Just in Time

Assumptions and Speculations

Olivier Flückiger — PhD Thesis Defense — 6/22/22

Why is it hard to compile a dynamic language?

```
x * factor
```

Why is it hard to compile a dynamic language?

```
factor ← 2
scale  ← function(x=1) {
  x * factor
}
scale(1)        # 2
scale(c(1,2))   # c(2,4)
scale(“1”)      # error
scale(obj)      # obj’
scale()         # 2
```

```
...
else if (TYPEOF(x) == REALSXP || TYPEOF(y) == REALSXP) {
  if (TYPEOF(x) != INTSXP) COERCE_IF_NEEDED(x, REALSXP, xpi);
  if (TYPEOF(y) != INTSXP) COERCE_IF_NEEDED(y, REALSXP, ypi);
  val = real_binary(oper, x, y);
} else {
  val = integer_binary(oper, x, y, call);
}

/* quick return if there are no attributes */
if (! xattr && ! yattr) {
  UNPROTECT(nprotect);
  return val;
}
...
```

GNU R, arith.c

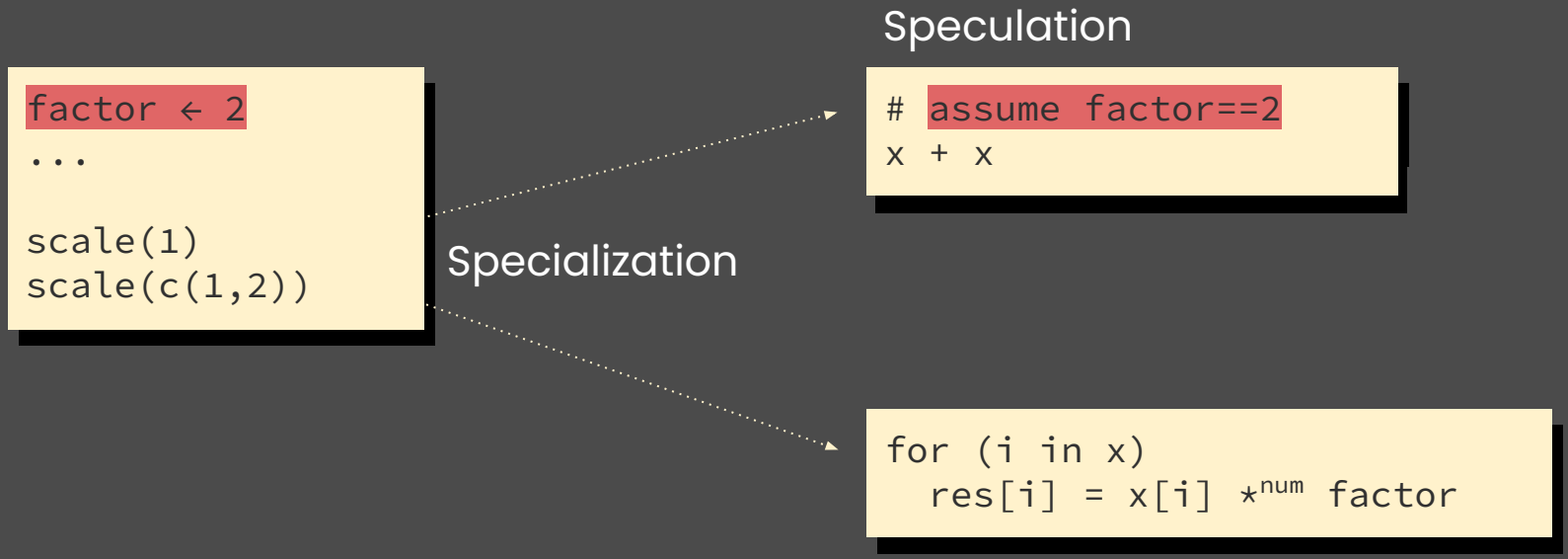
```
...
/* Handle some scaler operations immediately */
if (IS_SCALAR(arg1, REALSXP)) {
  double x1 = SCALAR_DVAL(arg1);
  if (IS_SCALAR(arg2, REALSXP)) {
    double x2 = SCALAR_DVAL(arg2);
    ans = ScalarValue2(arg1, arg2);
    switch (PRIMVAL(op)) {
      ...
      case TIMESOP: SET_SCALAR_DVAL(ans, x1 * x2);
      return ans;
```

`*`: 10k of instructions

GNU R, arith.c

How to compile a dynamic language?

```
x * factor
```



Outline

1. Specialization: Context Dispatch
2. Speculation: The Assume Instruction
3. Ř: Case Study

Thesis

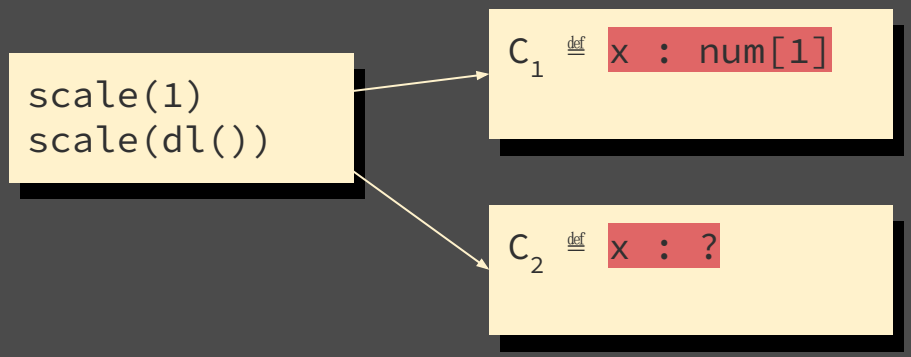
Assume and **context dispatch**

provide the basis for optimizations

based on run-time assumptions

in a competitive just-in-time compiler.

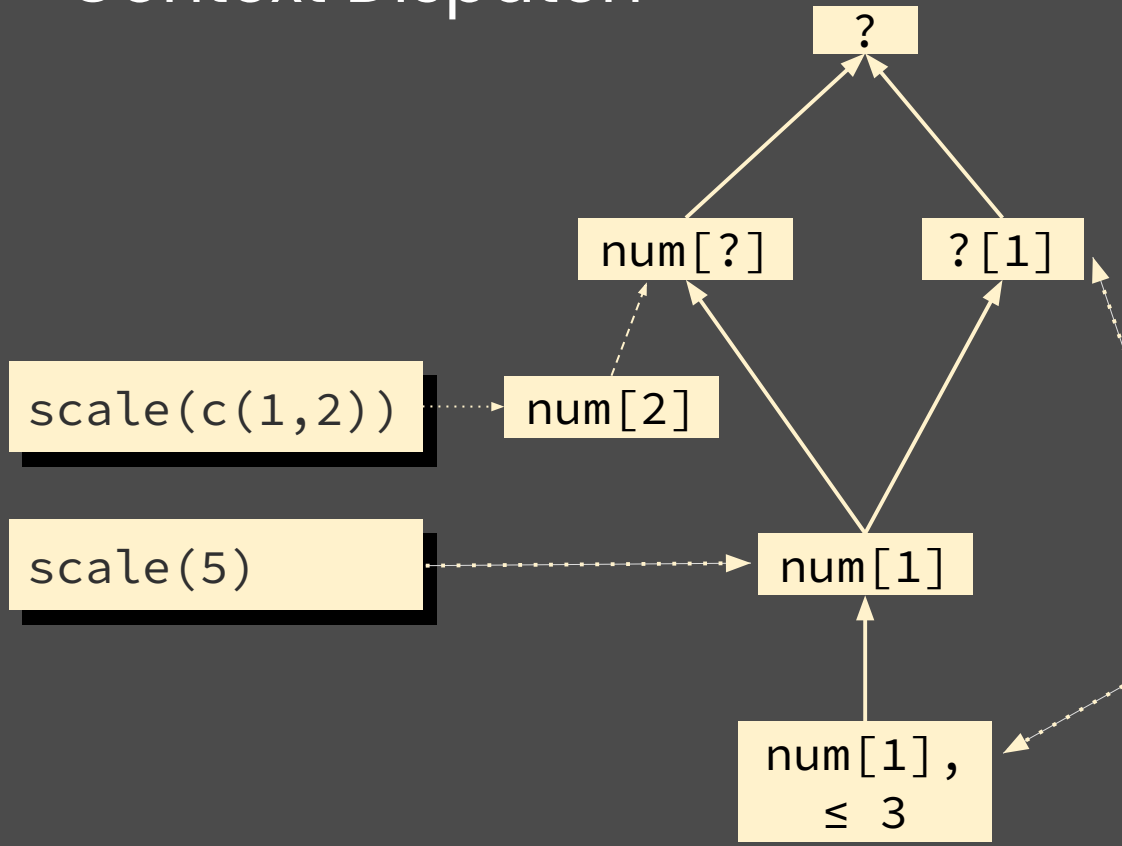
1. Specialization



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

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

Context Dispatch



$\forall$ S - Program State

$C_p < C_q \iff C_p(S) \Rightarrow C_q(S)$

C(S) - Current Context

n - Dispatch Target

$C < C_n \Rightarrow C_n(S)$

scale(x[[1]])

Context Dispatch in Practice

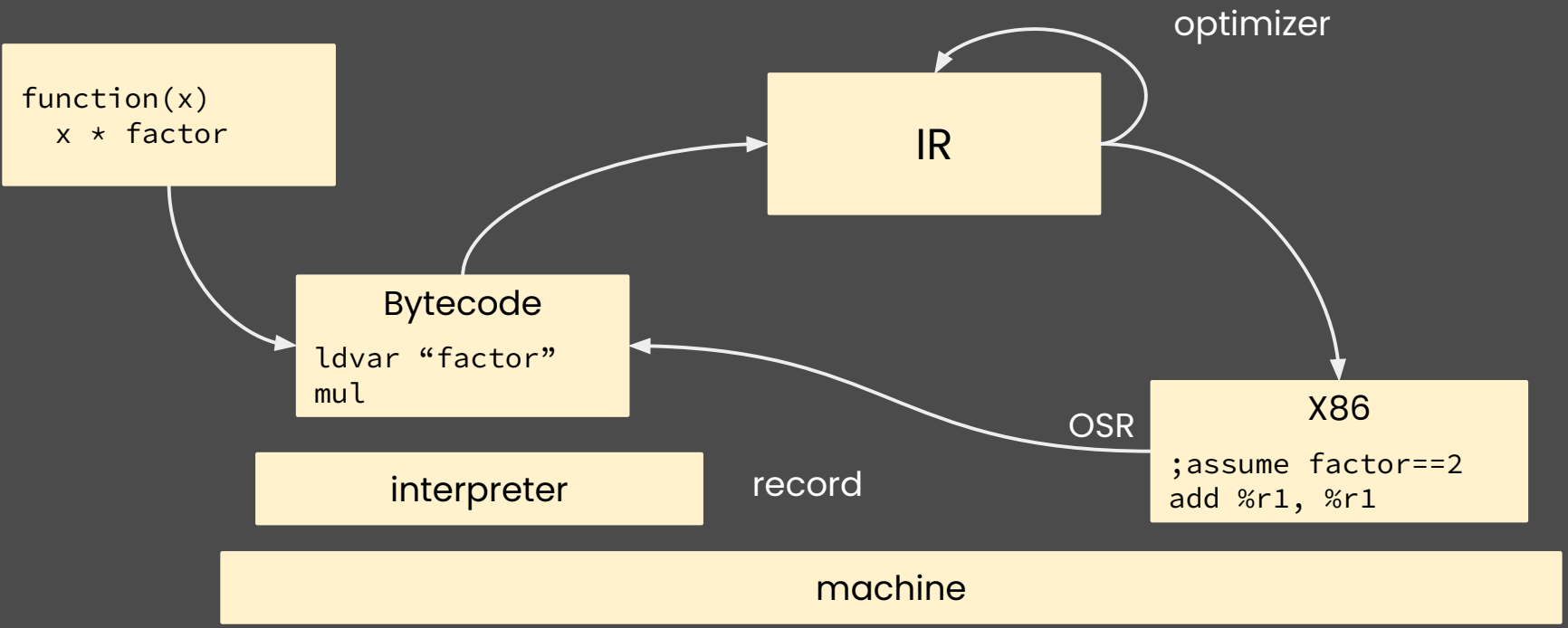
- 64 bit context, linearizes partial order
- Properties:
  - types,
  - optional arguments,
  - eagerness, reflection
- JITed after ~100 sub-optimal dispatches
- Few functions have many contexts
- Only for properties checked up-front

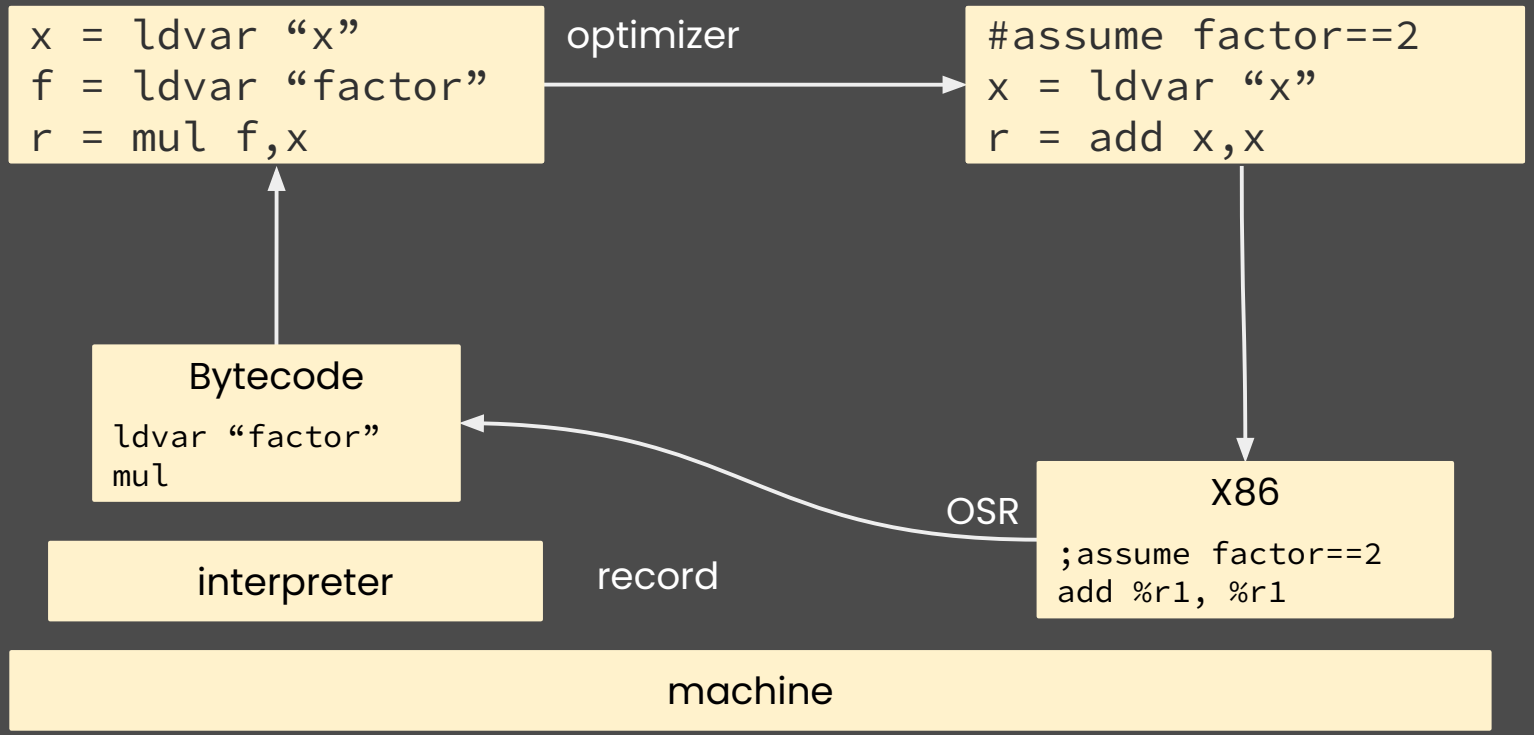
2. Speculation

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

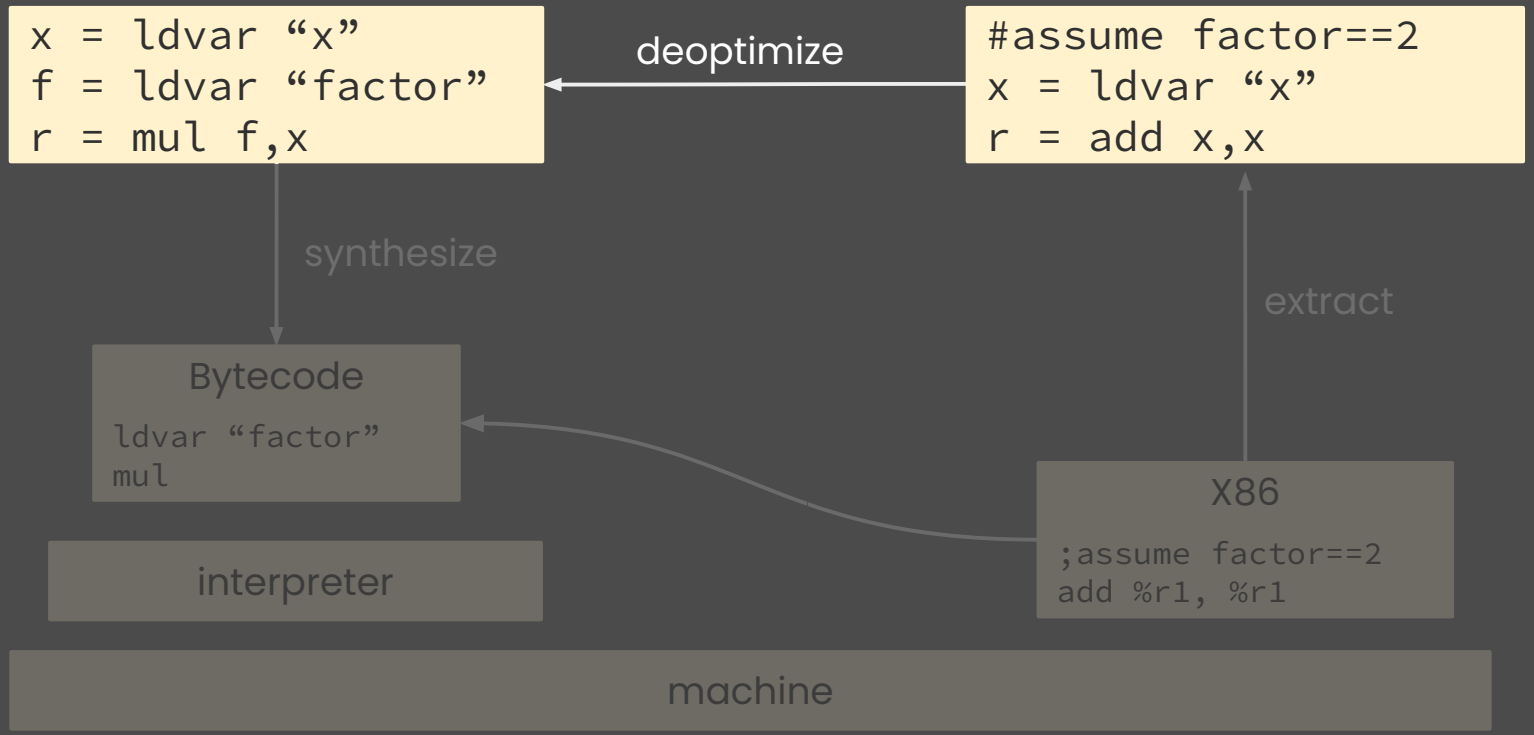
[POPL'18] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek
[POPL'22] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek

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)



How To Assume?

- Model OSR exit points
- Add new speculations
- Interaction with traditional optimizations

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”

2: f = ldvar “factor”

3: r = mul x, f
```

1. copy

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
```

1. copy
2. identity-anchor

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
```

1. copy
2. identity-anchor

Speculation

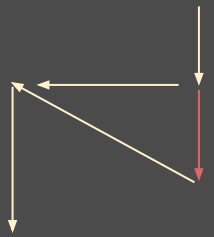
```
x * factor
```

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
```

1. speculate

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

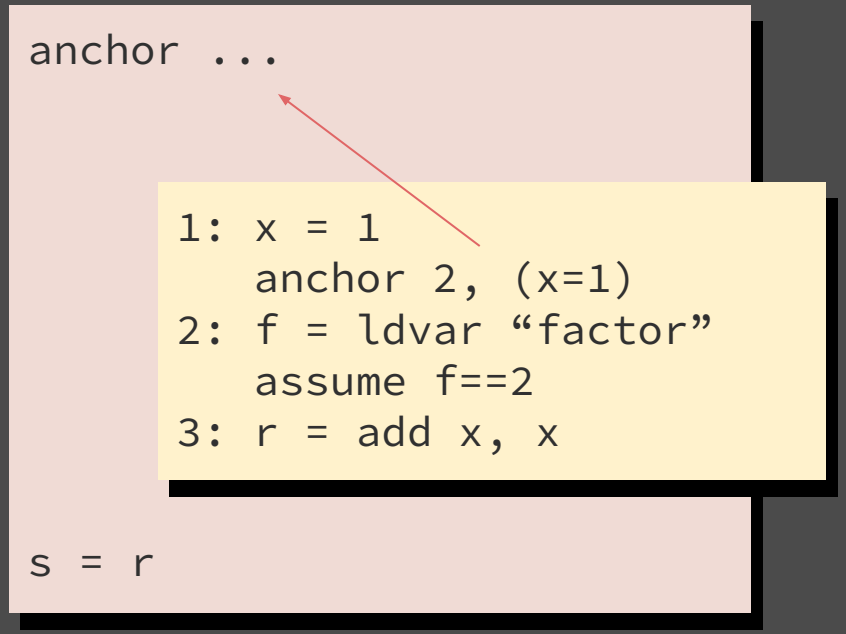
```
x * factor
```

```
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
```



Sourir Model IR

assume e^* **else** ξ $\tilde{\xi}^*$ assume instruction

ξ	::=	$F.V.L\ VA$	*target and varmap*
$\tilde{\xi}$	::=	$F.V.L\ x\ VA$	*extra continuation*
VA	::=	$[x_1 = e_1, .., x_n = e_n]$	*varmap*

- Traditional Optimizations: constant propagation, unreachable code elimination, and function inlining.
- Unrestricted deoptimization, predicate hoisting, assume composition
- Limitations: Correctness proof over pregenerated versions, fixed in CoreJIT. No native code generation.

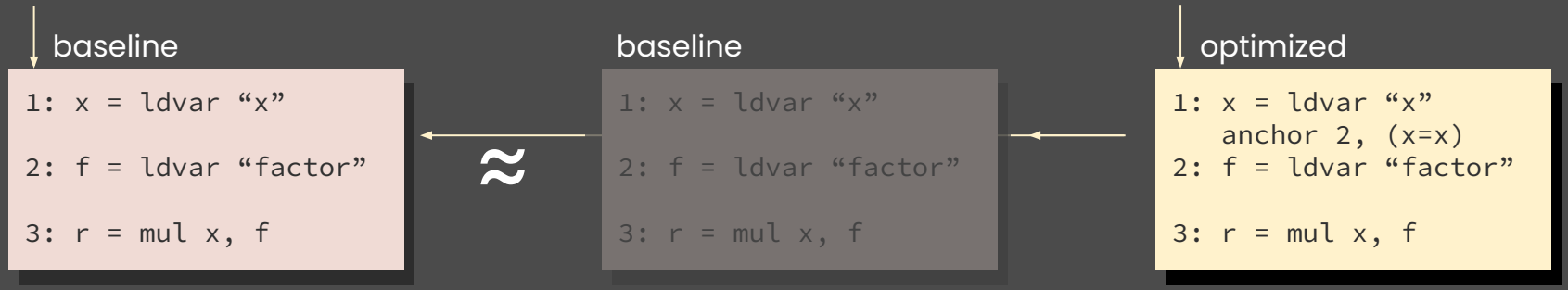
[AssumePass]

$$\frac{I(L) = \textbf{assume}\ e^*\ \textbf{else}\ \xi\ \tilde{\xi}^* \quad \forall m, M\ E\ e_m \to \textbf{true}}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\tau} \langle P\ I\ (L{+}1)\ K^*\ M\ E\rangle}$$

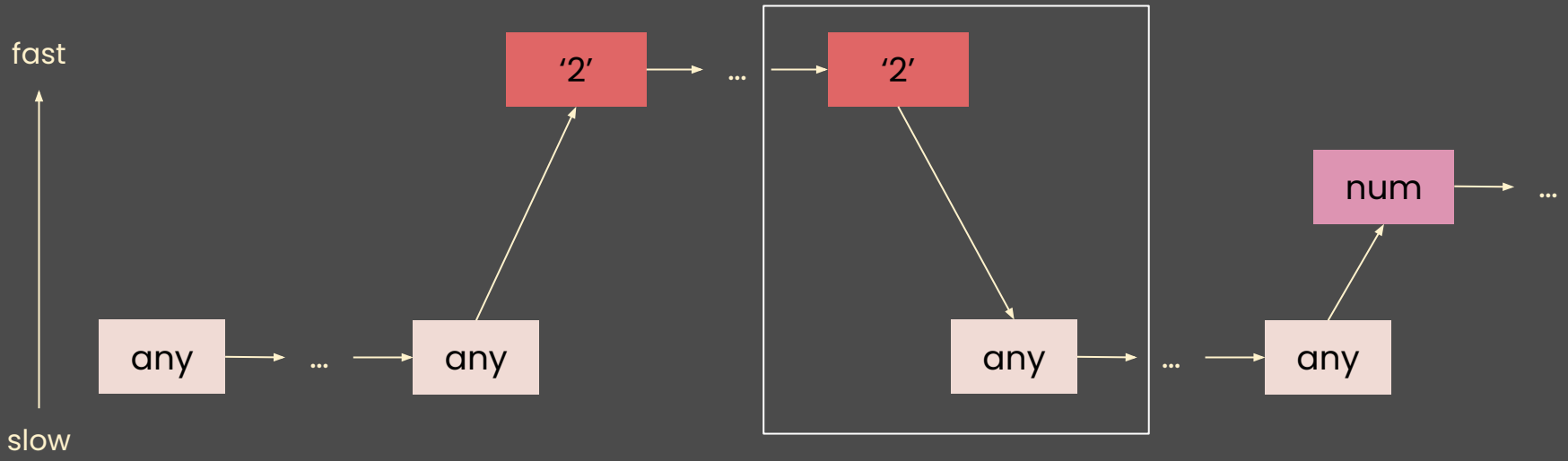
[AssumeDeopt]

$$\frac{I(L) = \textbf{assume}\ e^*\ \textbf{else}\ \xi\ \tilde{\xi}^* \quad \neg(\forall m, M\ E\ e_m \to \textbf{true})}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\tau} \text{deoptimize}(\langle P\ I\ L\ K^*\ M\ E\rangle, \xi, \tilde{\xi}^*)}$$

Proof Structure

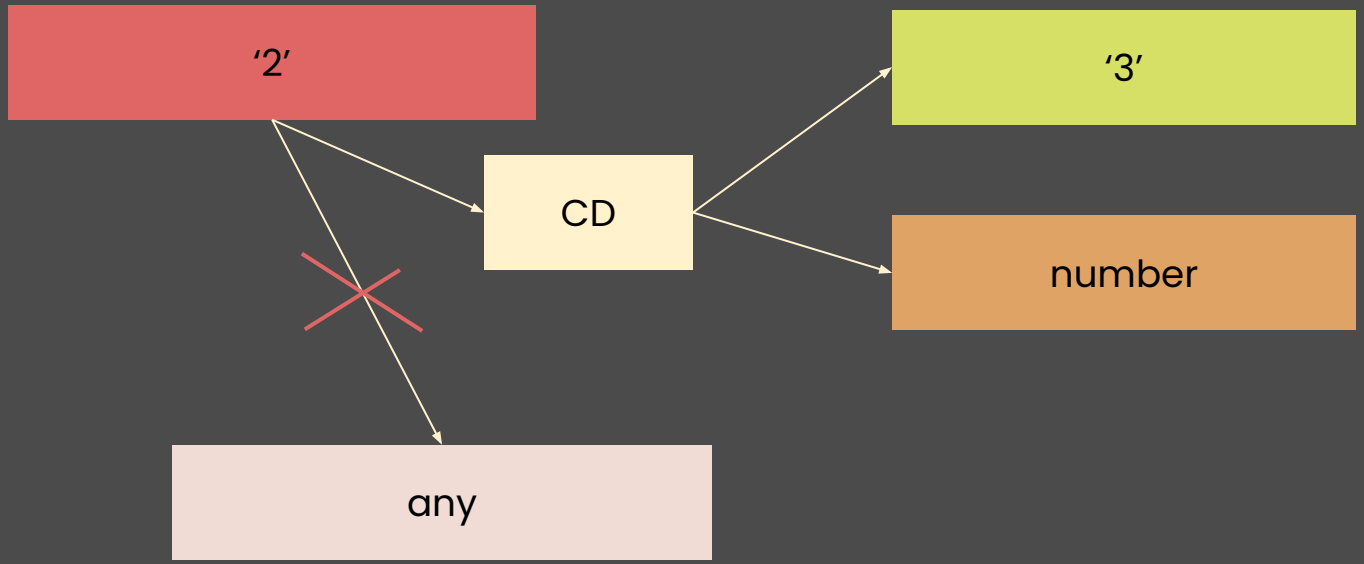


```
scale ← function(x) {
  # assume factor == ?
  x * factor
}
```



Deoptless: Assume + Context Dispatch

```
scale ← function(x) {
  # assume factor == ?
  x * factor
}
```



[PLDI’22] Olivier Flückiger, Jan Ječmen, Sebastián Krynski, and Jan Vitek

3. Ř

- 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.

[DLS'19] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek
[DLS'20] Olivier Flückiger, Sebastián Krynski, Andreas Wälchli, and Jan Vitek

Ř Eval

Assume and context dispatch provide the **basis** for optimizations based on run-time assumptions in a **competitive** just-in-time compiler.

	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

Assumptions and Speculations…

Context Dispatch

```
scale(1)
scale(dl())
```

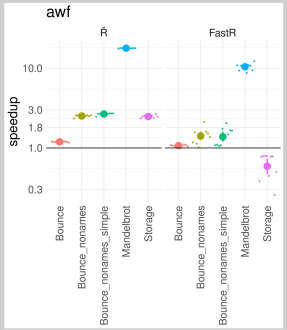
$C_1 \stackrel{\text{def}}{=} x : \text{num}[1]$

$C_2 \stackrel{\text{def}}{=} x : ?$

Assume Formalized

$$\frac{I(L) = \textbf{assume}\ e^*\ \textbf{else}\ \xi\ \tilde{\xi}^* \quad \neg(\forall m, M\ E\ e_m \rightarrow \textbf{true})}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\tau} \text{deoptimize}(\langle P\ I\ L\ K^*\ M\ E\rangle, \xi, \tilde{\xi}^*)} \quad [\text{AssumeDeopt}]$$

Ř



Assume and context dispatch provide the basis for optimizations based on run-time assumptions in a competitive just-in-time compiler.