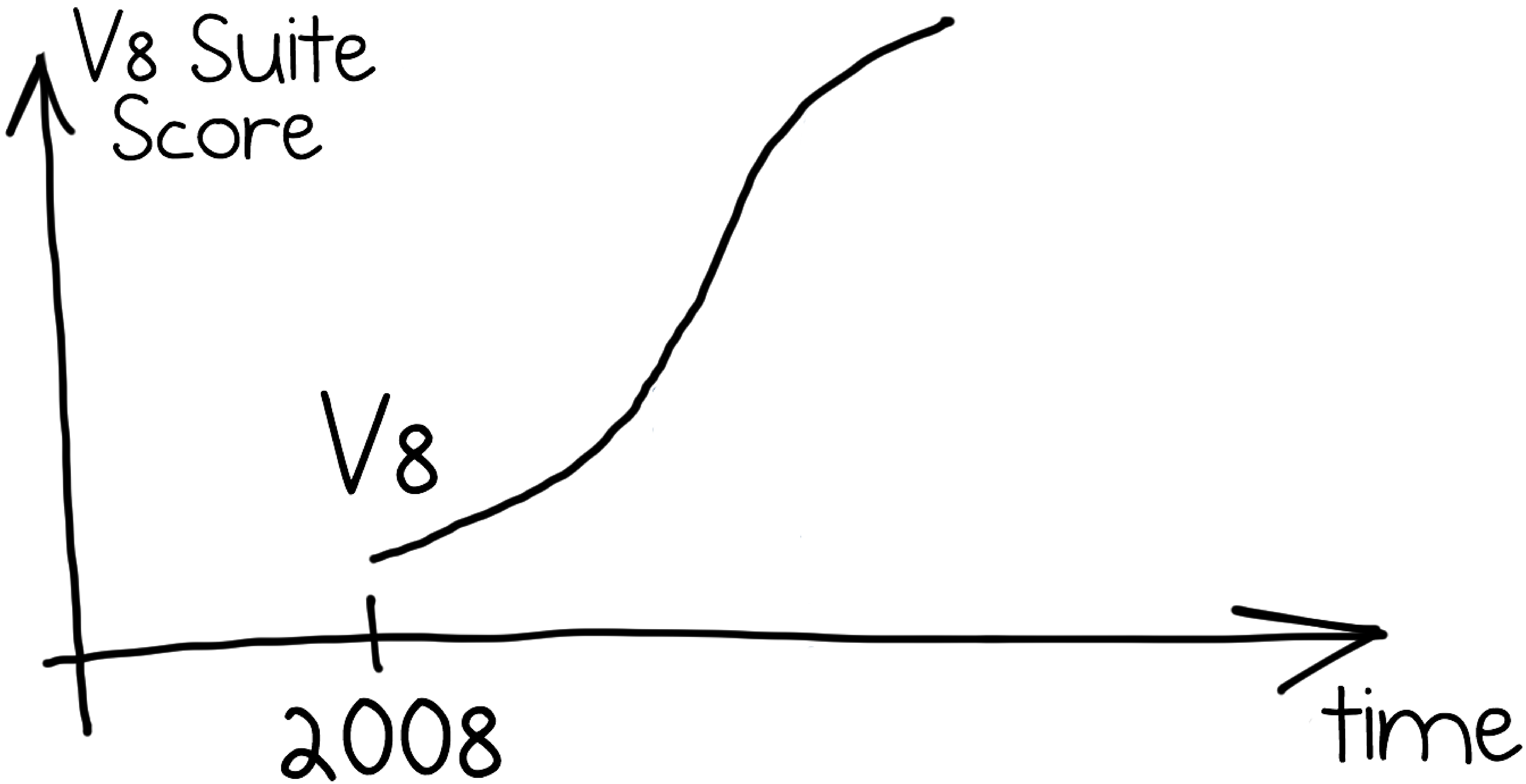
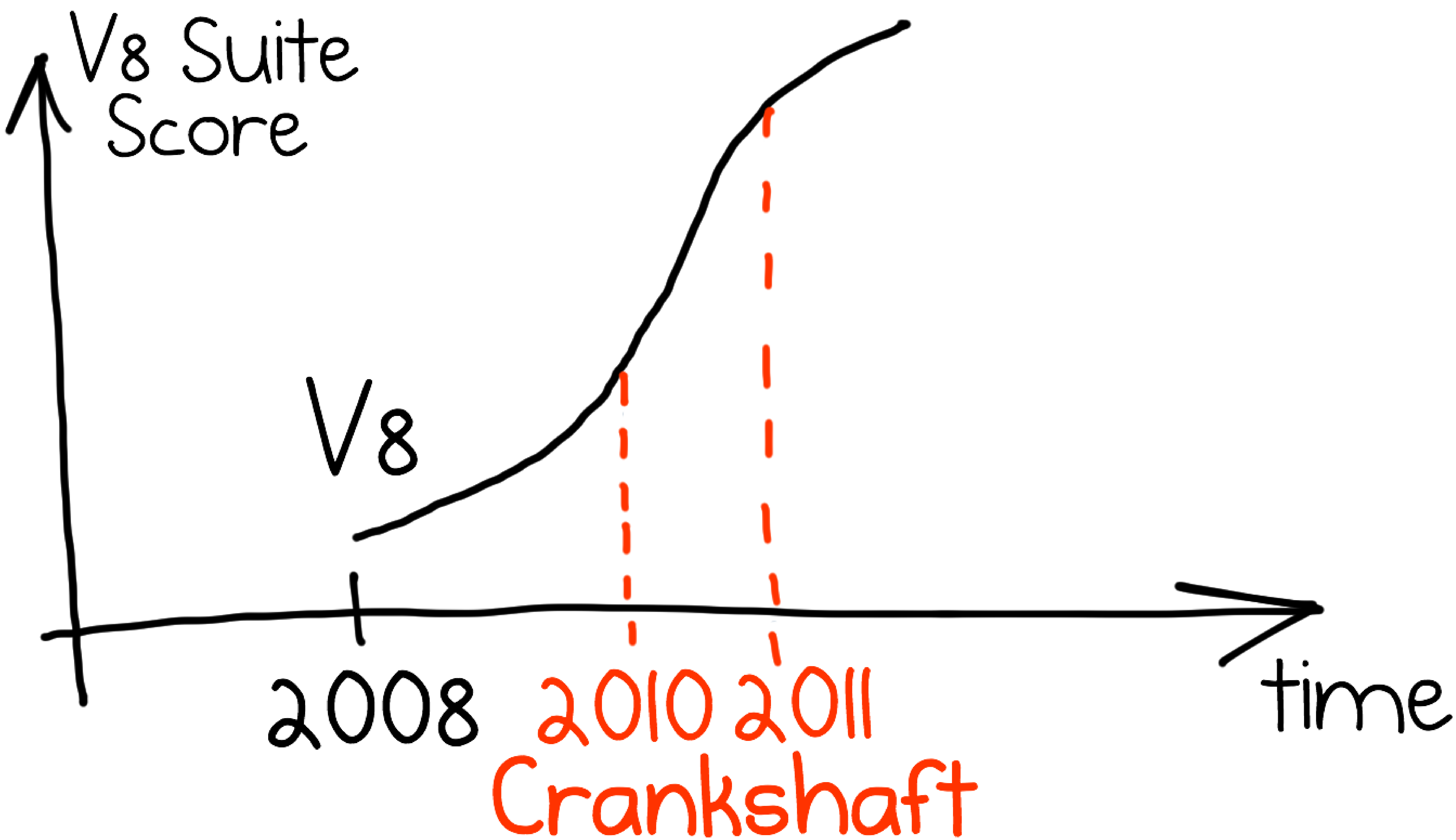
Building an optimizing compiler for Dart

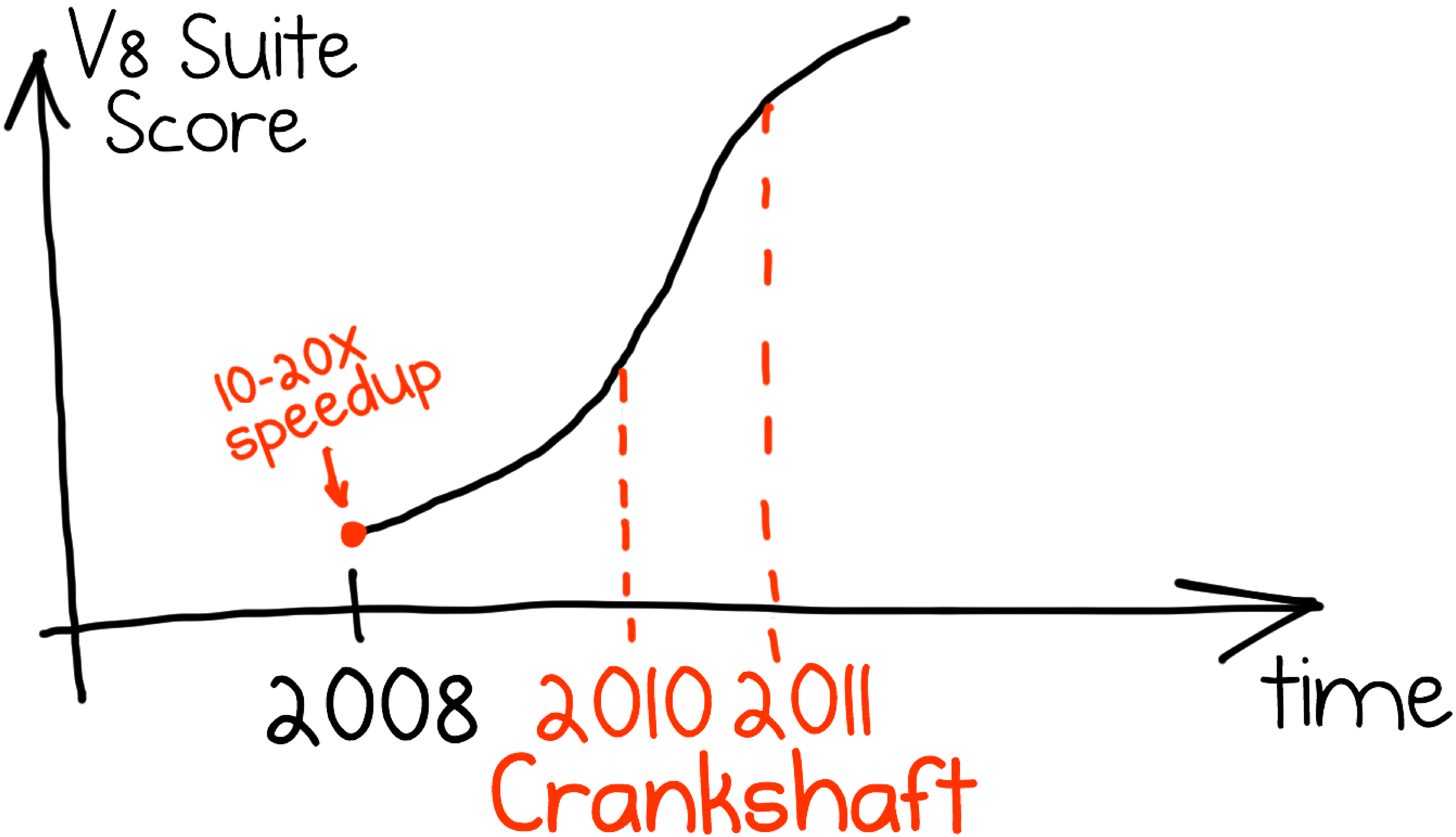
(with historical excursion into V8)

Vyacheslav Egorov

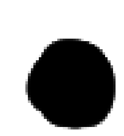
http://s3.mrale.ph/StrangeLoop2013.pdf


V8 Suite
Score
V8
2008
time


V8 Suite Score
V8
2008
2010
2011
Crankshaft
time


V8 Suite Score
10-20x speedup
2008
2010
2011
Crankshaft
time

```
obj.prop
obj[i]
obj.foo()
```



A

obj.prop
obj[i]
obj.foo()

[ecx+23]
[eax+edx*8+7]
0x582a70

A

B

obj.prop

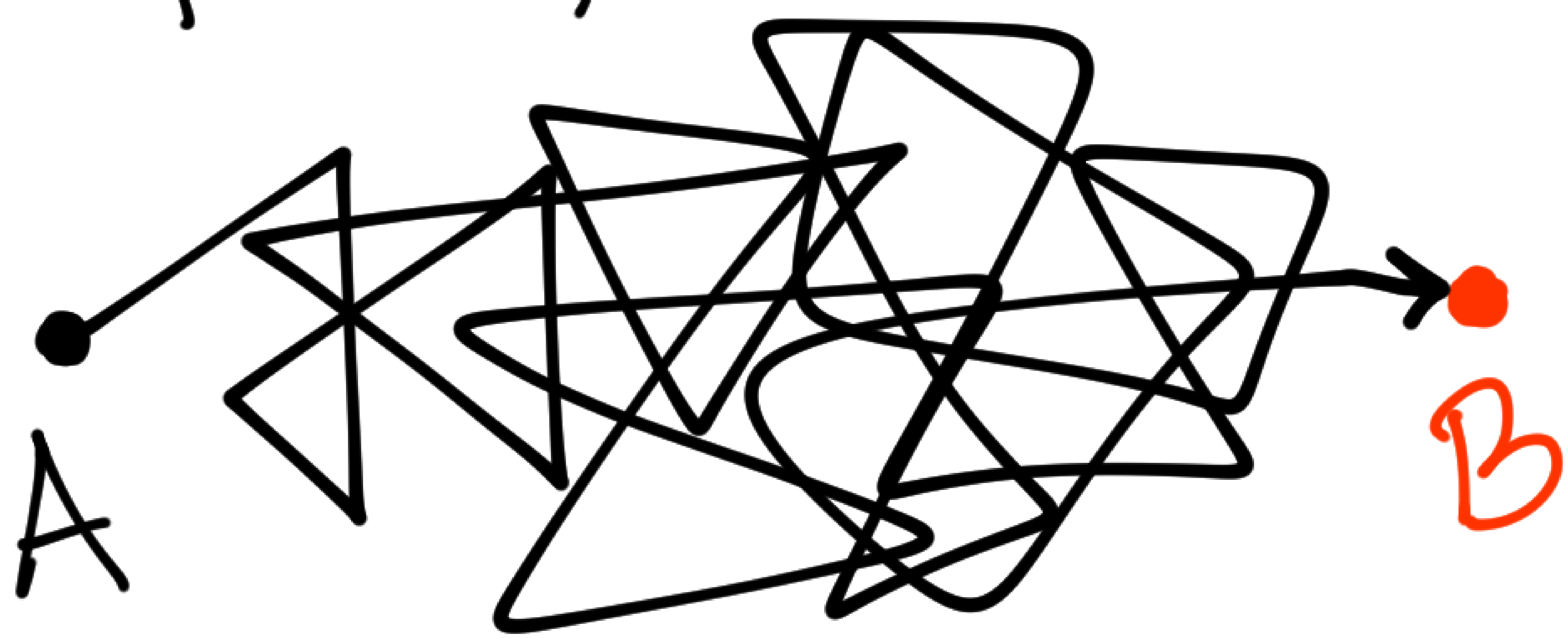
obj[i]

obj.foo()

[ecx+23]

[eax+edx*8+7]

0x582a70



obj.prop

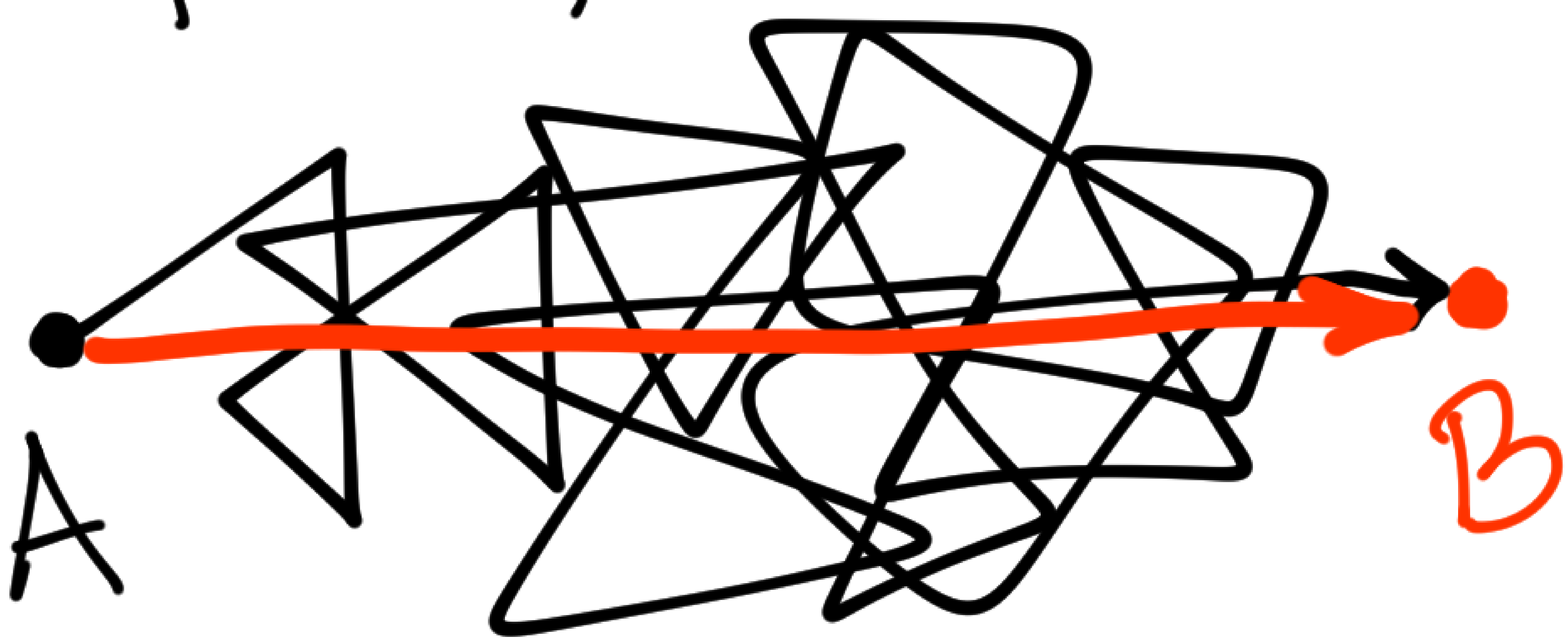
obj[i]

obj.foo()

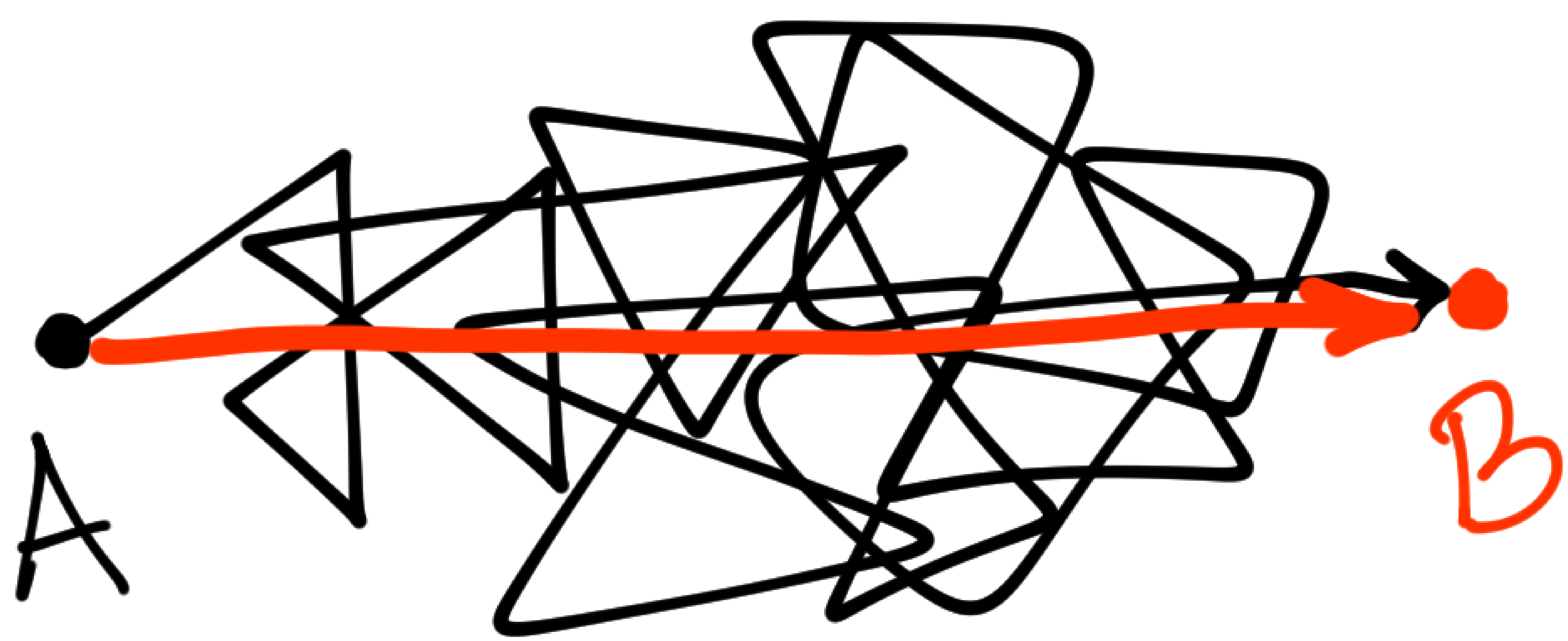
[ecx+23]

[eax+edx*8+7]

0x582a70



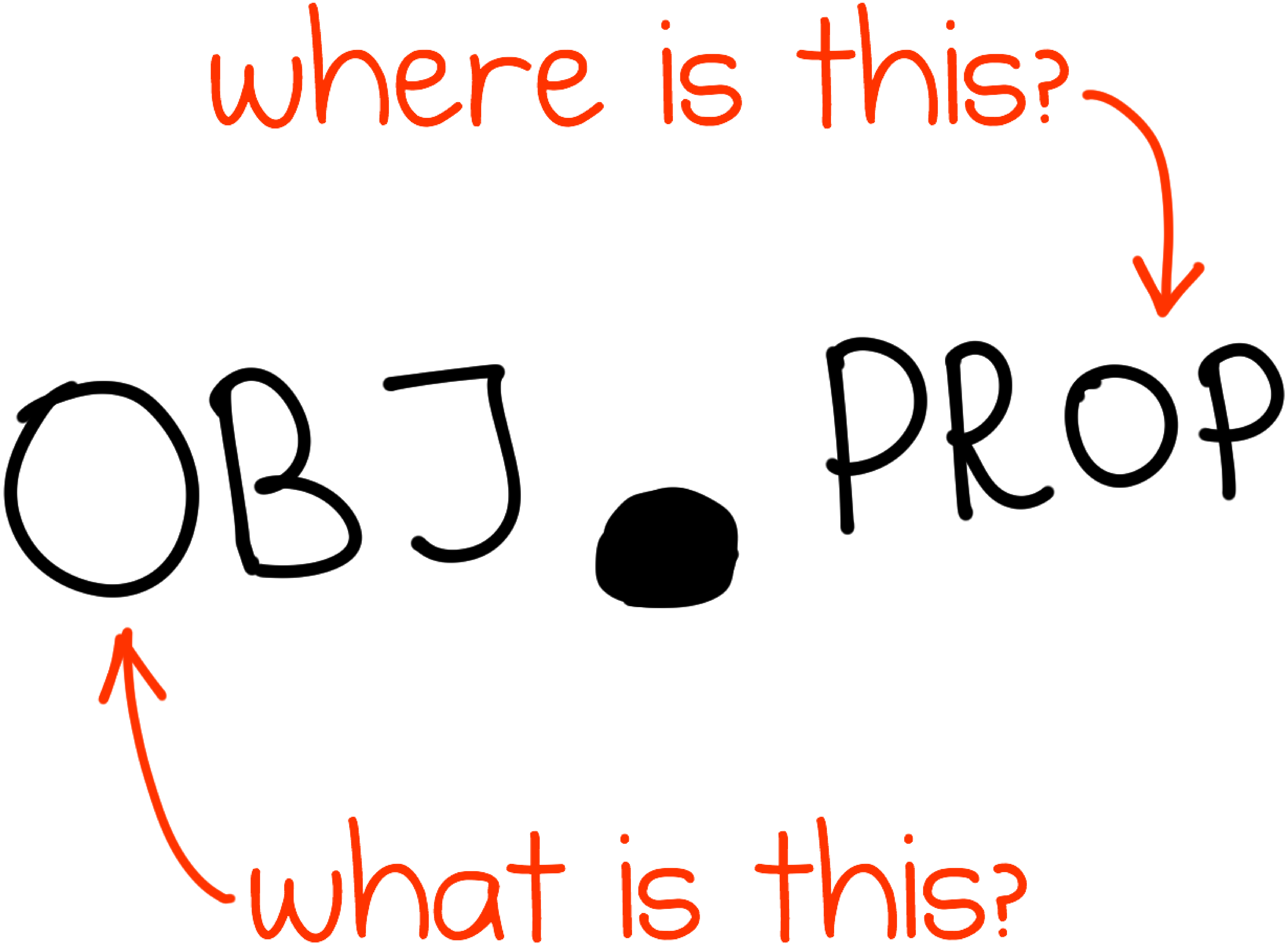
Optimizing compilation is the art of taking shortcuts

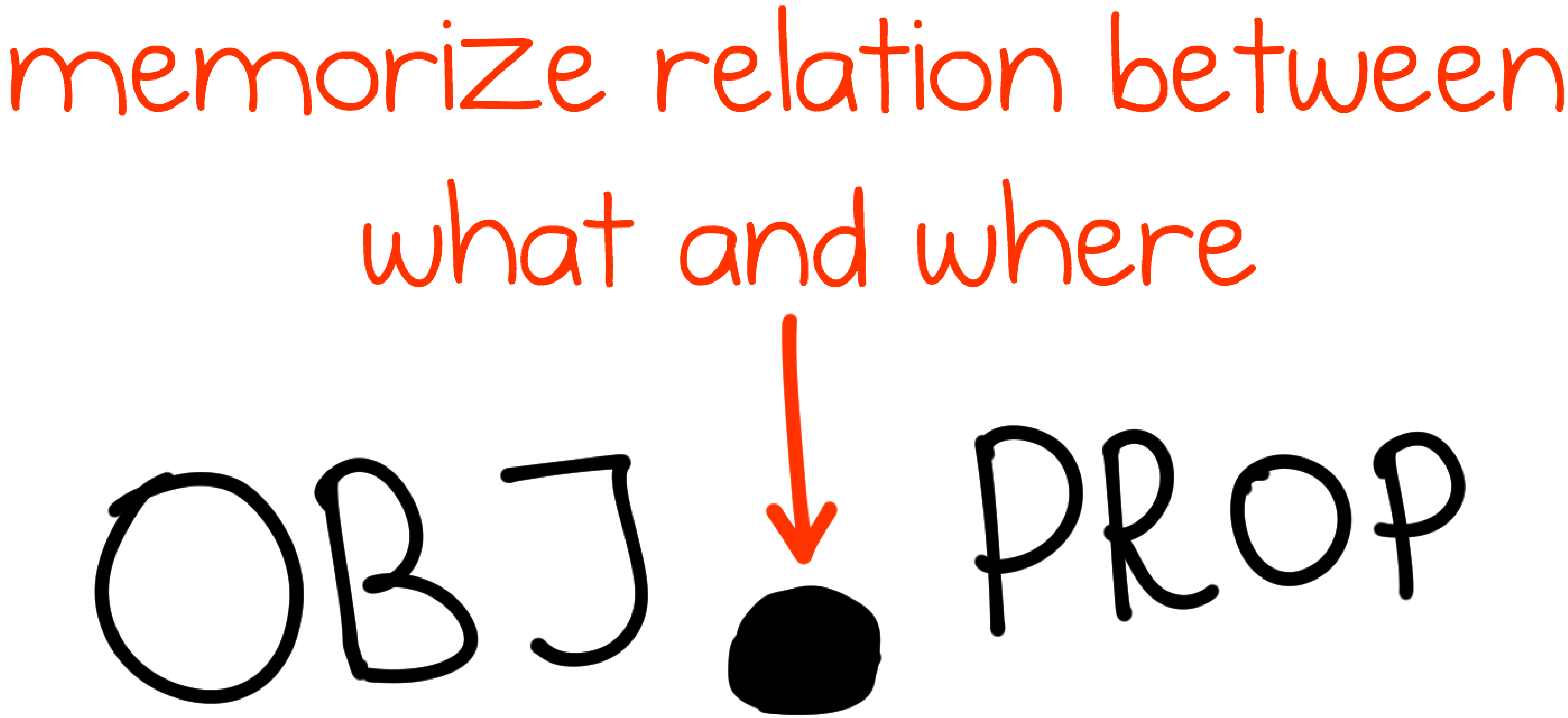


- Representation
- Resolution
- Redundancy

OBJ . PROP

OBJ . PROP


where is this?
OBJ . PROP
what is this?

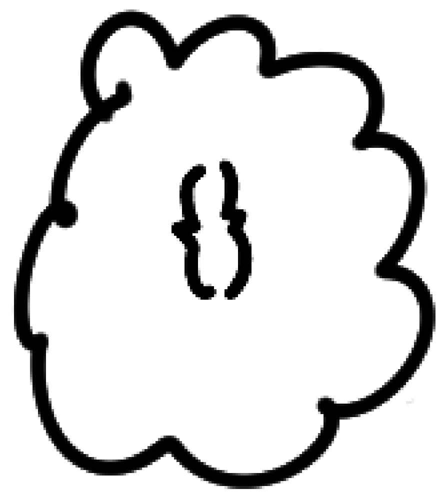

memorize relation between
what and where
OBJ . PROP

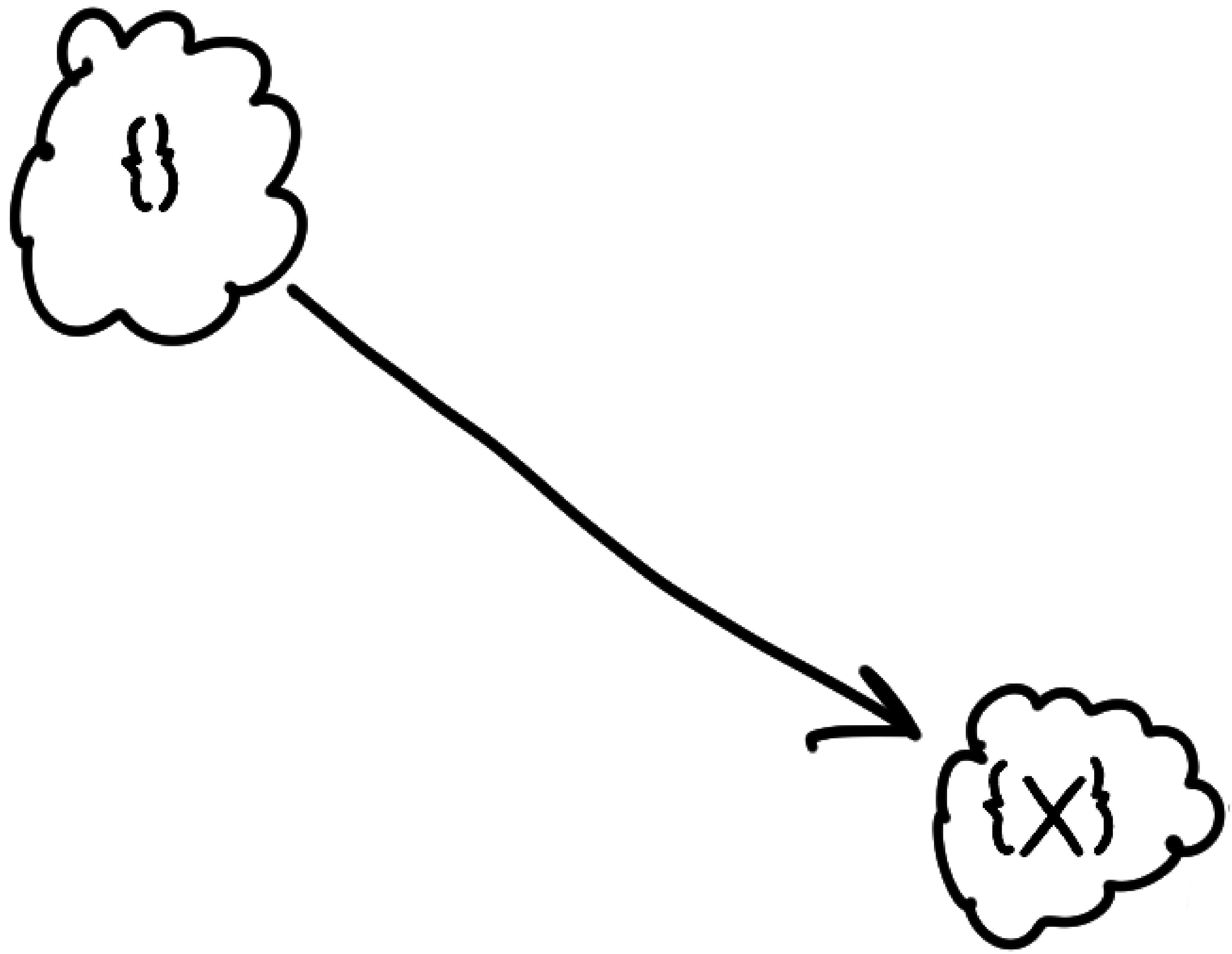
memorize relation between
what and where

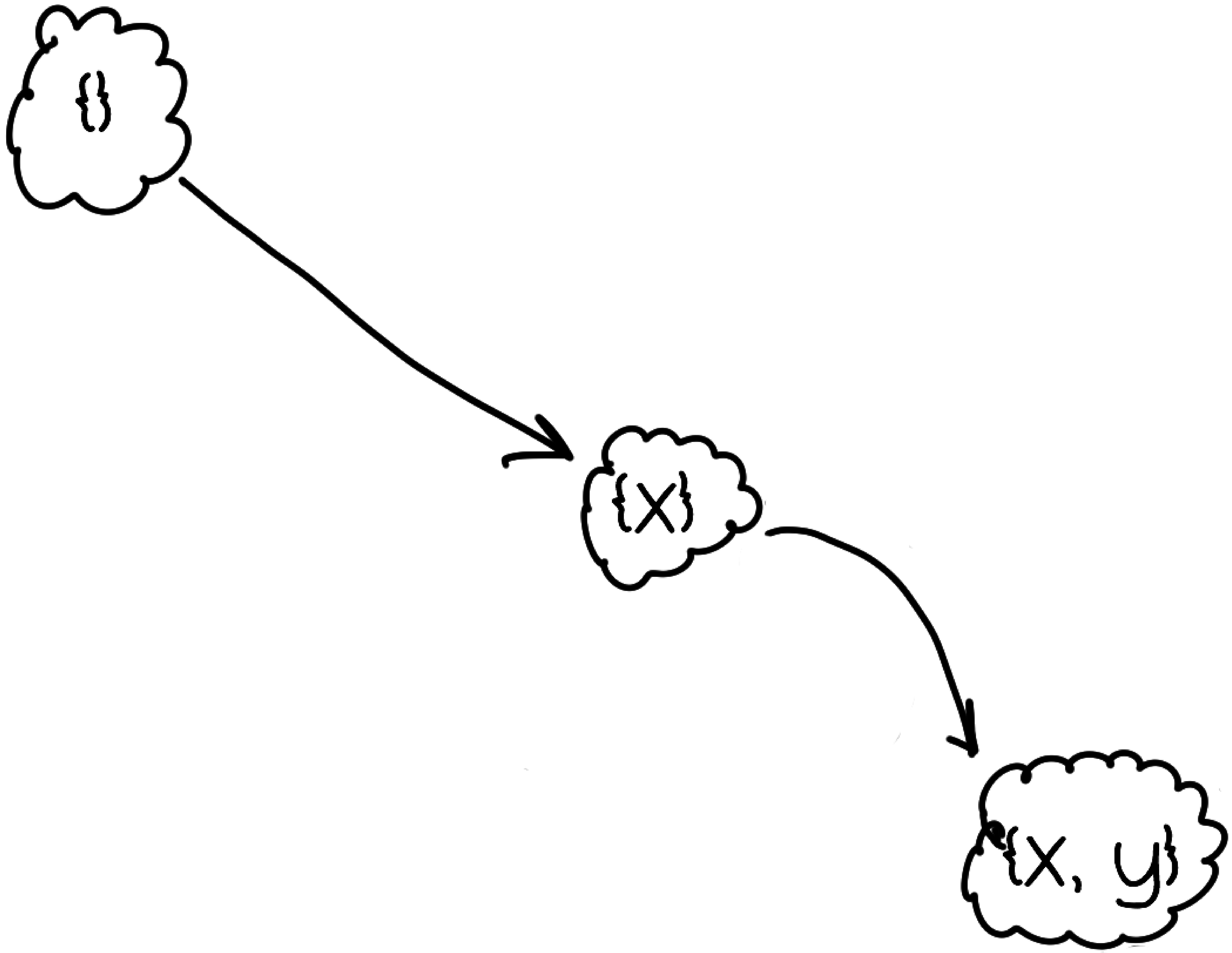
OBJ.PROP

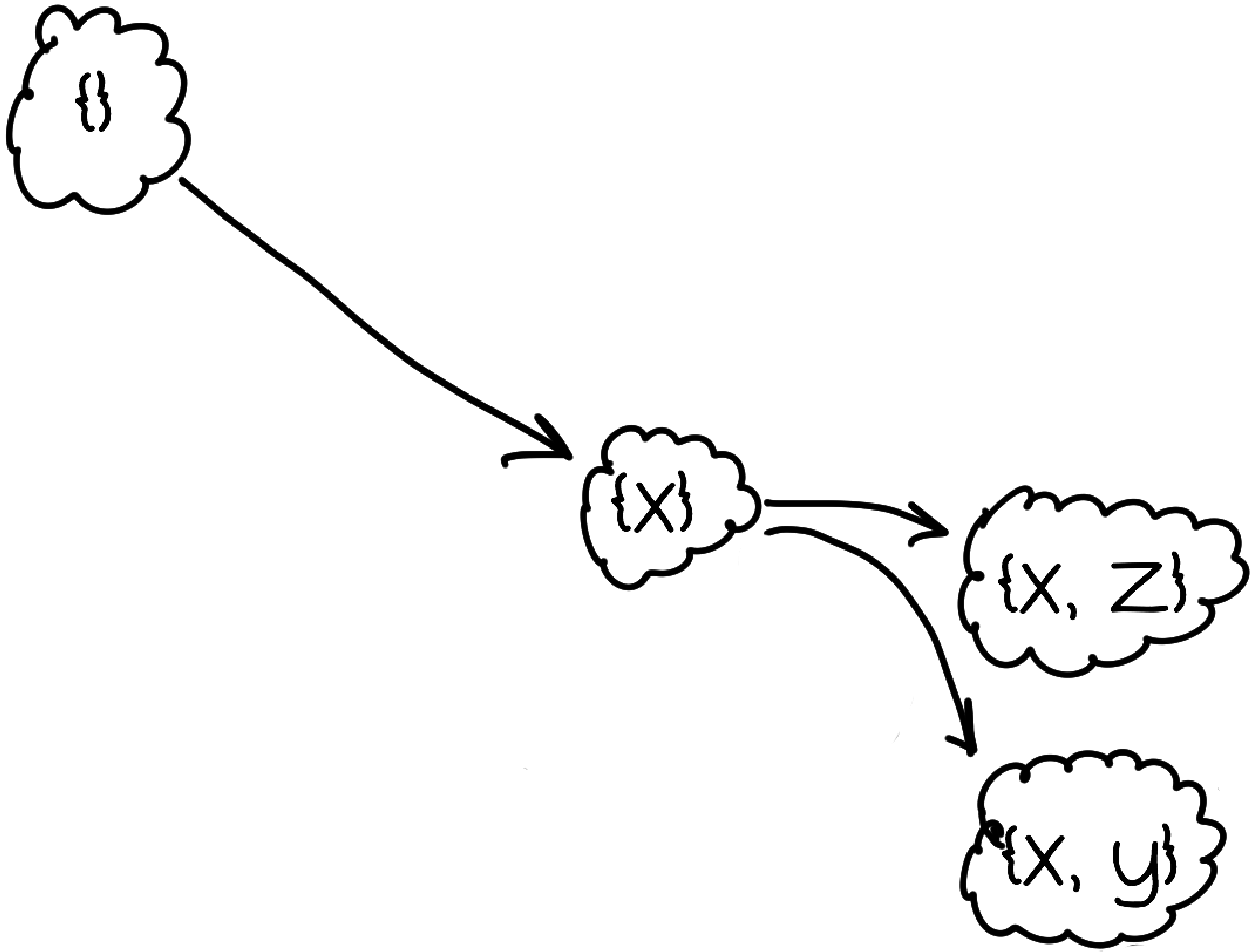
Inline Caching

Object

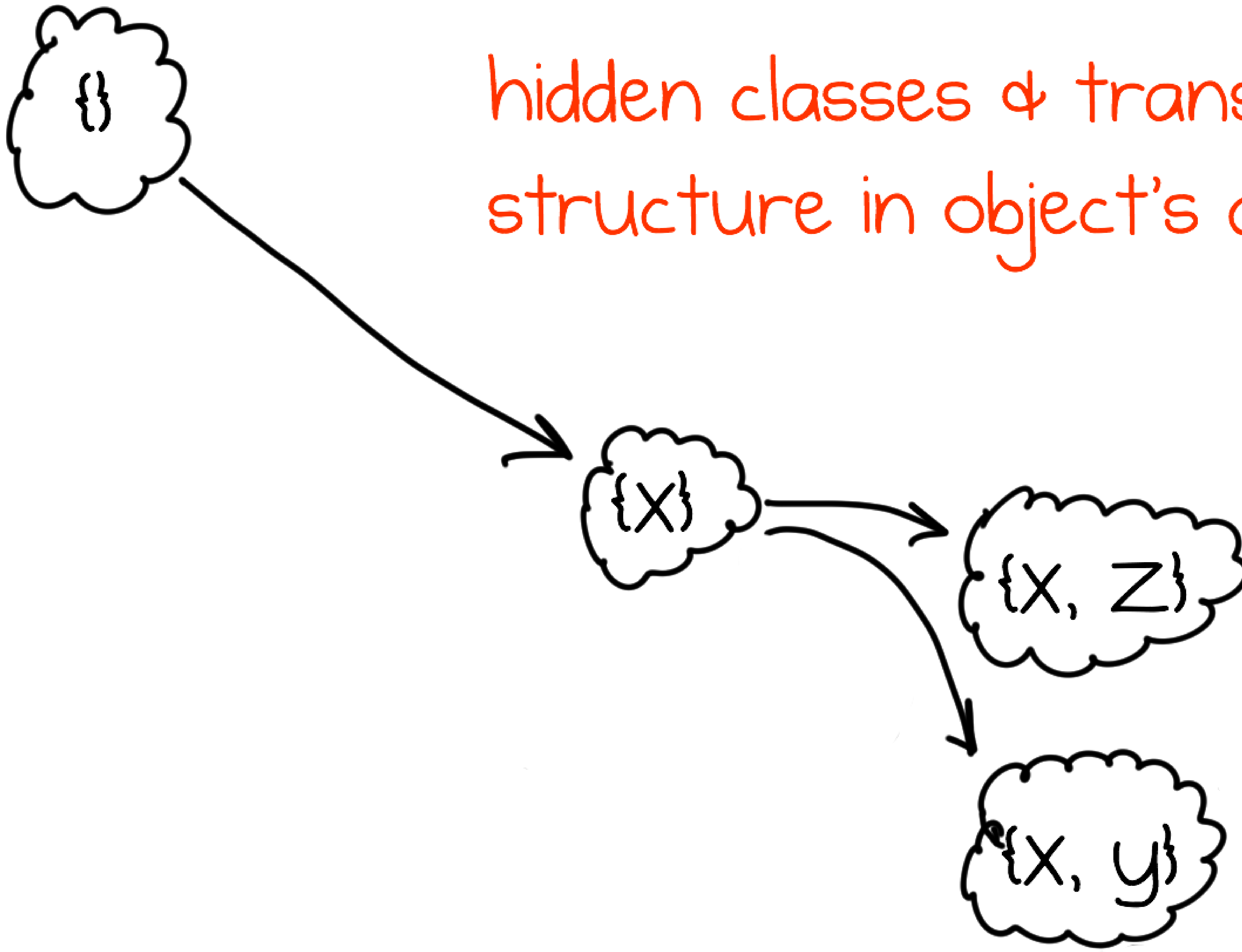



{}
{X}

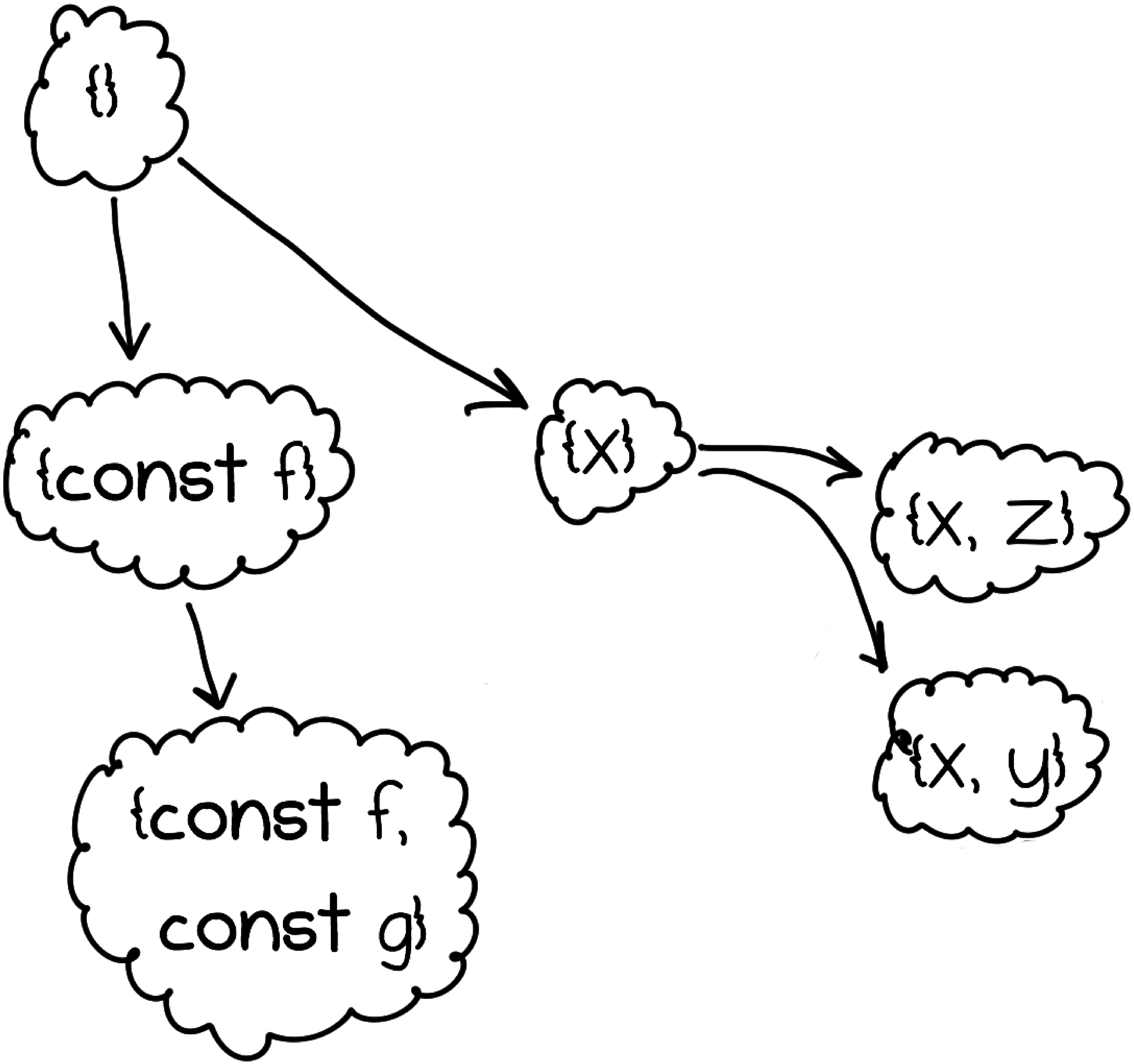

{}
{x}
{x, y}

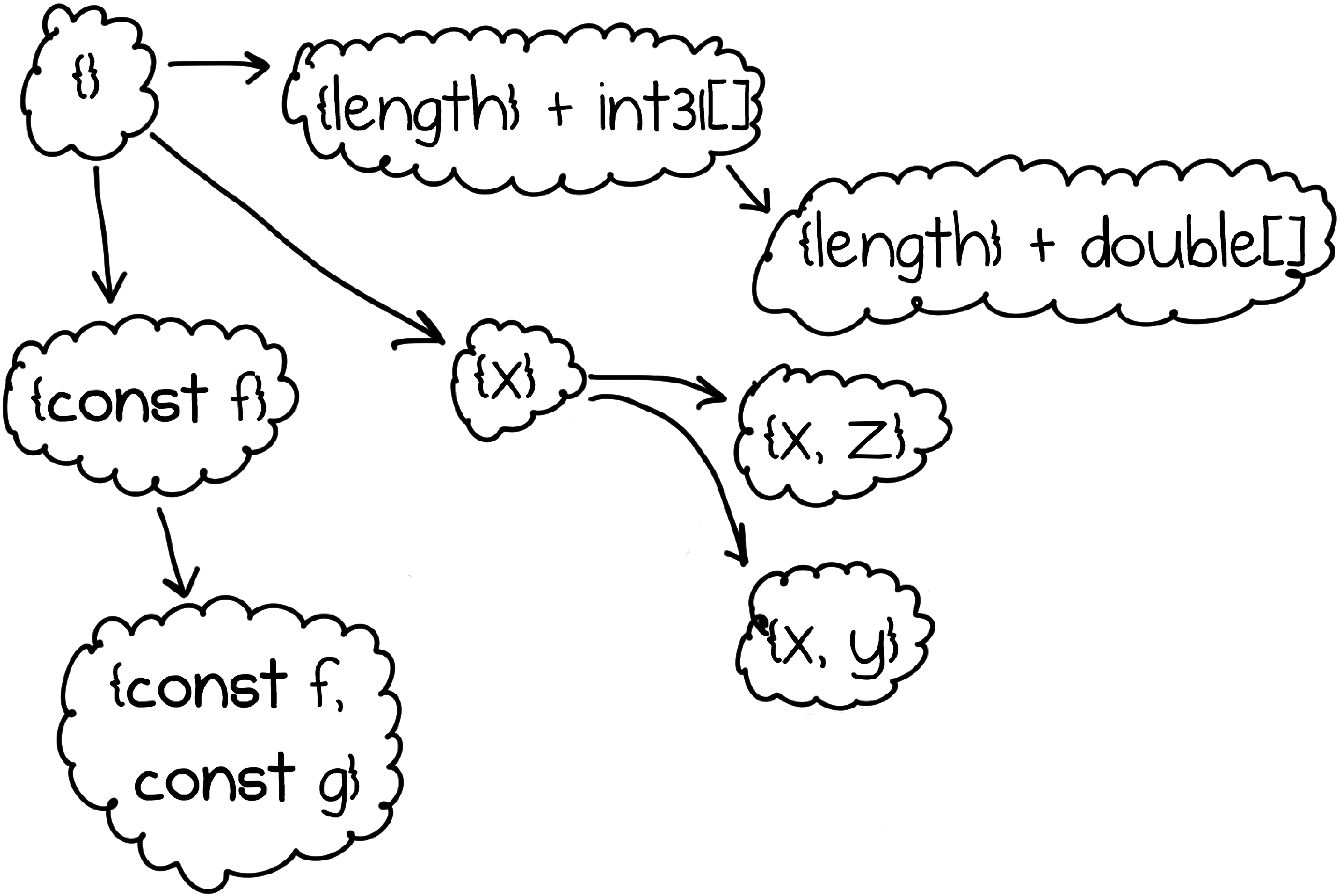

{}
{x}
{x, z}
{x, y}

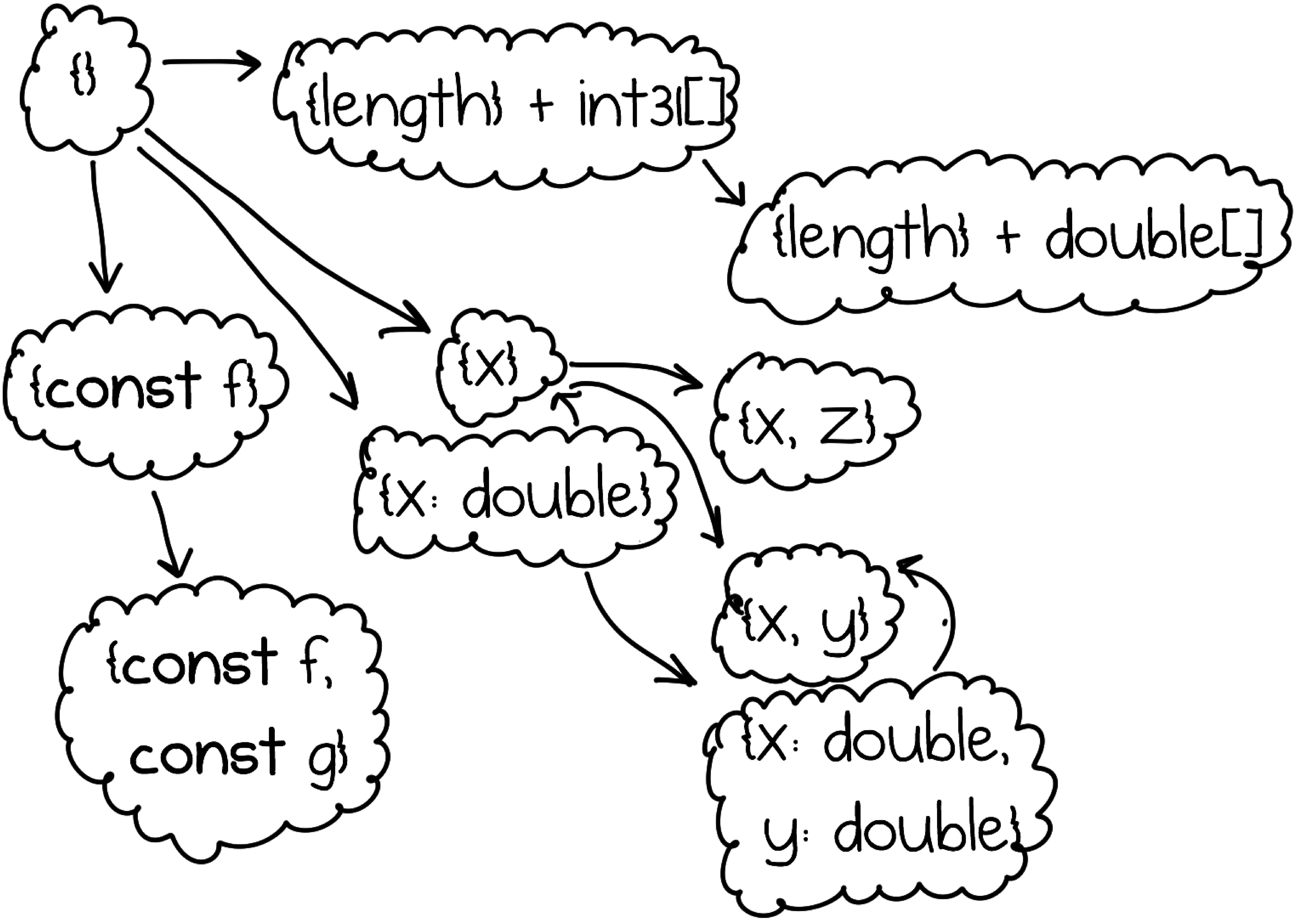
hidden classes & transitions reveal structure in object's dynamic history



objects constructed in the same way should have the same hidden class


{}
{const f}
{const f,
const g}
{x}
{x, z}
{x, y}


{}
{length} + int31[]
{length} + double[]
{const f}
{const f, const g}
{x}
{x, z}
{x, y}


{}
{length} + int31[]
{length} + double[]
{const f}
{const f,
const g}
{x}
{x: double}
{x, z}
{x, y}
{x: double,
y: double}

very powerful
and
very complex

(affects everything: GC, compiler, runtime, built-ins) very powerful

and

very complex

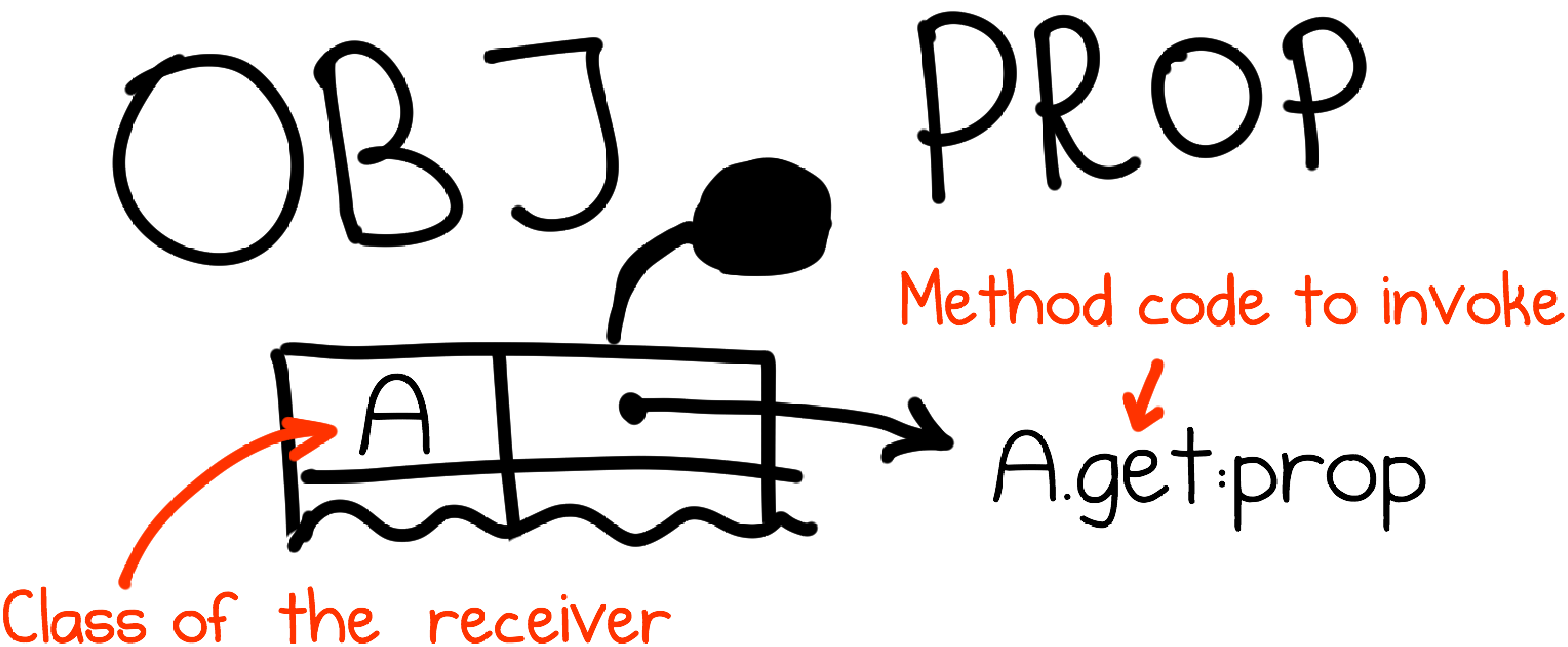
(affects everything: GC, compiler, runtime, built-ins)

Fortunately Dart has static class declarations

In Dart VM

OBJ . PROP

In Dart VM


OBJ . PROP
A
Method code to invoke
A.get:prop
Class of the receiver

In V8

OBJ.PROP

In V8

OBJ.PROP

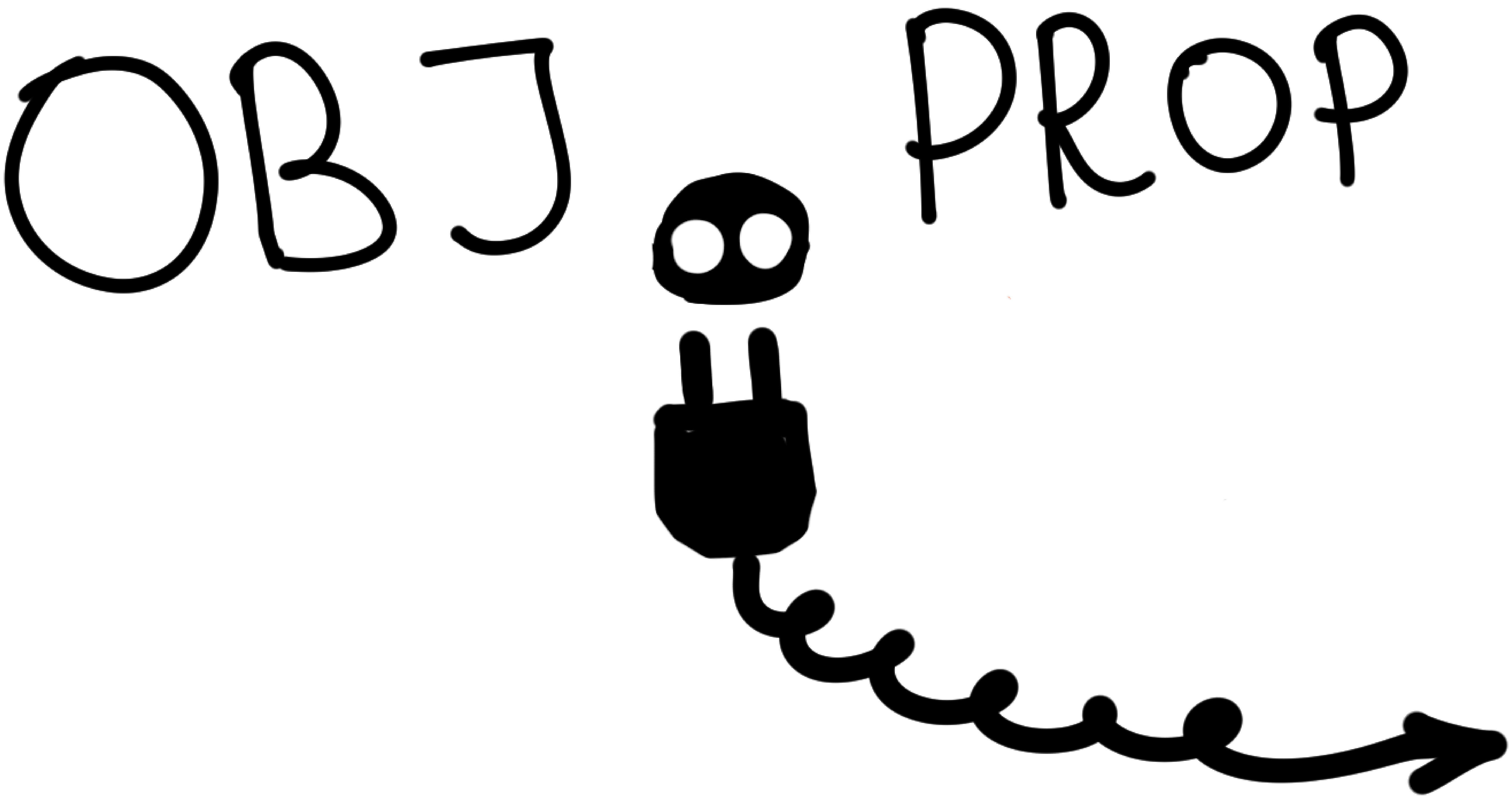
In V8 (American Version)

OBJ . PROP

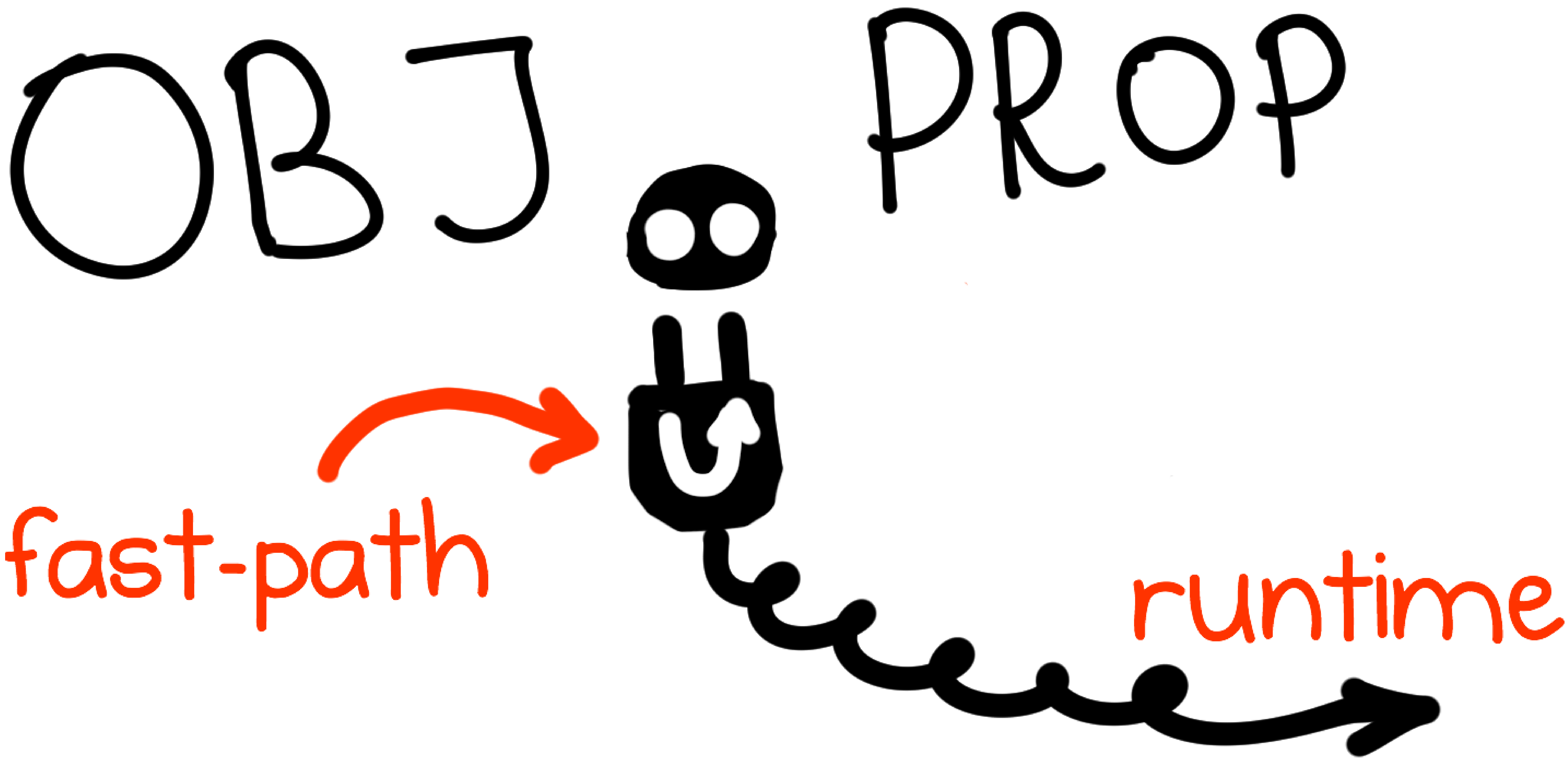
In V8

OBJ . PROP

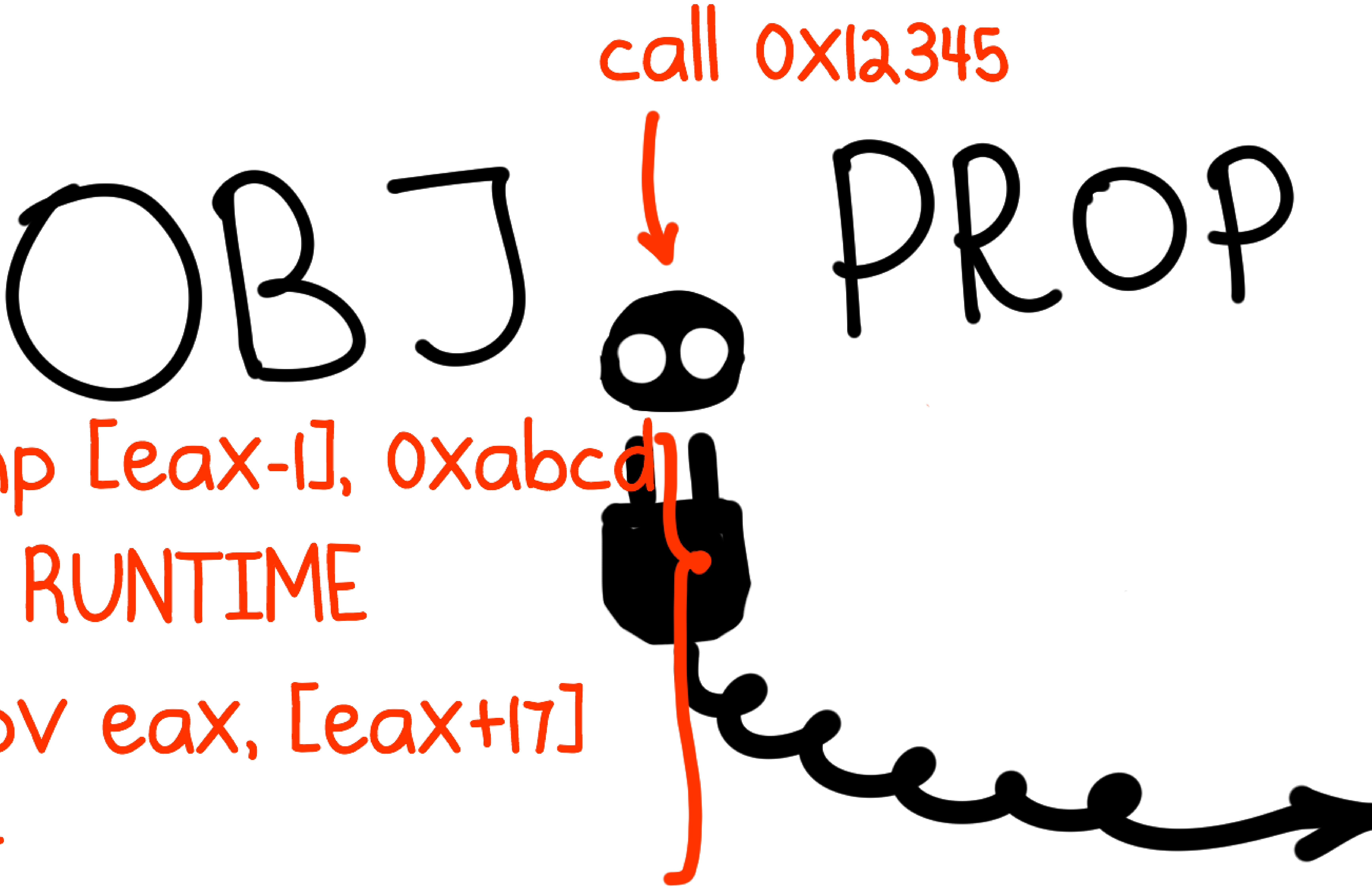
In V8



In V8



In V8




```
cmp [eax-1], 0xabcd
jne RUNTIME
mov eax, [eax+17]
ret
```

In V8

OBJ PROP

In V8

If all you have is an IC then everything looks like an IC-stub

In V8 inline caches designed to provide **peak performance** locally

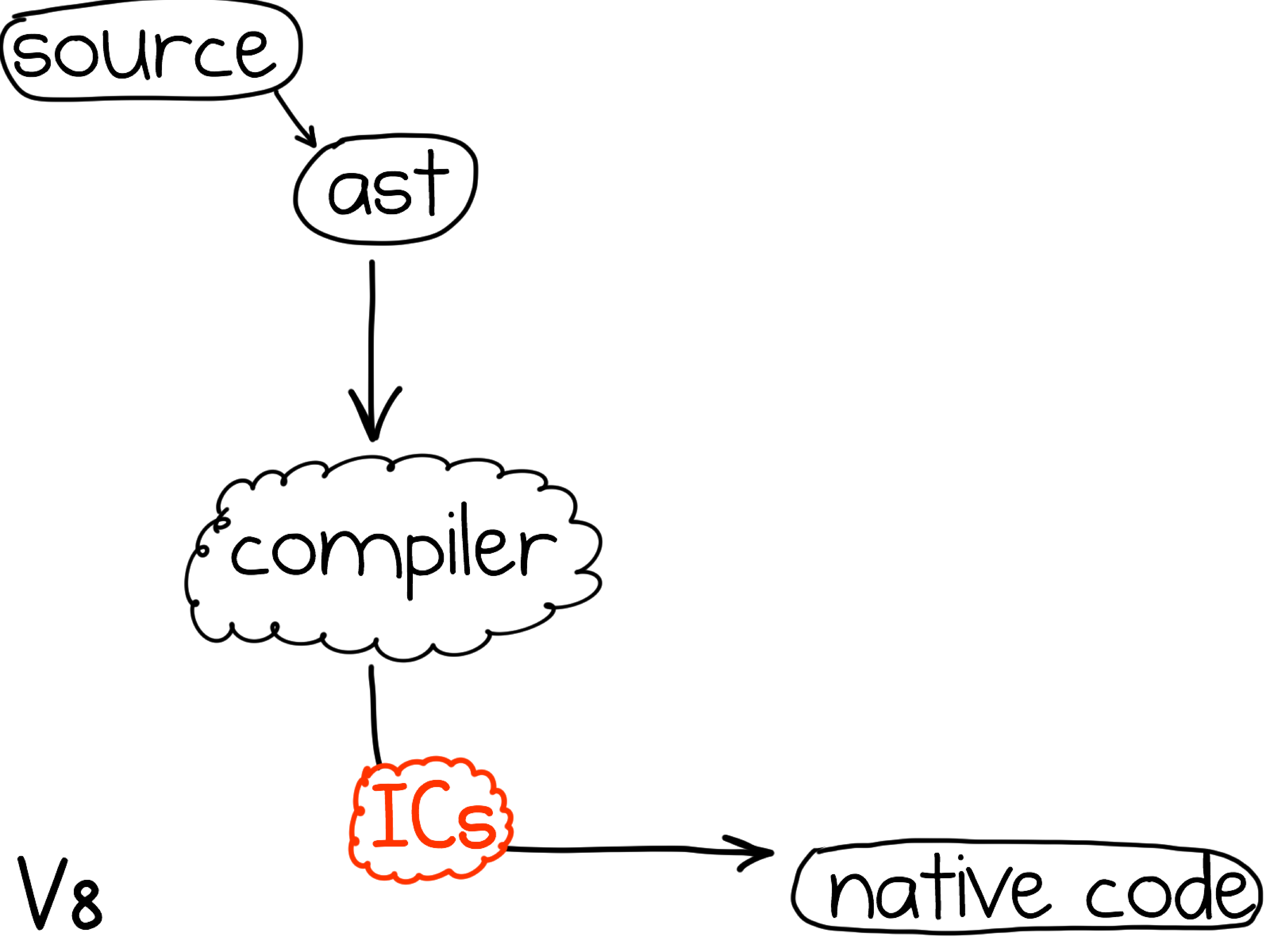
VS

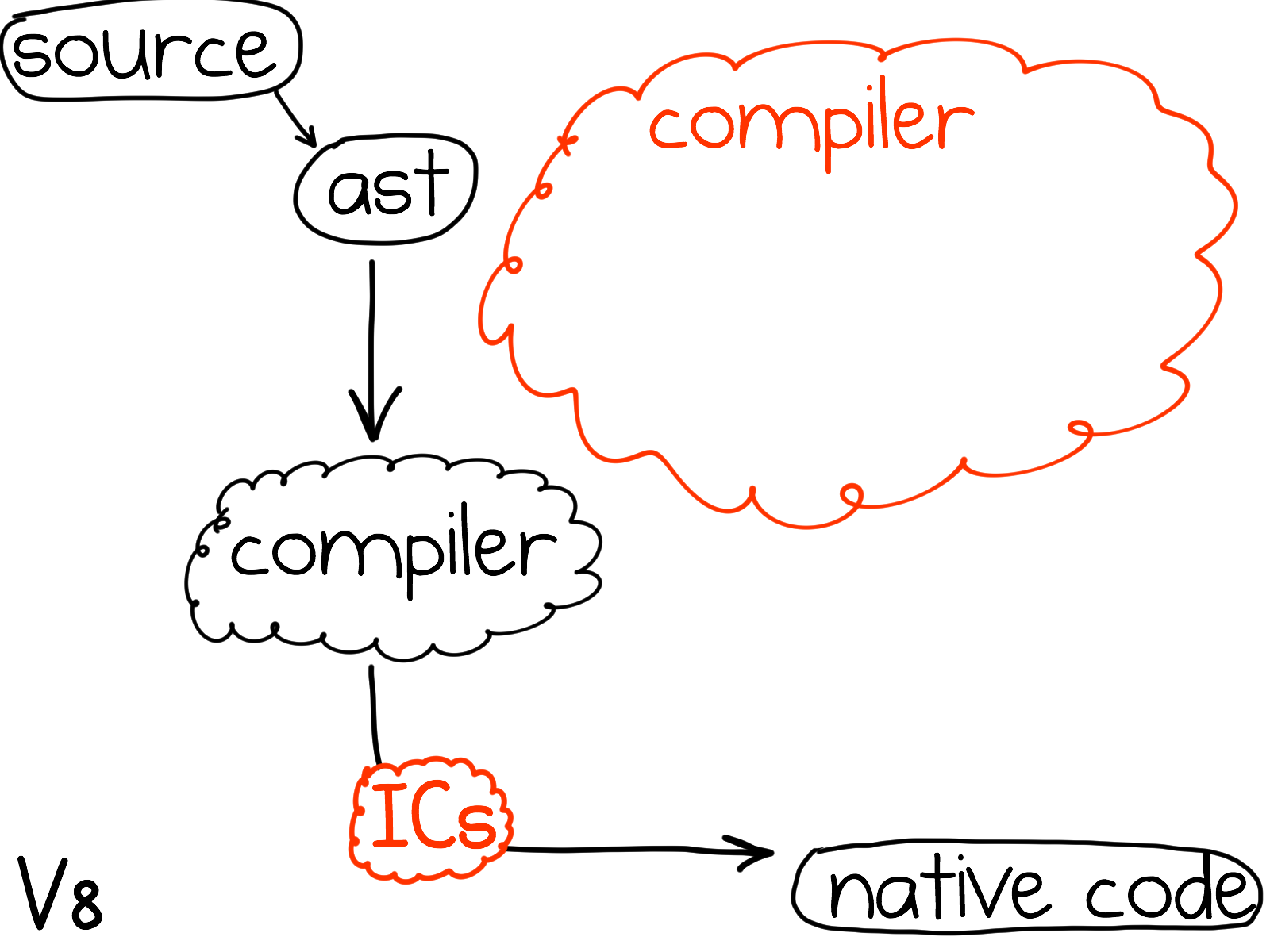
In Dart VM they simply collect **type feedback**, performance improvements are secondary

In V8 inline caches designed to provide peak performance locally

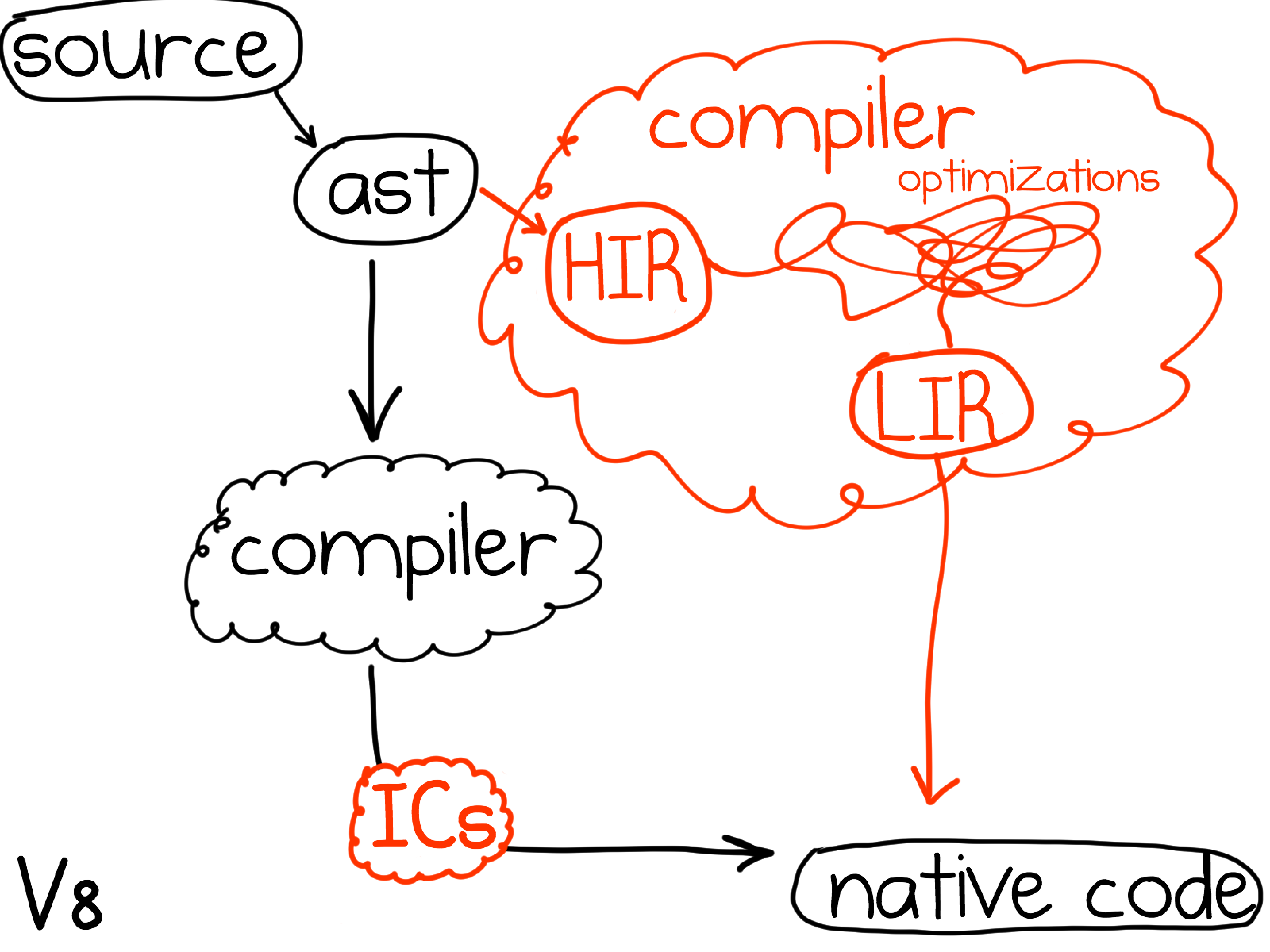
VS

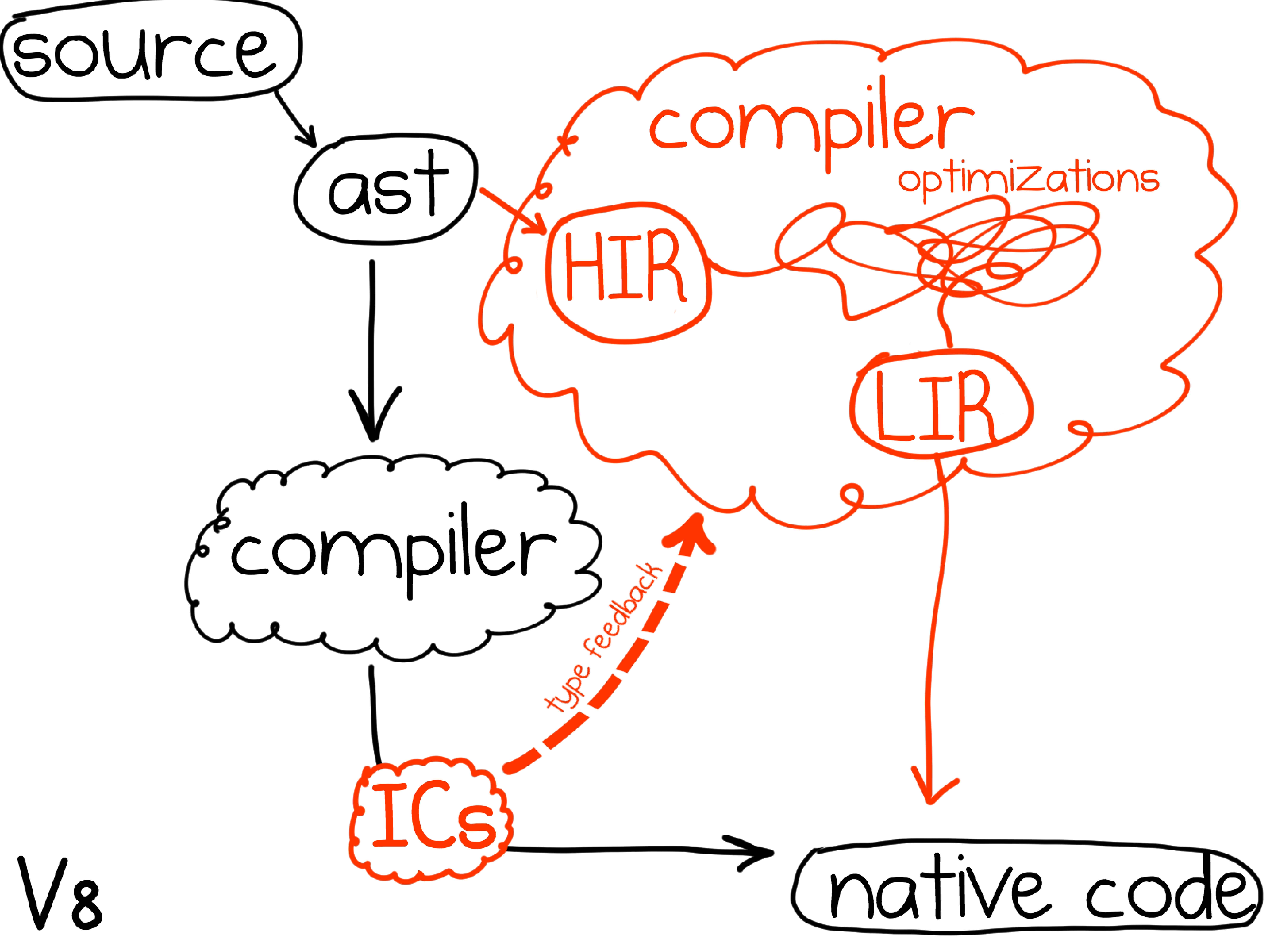
In Dart VM they simply collect type feedback, performance improvements are secondary

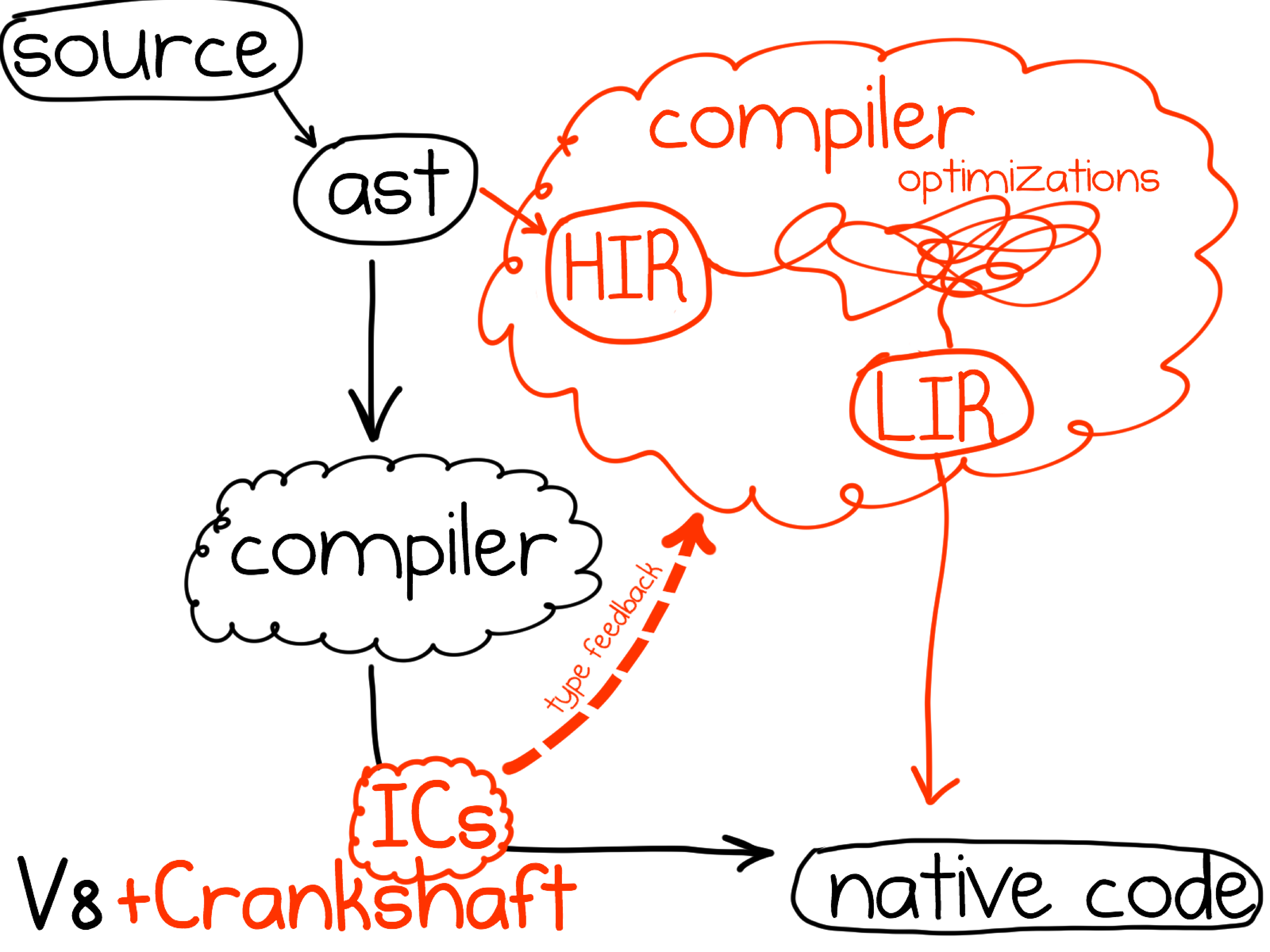

source
ast
compiler
ICs
native code
V8

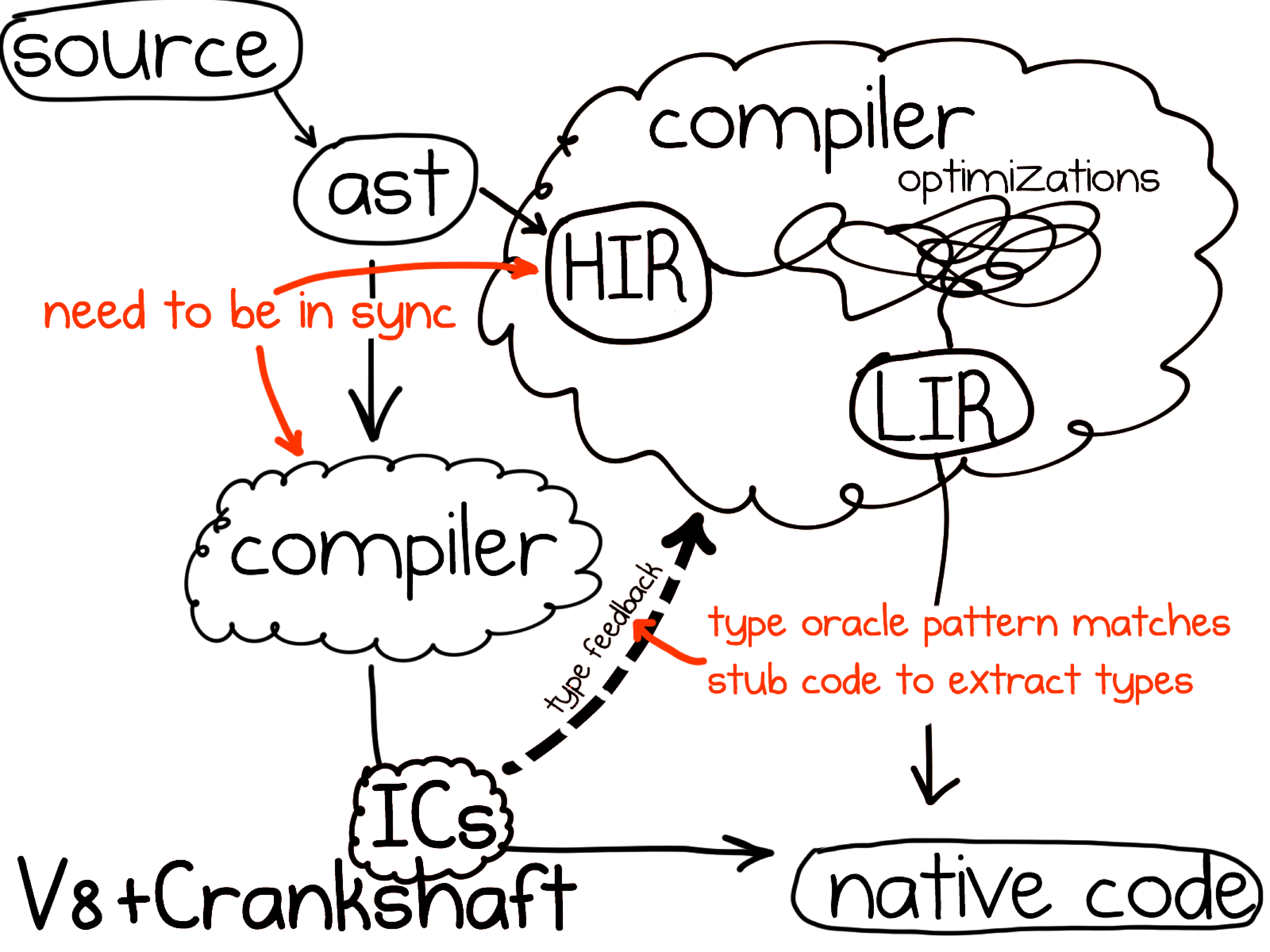


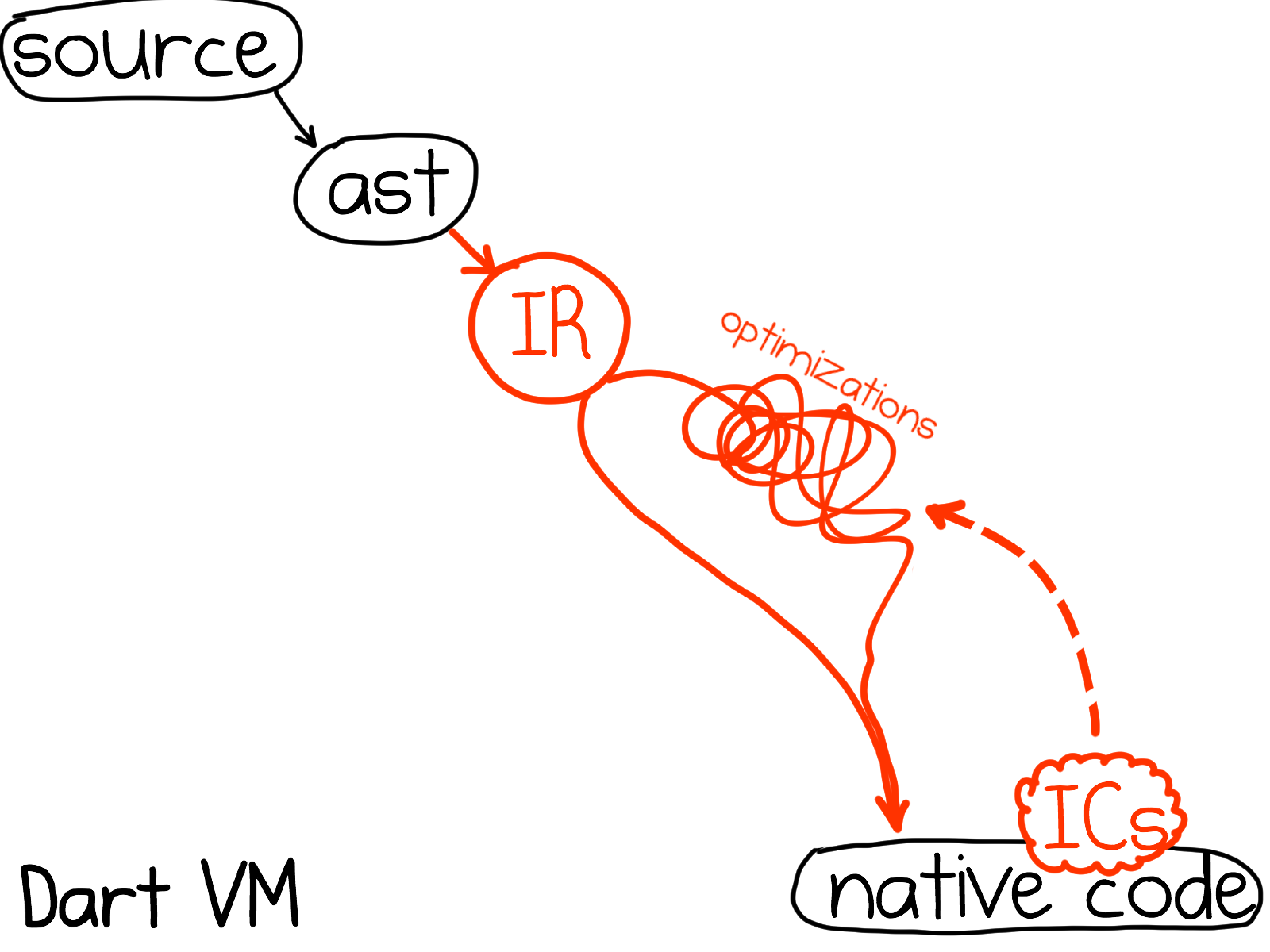
source
ast
compiler
compiler
ICs
native code
V8

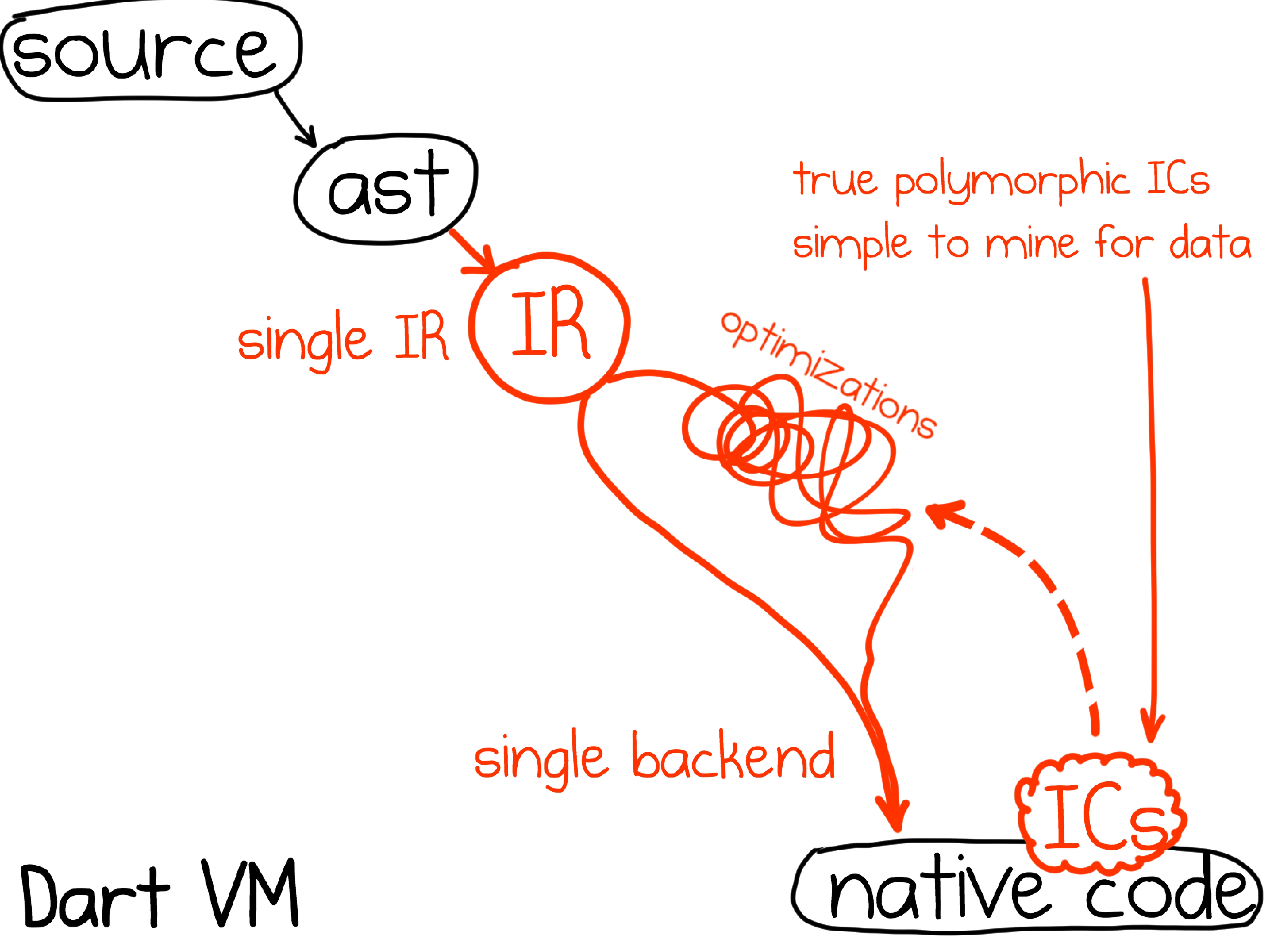

source
ast
compiler
optimizations
HIR
LIR
compiler
ICs
native code
V8


source
ast
compiler
optimizations
HIR
LIR
compiler
type feedback
ICs
native code
V8


source
ast
compiler
compiler
HIR
optimizations
LIR
type feedback
ICs
native code
V8 +Crankshaft


source
ast
compiler
optimizations
HIR
LIR
need to be in sync
compiler
type feedback
type oracle pattern matches
stub code to extract types
ICs
native code
V8+Crankshaft


source
ast
IR
optimizations
ICs
native code
Dart VM



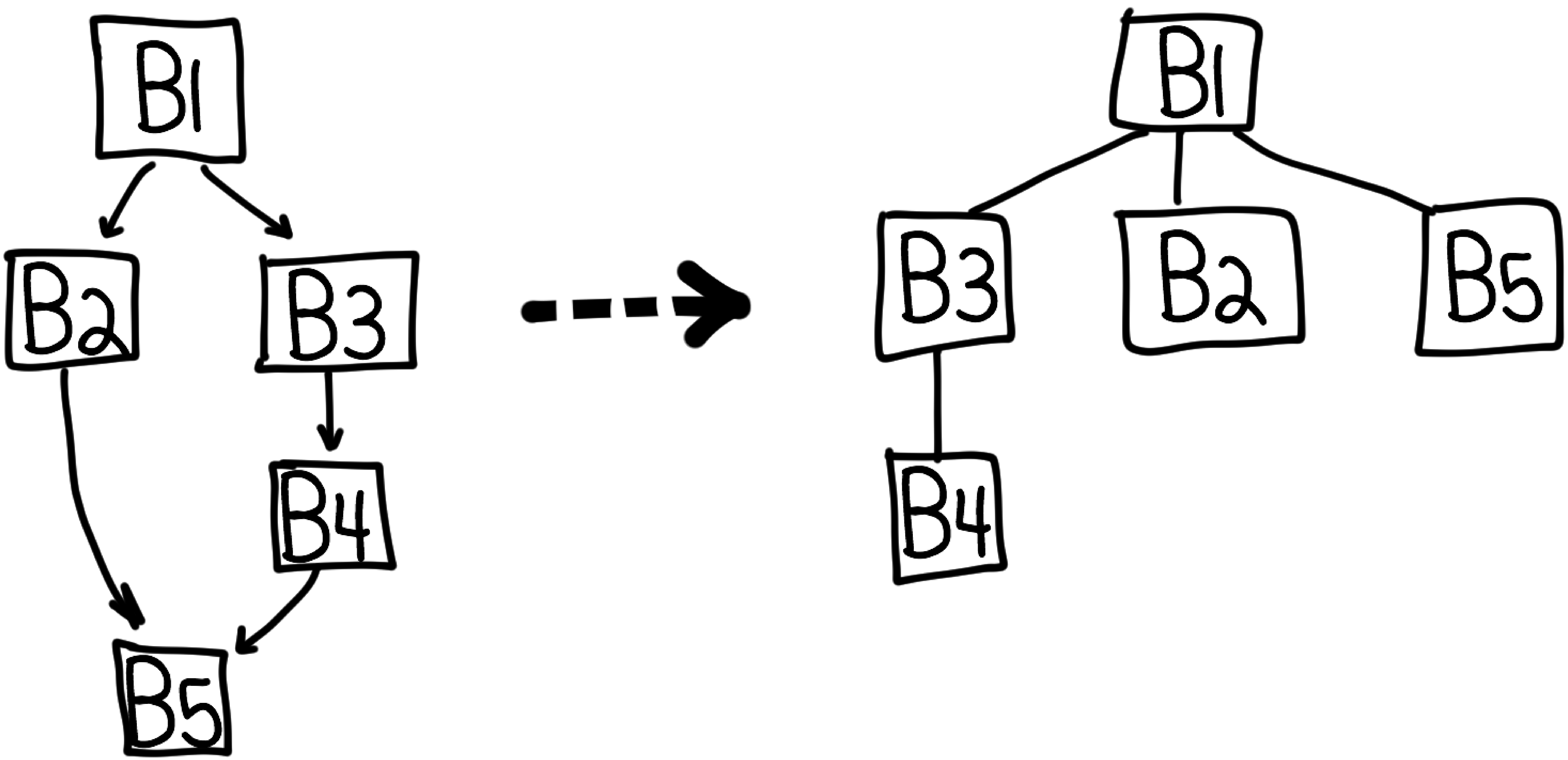
source
ast
IR
single IR
optimizations
true polymorphic ICs
simple to mine for data
single backend
ICs
native code
Dart VM

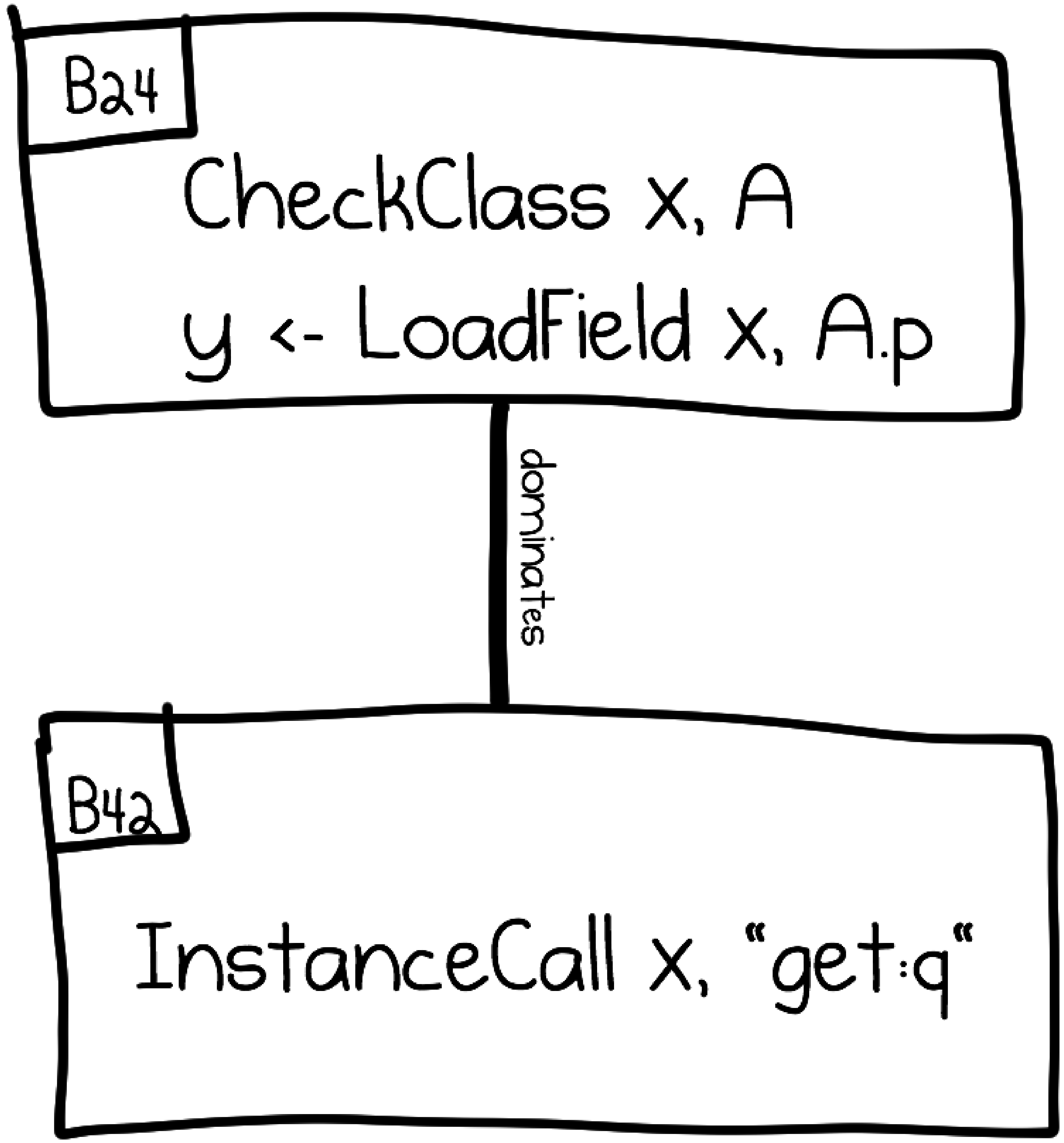
Optimizations

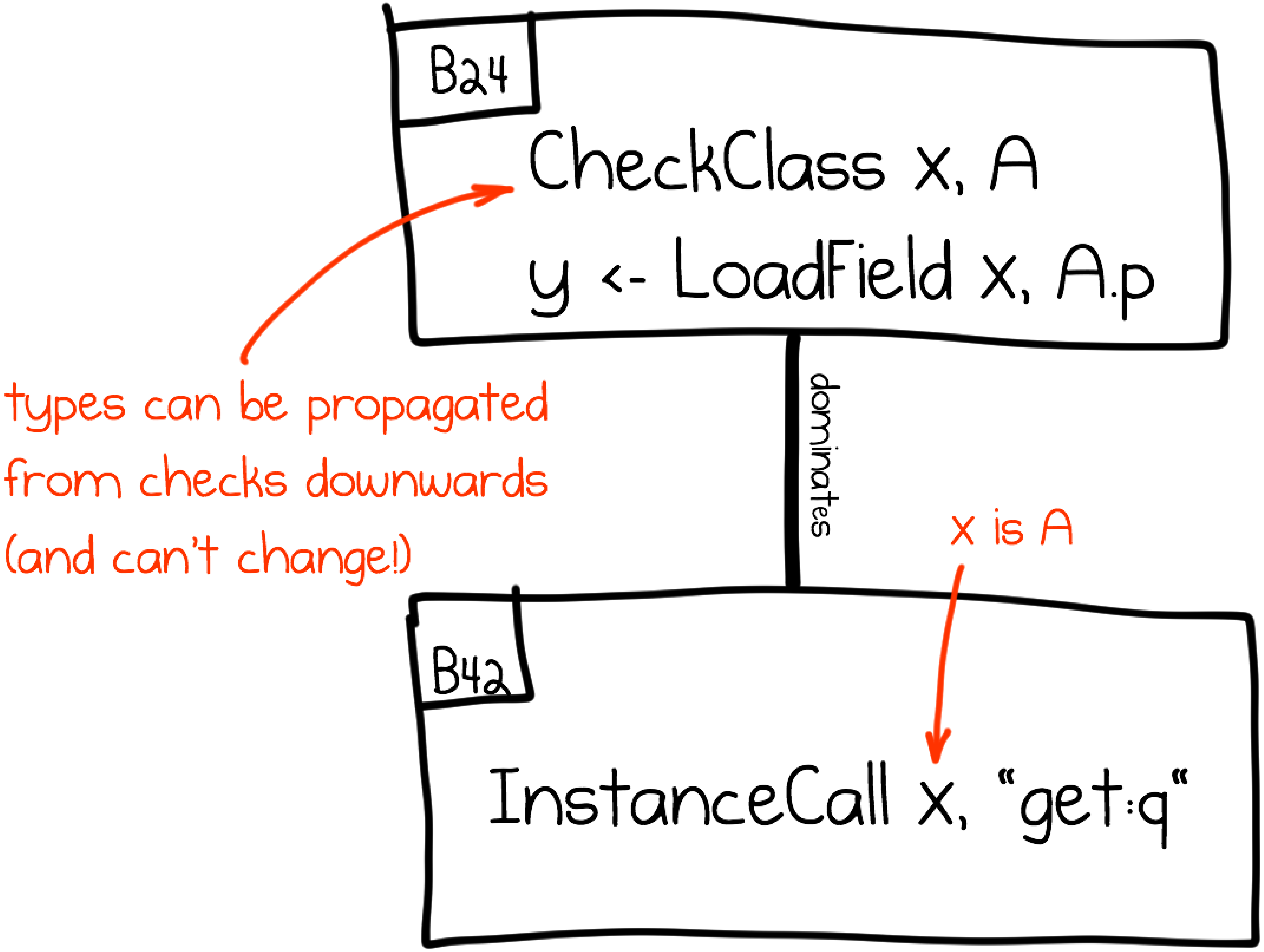
(from this point on we will mostly be talking about Dart VM)

- inlining
- constant propagation
- type inference
- range inference
- primitives unboxing
- common subexpression elimination
- loop invariant code motion
- load forwarding
- allocation sinking
- block reordering
- branch folding

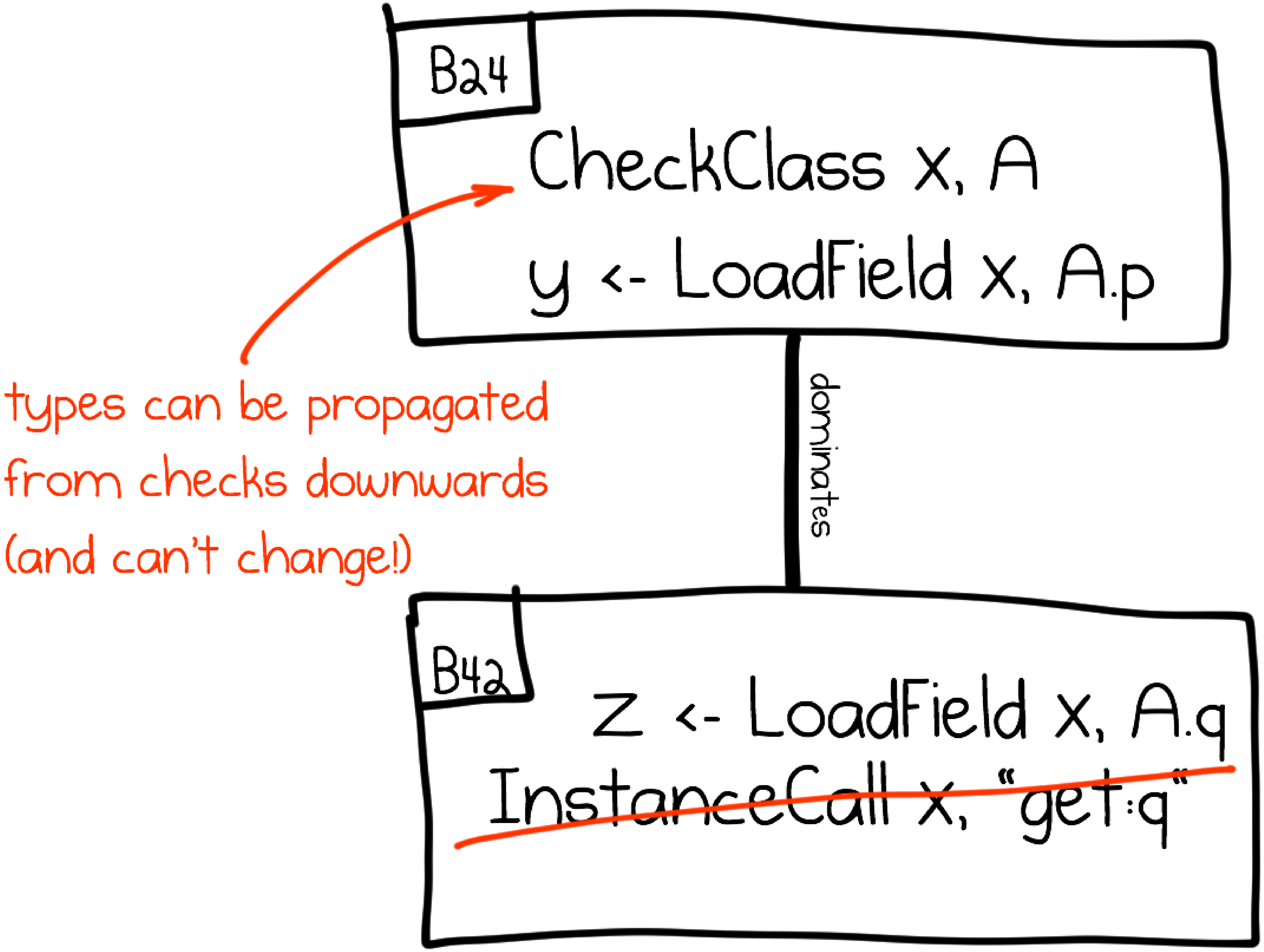
Most optimization passes are dominator tree based




B24
CheckClass x, A
y <- LoadField x, A.p
dominates
B42
InstanceCall x, "get:q"



B24
CheckClass x, A
y <- LoadField x, A.p
types can be propagated
from checks downwards
(and can't change!)
dominates
x is A
B42
InstanceCall x, "get:q"


B24
CheckClass x, A
y <- LoadField x, A.p
dominates
B42
z <- LoadField x, A.q
InstanceCall x, "get:q"
types can be propagated
from checks downwards
(and can't change!)

- remove redundant checks

- avoid (re)optimizing non-executed code if we have enough type information

- reduce polymorphism after inlining of generic functions

- constant fold is (instance-of) checks

[**checked** mode inserts **assert(v is T)**]

```
y <- LoadField x, A.f
```

compiler knows where this field is

```
y <- LoadField x, A.f
```

compiler knows where
this field is

y <- LoadField x, A.f

compiler (usually) does not know
what the field contains

[because Dart type annotations are just comments in production mode] y <- LoadField x, A.f {C}

globally track possible type of each field and assume type when loading value

GuardField A.f {C}, z

StoreField x, A.f, z

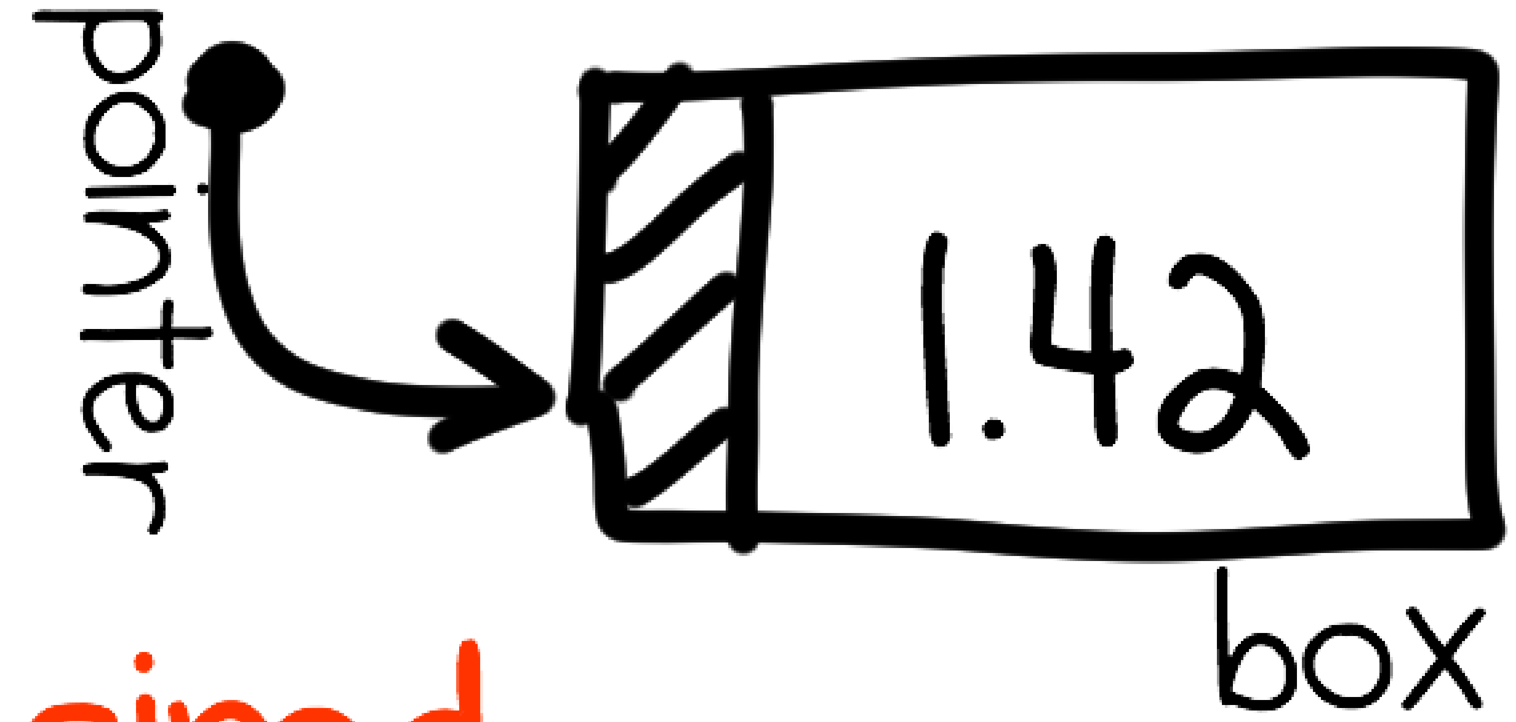
y <- LoadField x, A.f {C}

guard assumed type of field on each store [deoptimize code depending on invalidated assumptions]

GuardField A.f {C}, z

StoreField x, A.f, z

primitives unboxing



simple for double and simd:

- just look at the type

not so simple for int:

- requires range profiling for op's results

=> currently VM does not unbox int

primitives unboxing

works well enough because

- **double** and **int** different types
- most interesting ints fit into tagged smi encoding

+ compiler has some support for unboxed 64bit ints

primitives unboxing

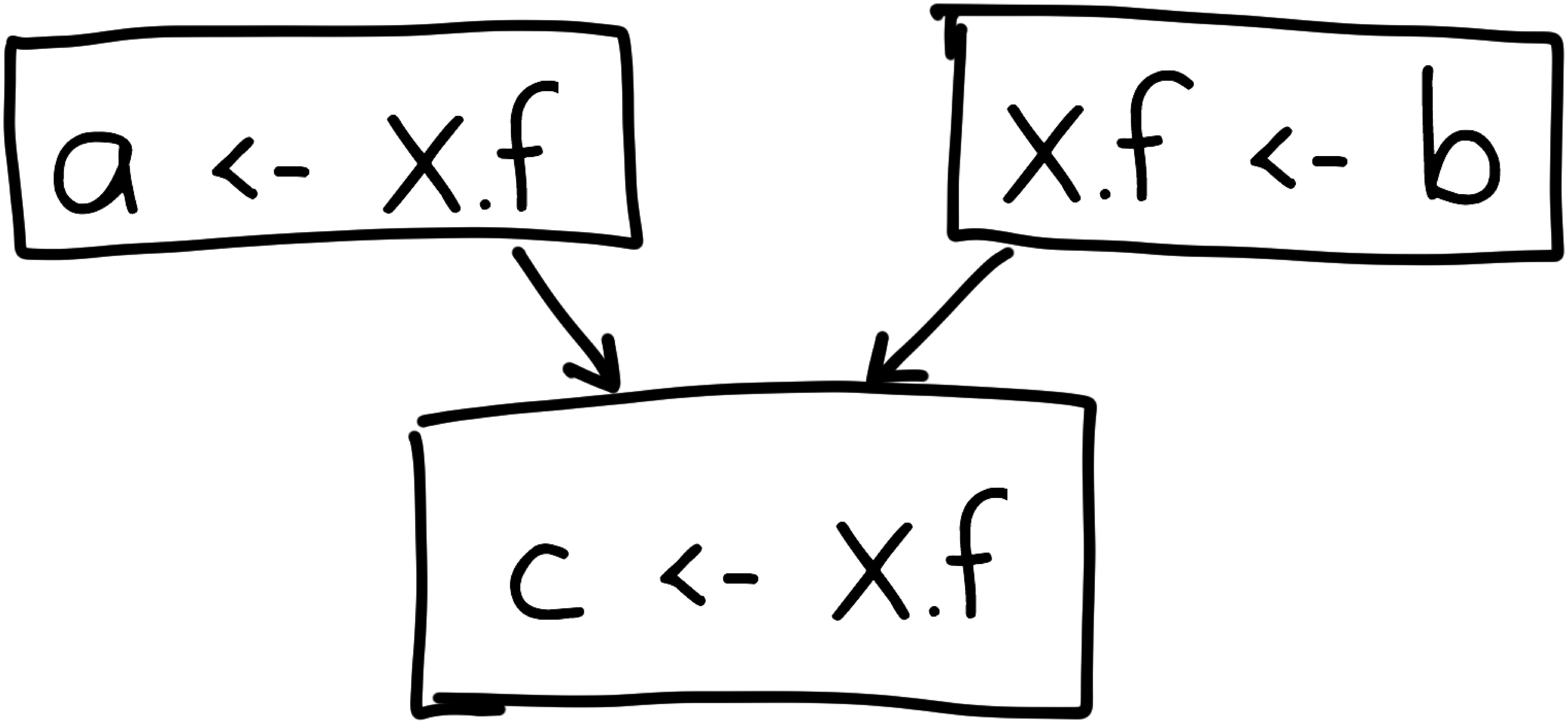
compare to V8:

JavaScript has only double

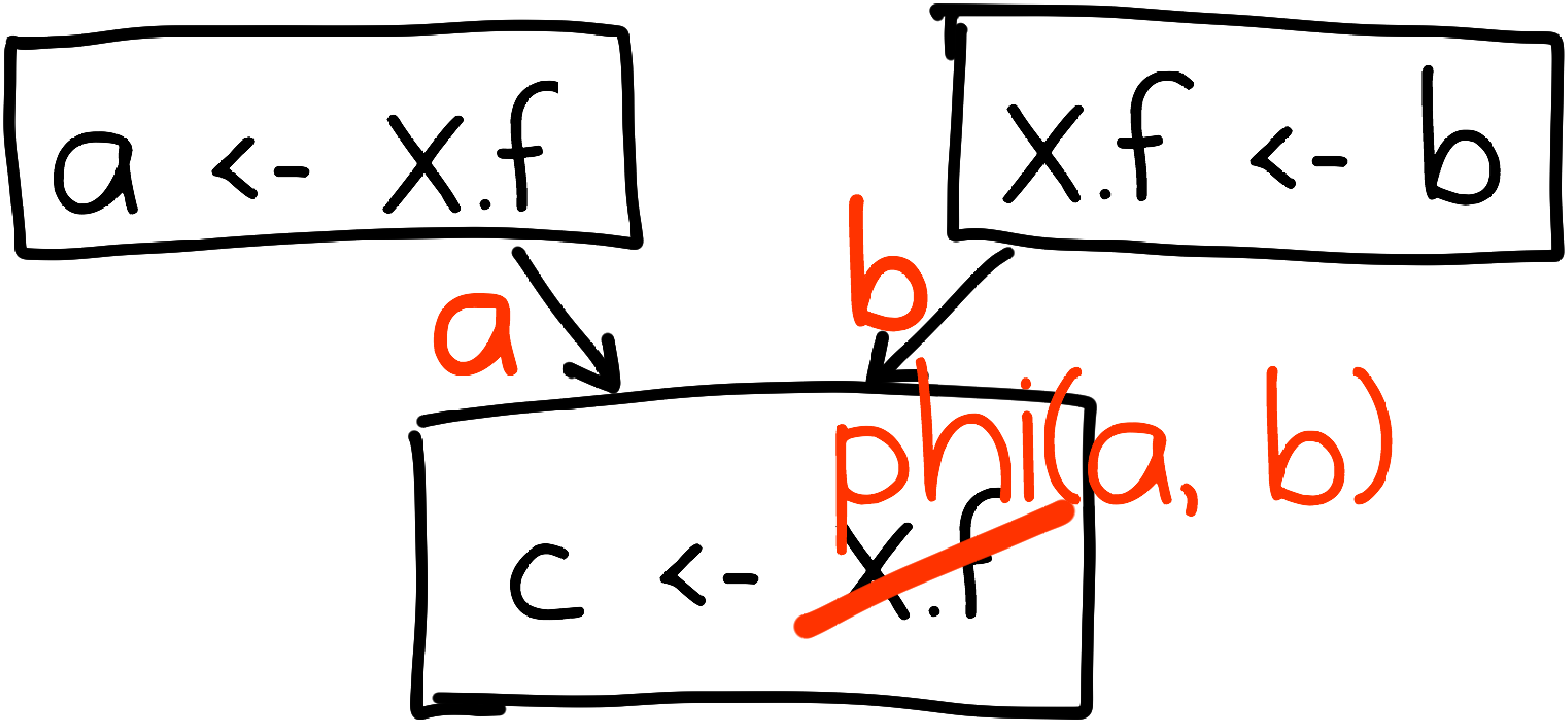
... but bitwise ops coerce into int32, uint32 range

- arithmetic ops collect range feedback: smi, int32, double
- compiler tries to guess best representation

load forwarding



load forwarding



allocates temporary iterator

```
for (var item in list) {
  // use item
}
```

```
var it = new Iterator(list);
while (it.moveNext()) {
  var item = it.current;
}
```

```
var it = alloc(Iterator);
it.list = list;
it.idx = -1;
while (++it.idx < it.list.length) {
  var item = it.list[it.idx];
}
```

```
var it = alloc(Iterator);
it.list = list;
it.idx = *idx = -1;
while ((it.idx = ++*idx) <
                list.length) {
  var item = list[*idx];
}
```

```
★idx = -1;
while (++★idx < list.length) {
  var item = list[★idx];
}
```

last step was allocation sinking

```
*idx = -1;
while (++*idx < list.length) {
  var item = list[*idx];
}
```

... allocation was sunk into deopt side exits

```
★idx = -1;

while (++★idx < list.length) {
  var item = list[★idx];
}
```

but I simplified things a lot, in reality
many optimizations have to work together

```
bool moveNext() {
  int length = _iterable.length;
  if (_length != length) {
    throw new ConcurrentModificationError(_iterable);
  }
  if (_index >= length) {
    _current = null;
    return false;
  }
  _current = _iterable.elementAt(_index);
  _index++;
  return true;
}
```

if possible check will be folded away

similar example

```
list.forEach((item) {
  // use item
});
```

load forwarding + allocation sinking are crucial to reduce the cost of abstractions

the trap of inlining

almost impossible to predict whether it is beneficial to inline until you try

the trap of inlining

almost impossible to predict whether it is beneficial to inline until you try

trying costs

the trap of inlining

almost impossible to predict whether it is beneficial to inline until you try

thus have to be conservative

the trap of inlining

on the other hand inlining
exposes redundancy
that could be eliminated

the trap of inlining

"solution": force inlining of important methods in core library

[does not help user code, if normal inlining heuristics do not "hit" it]

The End