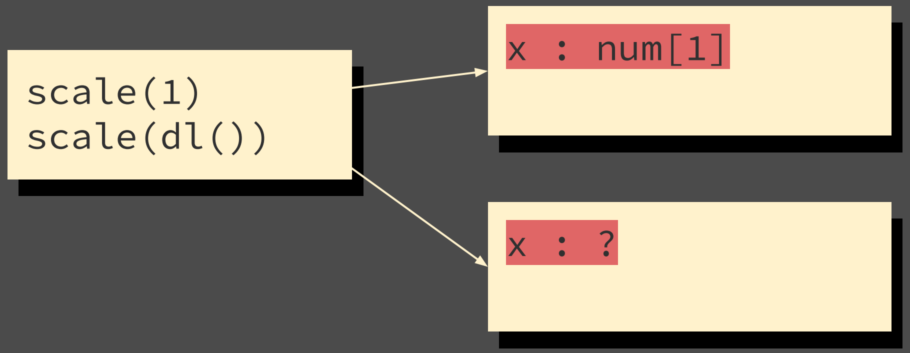
Speculation

```
# assume factor==2
x + x
```

Specialization

```
for (i in x)
  res[i] = x[i] *num factor
```

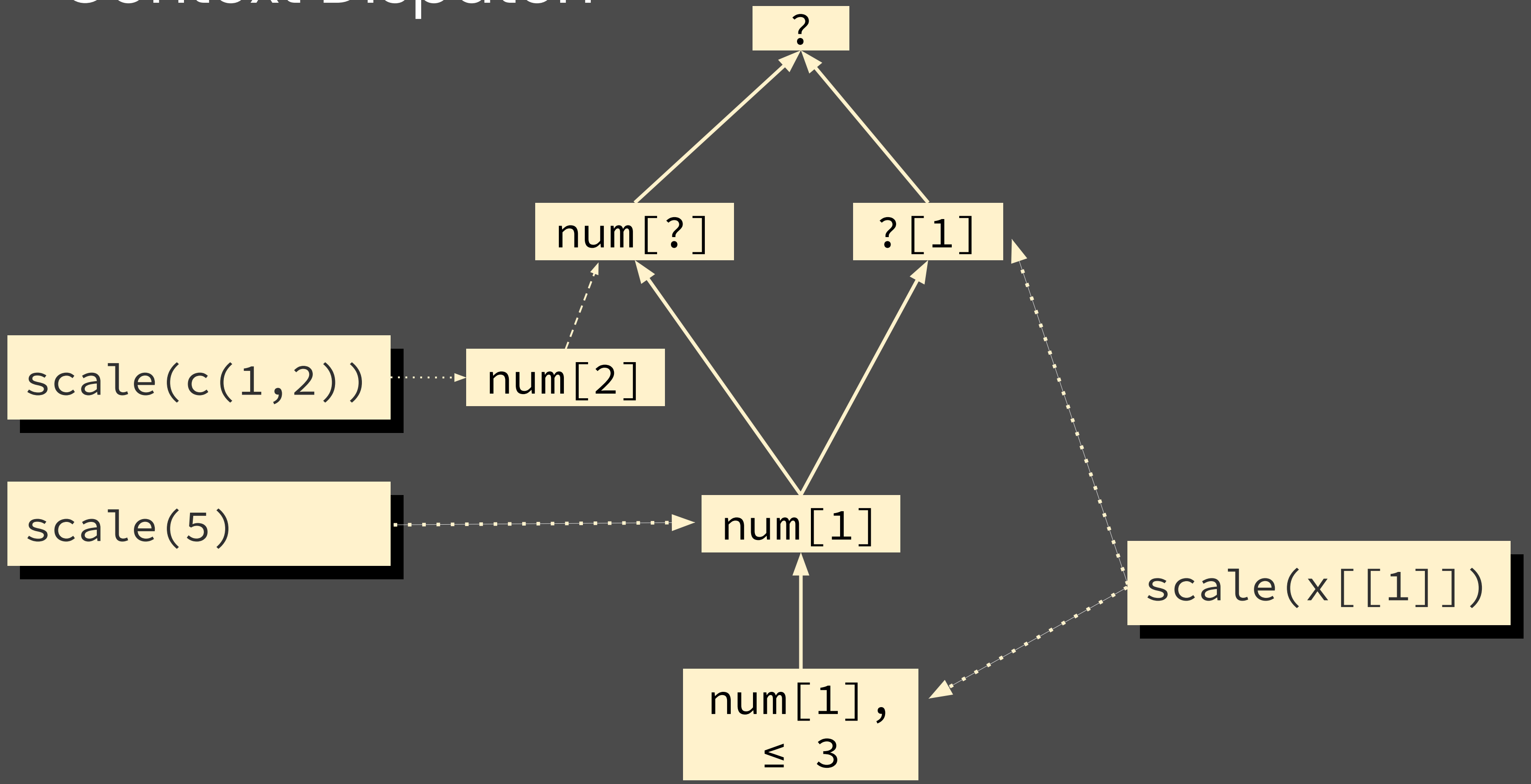
Specialization



- Communicate summary information from caller to callee, like in a modular analysis
- Share specialized code between different call-sites with compatible summaries

[OOPSLA'20] Context Dispatch for Function Specialization,
Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek

Context Dispatch



Context Dispatch in Practice

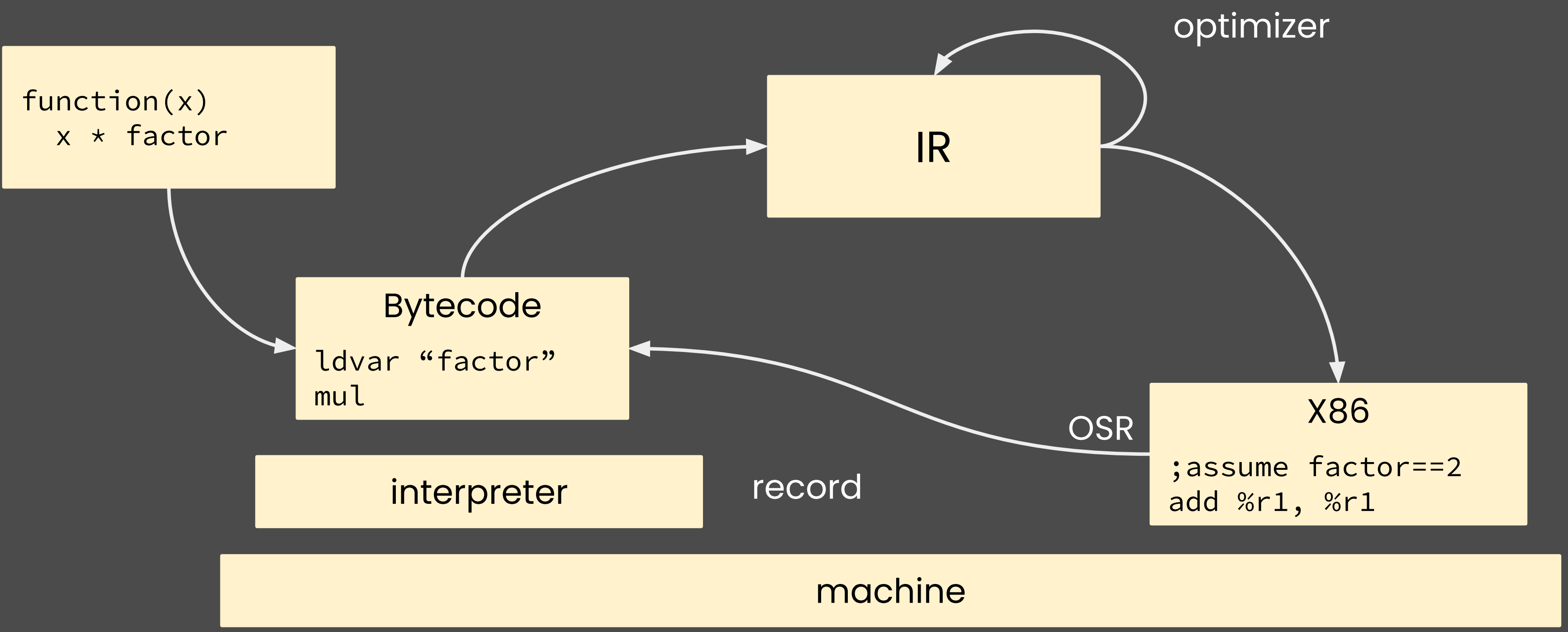
- 64 bit context, linearizes partial order
- Properties:
  - types,
  - optional arguments,
  - eagerness, reflection
- JITed after ~100 sub-optimal dispatches
- Few functions have many contexts
- Only for properties checked up-front

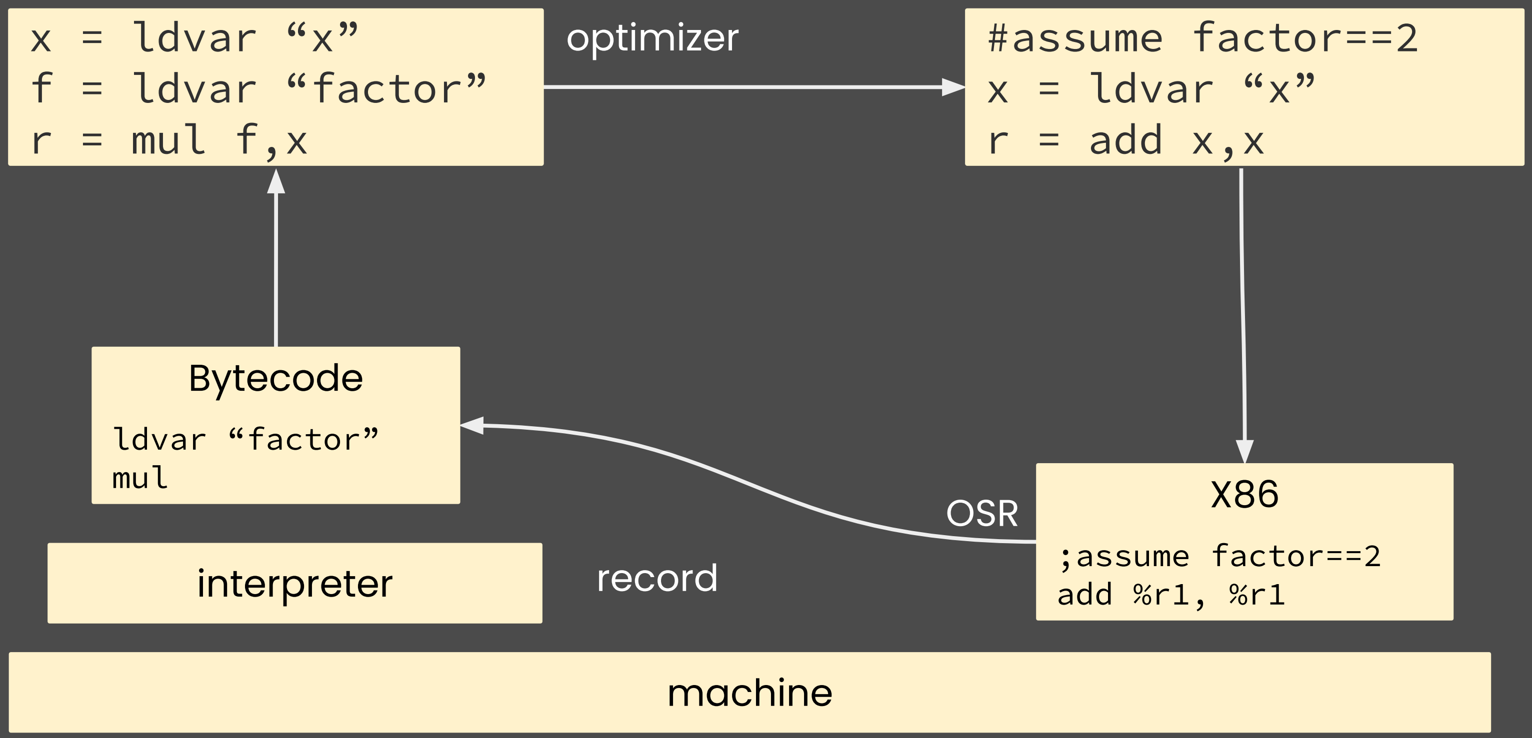
Speculation

```
scale  ← function(x) {
  …
  # assume factor==2
  x + x
}
```

[POPL'18] Speculative Optimization with Dynamic Deoptimization,
Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek

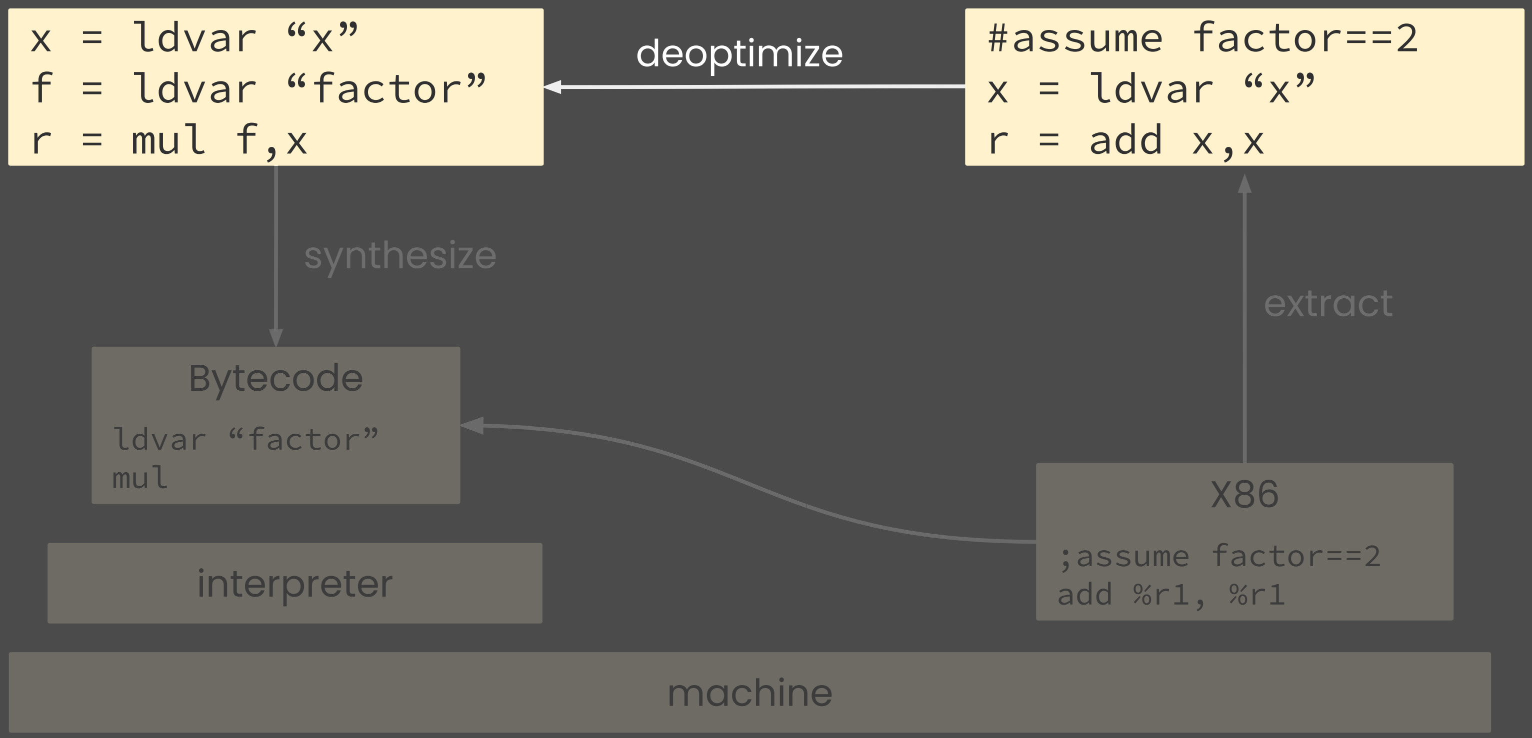
Why is it hard to optimize under assumptions?





x = ldvar "x"
f = ldvar "factor"
r = mul f,x
optimizer
#assume factor==2
x = ldvar "x"
r = add x,x
Bytecode
ldvar "factor"
mul
OSR
X86
;assume factor==2
add %r1, %r1
interpreter
record
machine

On-Stack Replacement (OSR)



Inserting OSR exit points

```
x * factor
```

baseline

```
1: x = ldvar "x"

2: f = ldvar "factor"

3: r = mul x, f
```

optimized

```
1: x = ldvar "x"
   anchor 2, (x=x)
2: f = ldvar "factor"

3: r = mul x, f
```

Speculation

```
x * factor
```

baseline

```
1: x = ldvar “x”

2: f = ldvar “factor”

3: r = mul x, f
```

optimized

```
1: x = ldvar “x”
   anchor 2, (x=x)
2: f = ldvar “factor”
   assume f==2
3: r = add x, x
```

Constant Folding

```
x * factor
```

baseline

```
1: x = ldvar "x"

2: f = ldvar "factor"

3: r = mul x, f
```

optimized

```
1: x = 1 #ldvar "x"
   anchor 2, (x=1)
2: f = ldvar "factor"
   assume f==2
3: r = add x, x
```

Inlining

```
x * factor
```

```
anchor ...
s = call scale(1)
```

```
1: x = 1
   anchor 2, (x=1)
2: f = ldvar “factor”
   assume f==2
3: r = add x, x
```

Inlining

`x * factor`

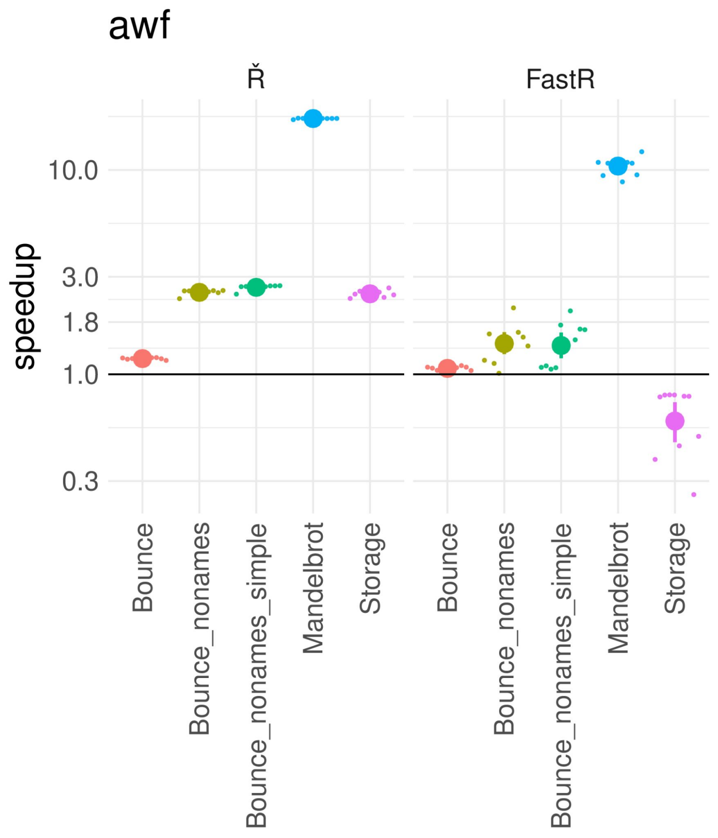
```
anchor ...

    1: x = 1
       anchor 2, (x=1)
    2: f = ldvar “factor”
       assume f==2
    3: r = add x, x

s = r
```

Ř Status

- A bug-compatible JIT compiler for the R language.
- Its IR closely follows sourir's assume and is structured around context dispatch.
- CD and assume are the only source of dynamic information for optimizations.

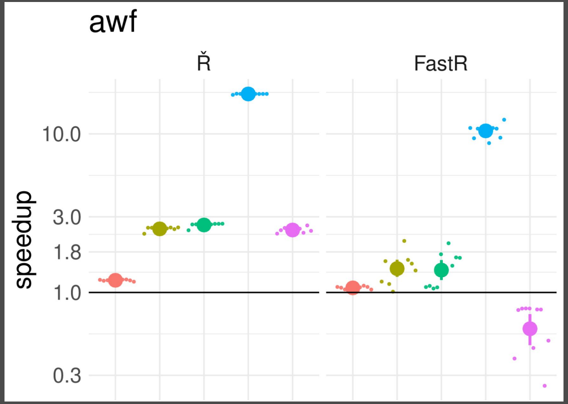
Demo....

Ř Eval



	vs. GNU R	vs. FastR	¬spec
AreWeFast	3.2x	1.8x	0.3x
RealWorld	1.8x	0.6x	0.4x
Shootout	1.7x	0.9x	0.6x

Ř, a JIT for R



| MkEnv $((x = a)^* \; : env)$ create env.

Speculation

```
# assume factor==2
x + x
```

```
scale(...)
```

Specialization

- r-vm.net
- o1o.ch