Rebuilding Optimizing Compiler for Dart

{ @mraleph | vegorov@google.com }

6 YEARS OF DART

(VM engineer perspective)

{ @mraleph | vegorov@google.com }

Excelsior JET

V8

Dart VM

LuaJIT

Dart VM (2012 - ...)

Dart 2011

«Dart targets a wide range of development scenarios: from a one-person project without much structure to a large-scale project needing formal types in the code to state programmer intent. *To support this wide range of projects, Dart has optional types; this means you can start coding without types and add them later as needed.* We believe Dart will be great for writing large web applications.»

— Dart: A language for structured web programming

```
Dog d = new Cat();
d.woof();
```

```
Dog d = new Cat();
d.woof();

class Cat {
  noSuchMethod(invocation) {
    if (invocation.memberName == #woof) {
      print('I am a 🐱 not a 🐶!');
    }
  }
}
```

dynamic language


JIT

initially
AST based

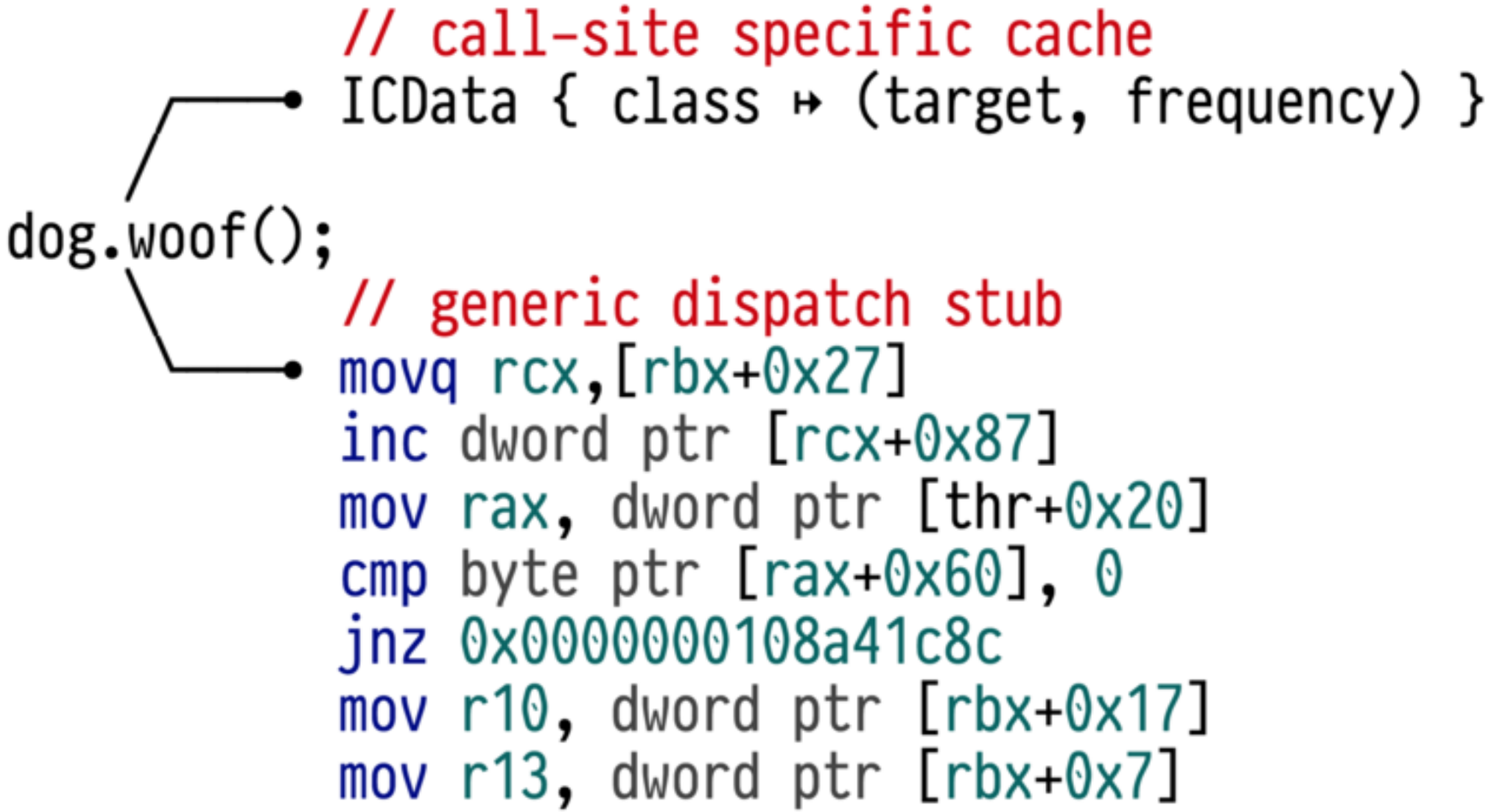
2012

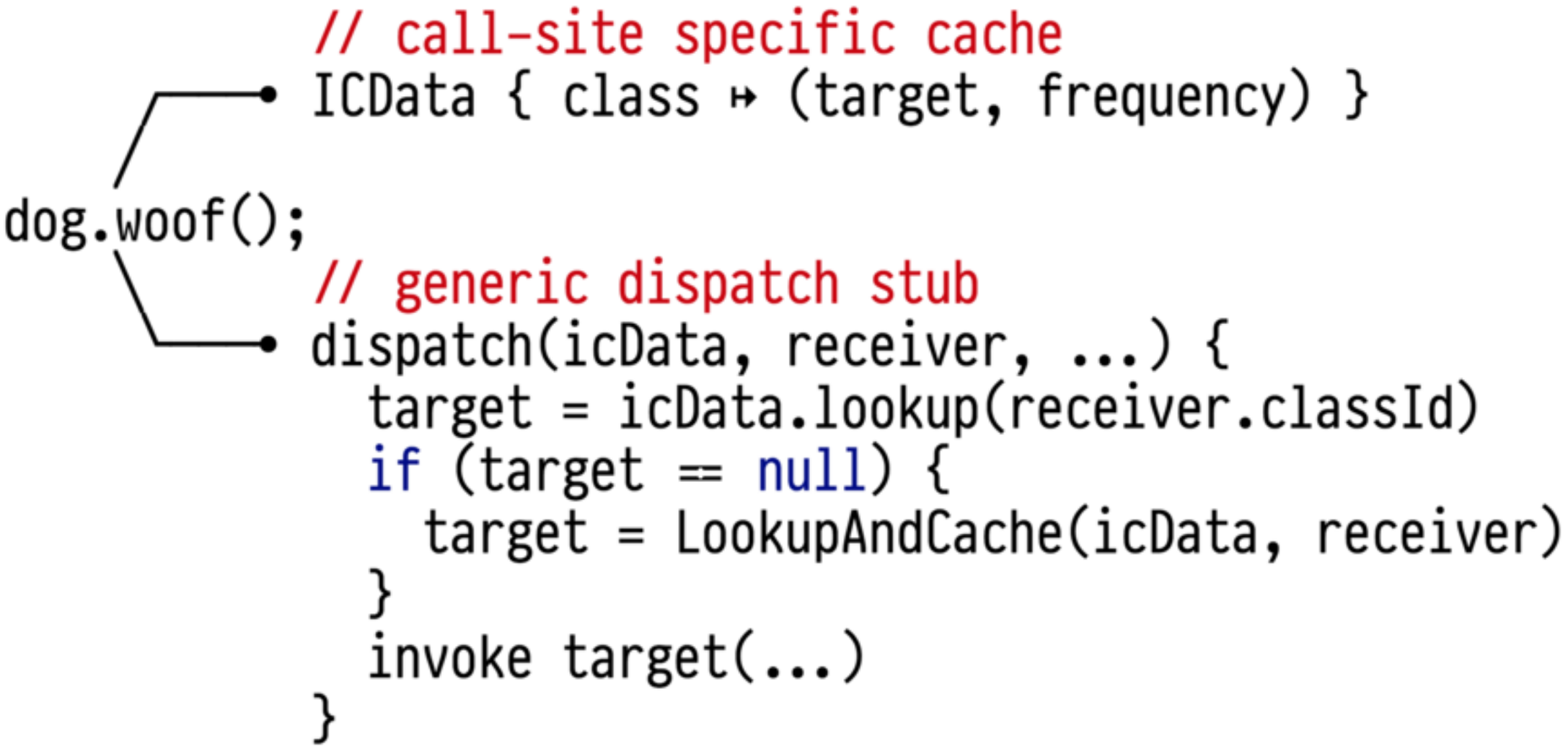
started new JIT compiler

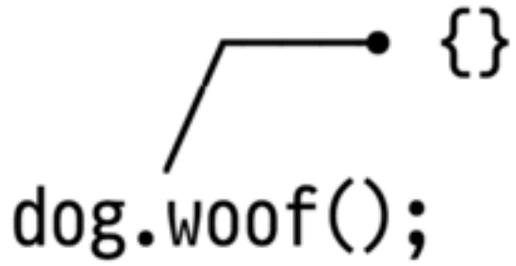
inline caching (IC)

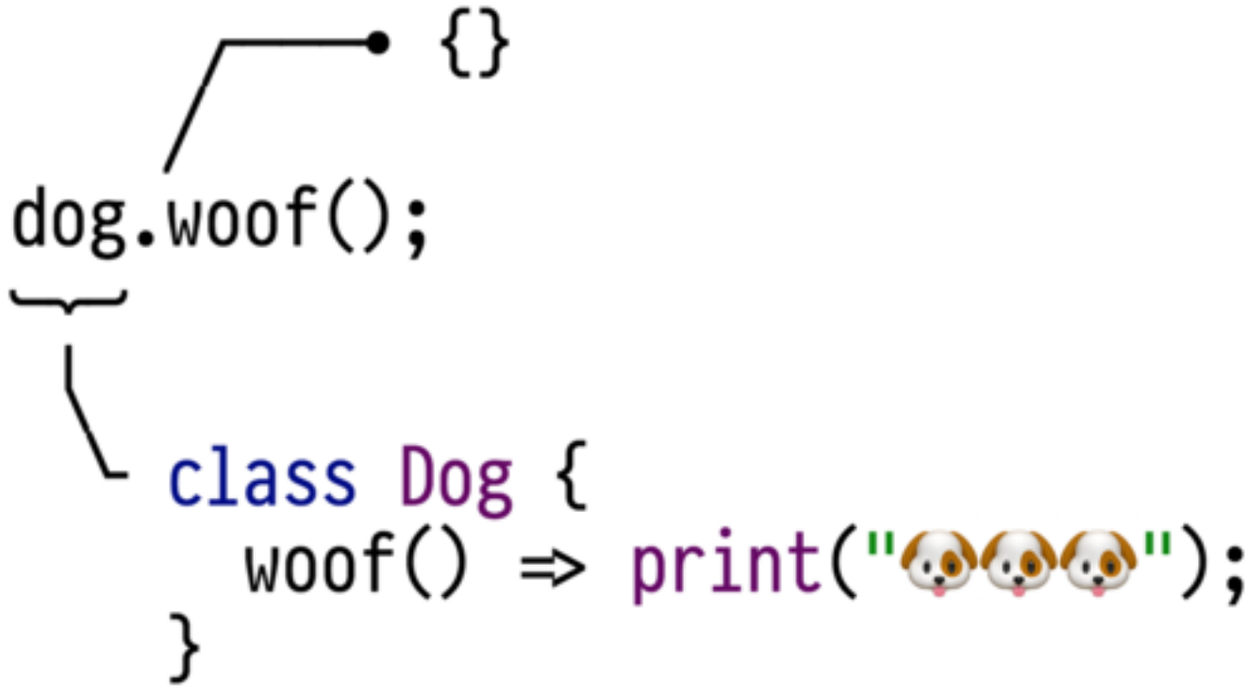
+

speculative opts


// call-site specific cache
ICData { class ↦ (target, frequency) }
dog.woof();
// generic dispatch stub
movq rcx,[rbx+0x27]
inc dword ptr [rcx+0x87]
mov rax, dword ptr [thr+0x20]
cmp byte ptr [rax+0x60], 0
jnz 0x0000000108a41c8c
mov r10, dword ptr [rbx+0x17]
mov r13, dword ptr [rbx+0x7]

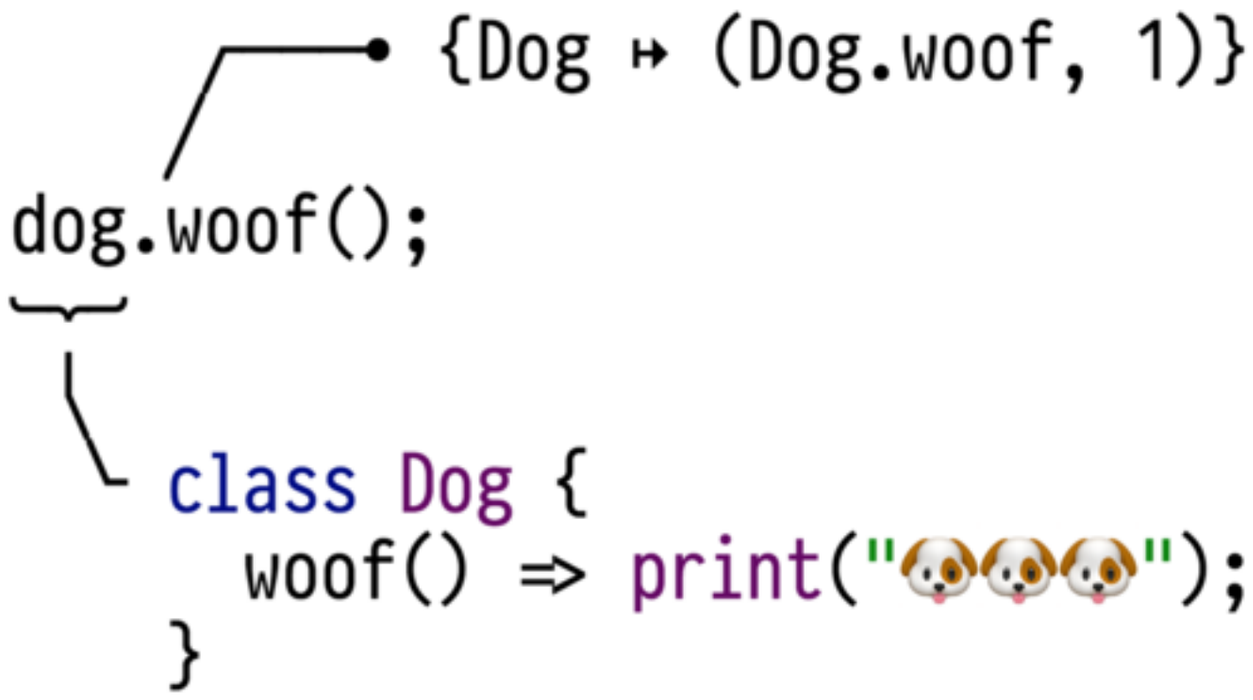

// call-site specific cache
ICData { class ↦ (target, frequency) }
dog.woof();
// generic dispatch stub
dispatch(icData, receiver, ...) {
target = icData.lookup(receiver.classId)
if (target == null) {
target = LookupAndCache(icData, receiver)
}
invoke target(...)
}

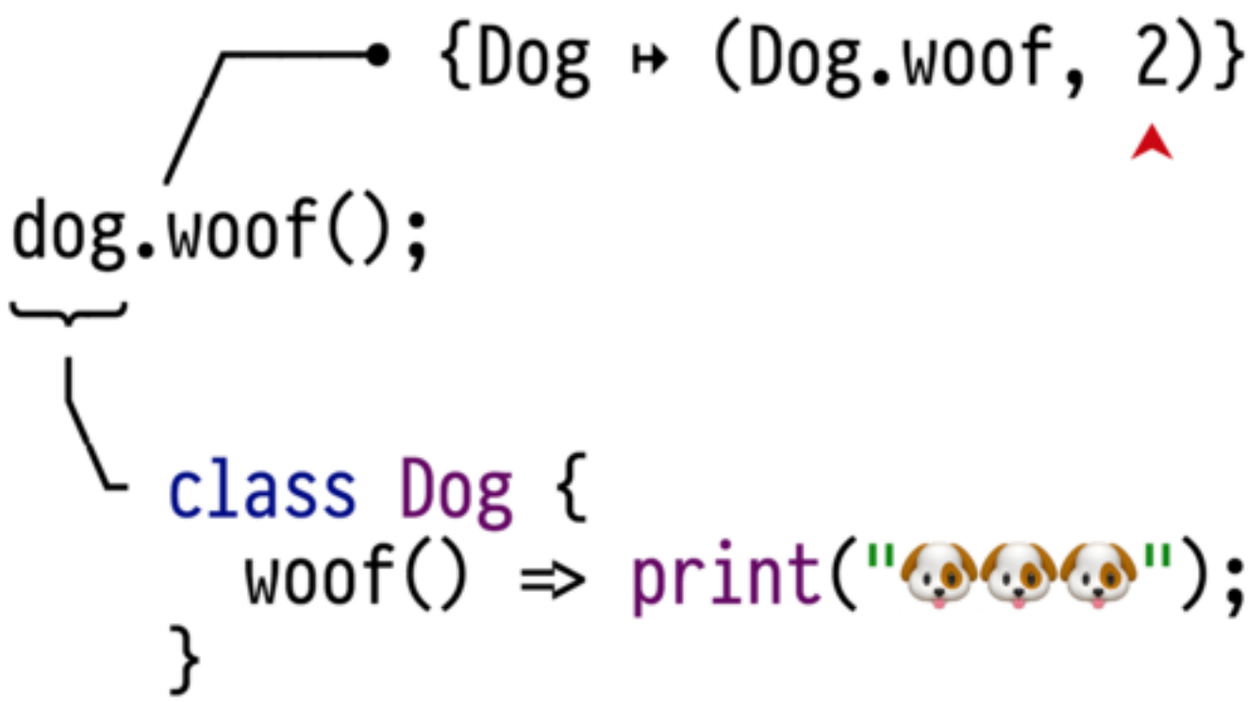

{}
dog.woof();

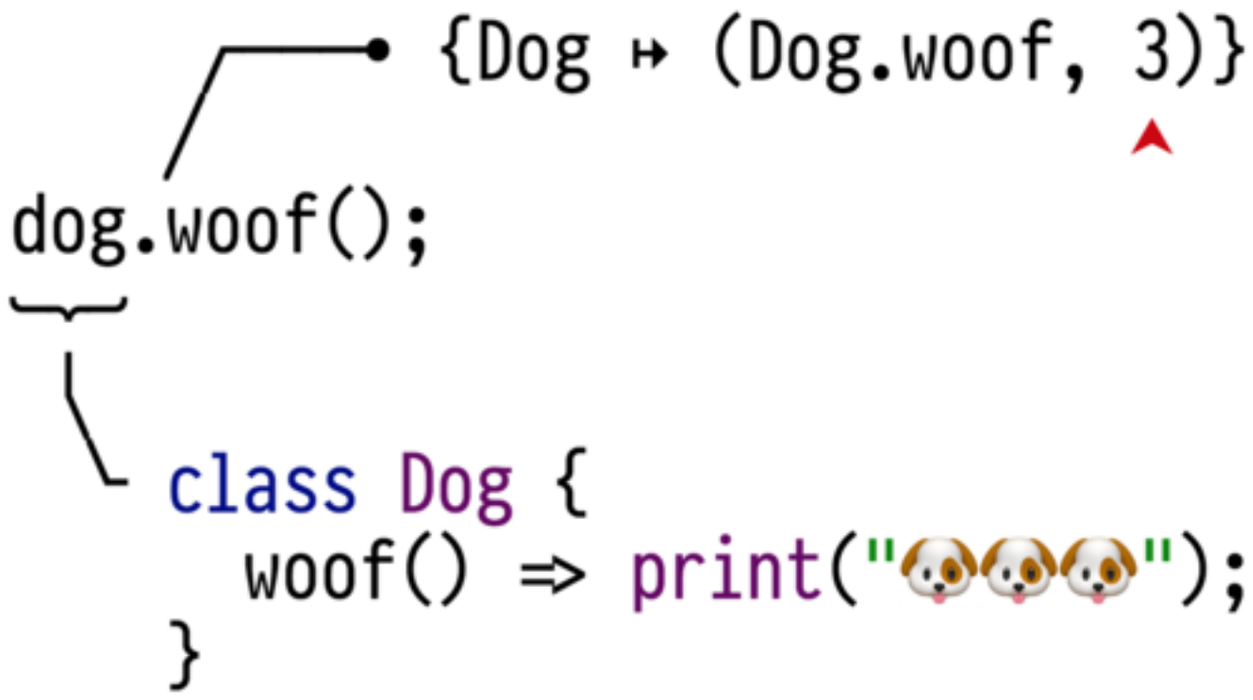

{}
dog.woof();
class Dog {
woof() => print("🐶🐶🐶");
}

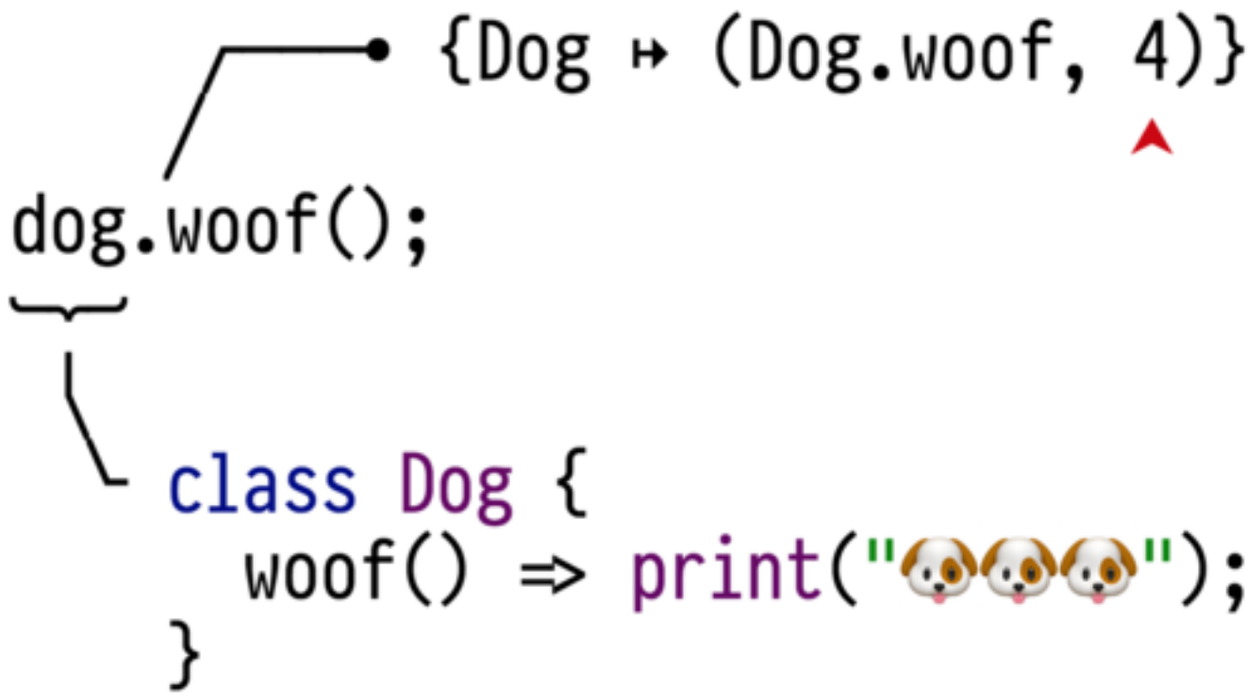

{}
↳LookupAndCache()
dog.woof();
class Dog {
woof() => print("🐶🐶🐶");
}

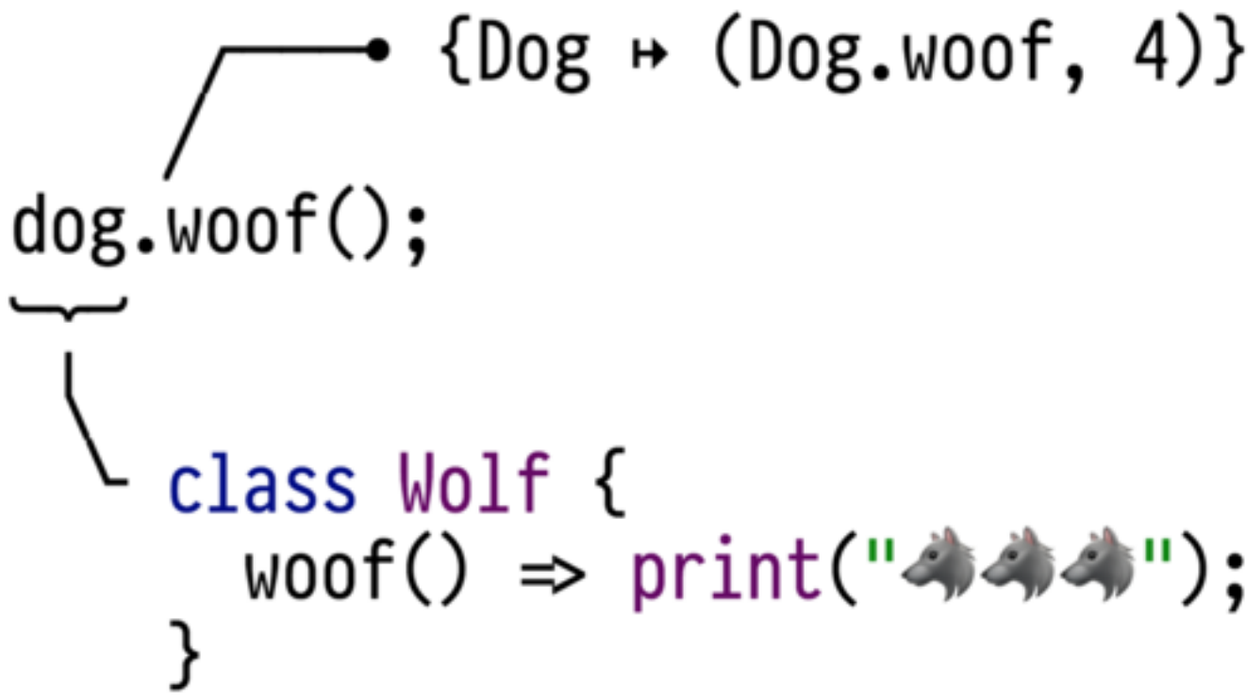

{}
↳LookupAndCache()
dog.woof();
class Dog {
woof() => print("🐶🐶🐶");
}

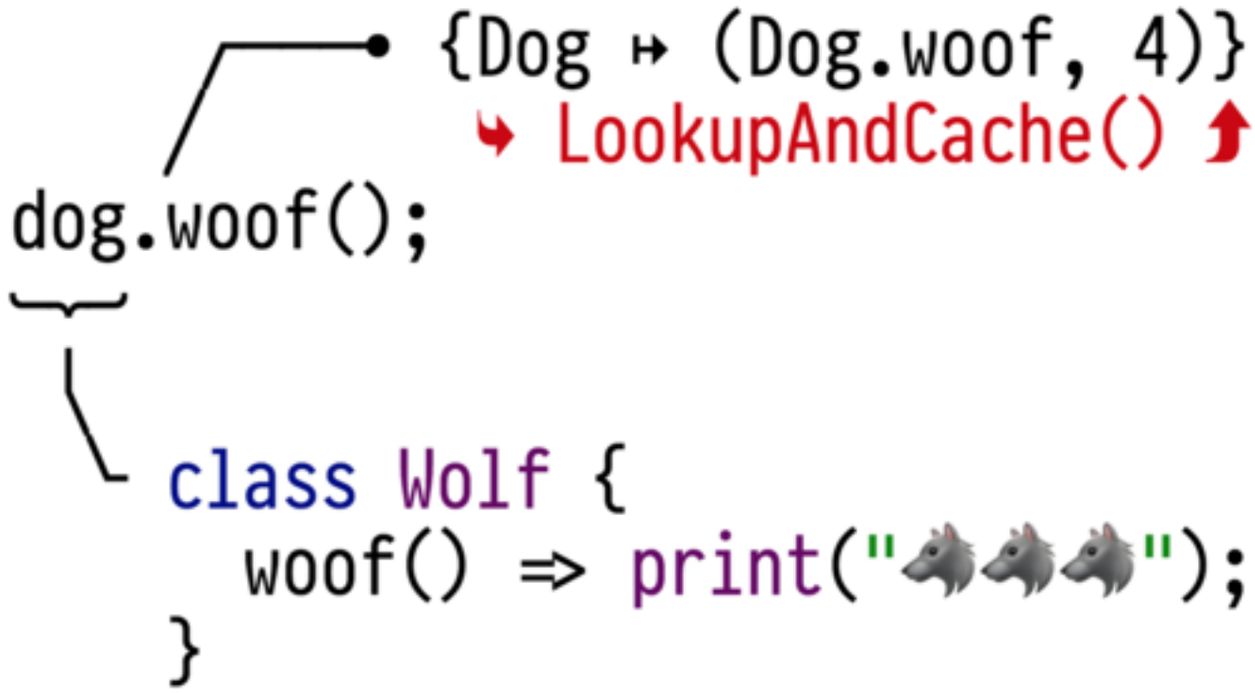

{Dog ↦ (Dog.woof, 1)}
dog.woof();
class Dog {
woof() ⇒ print("🐶🐶🐶");
}

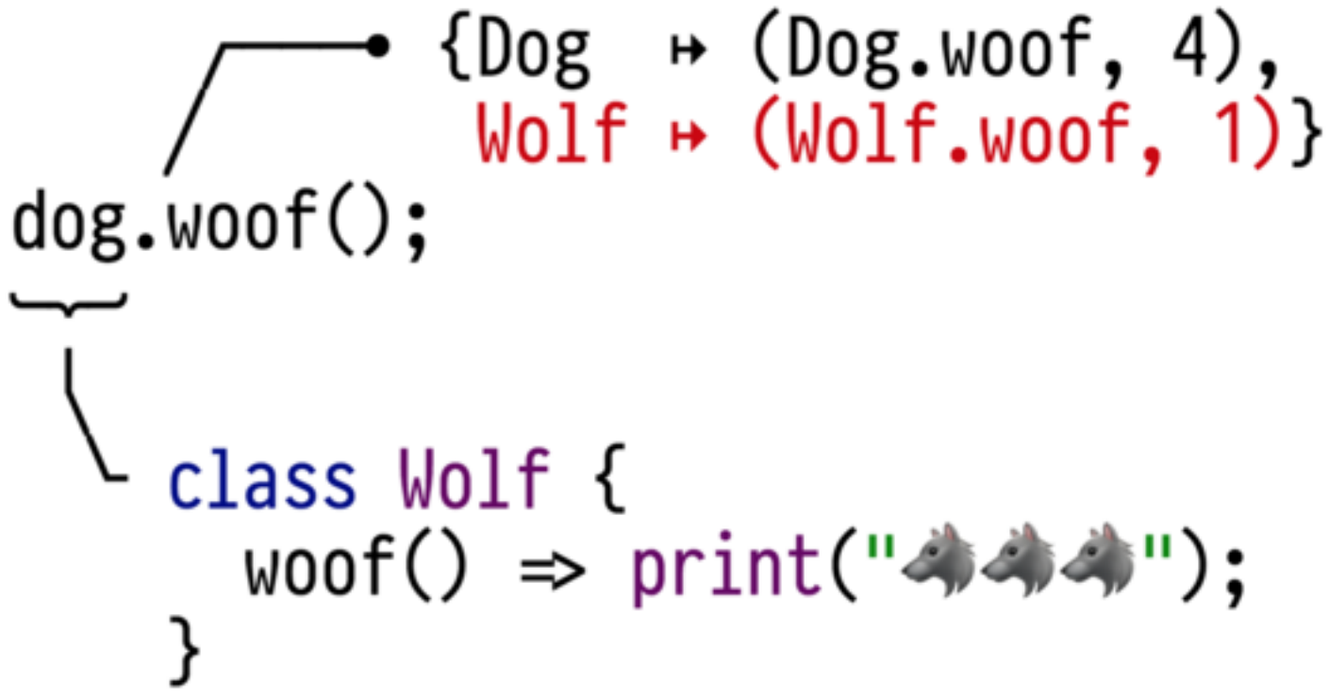

{Dog ↦ (Dog.woof, 2)}
dog.woof();
class Dog {
woof() ⇒ print("🐶🐶🐶");
}


{Dog ↦ (Dog.woof, 3)}
dog.woof();
class Dog {
woof() ⇒ print("🐶🐶🐶");
}

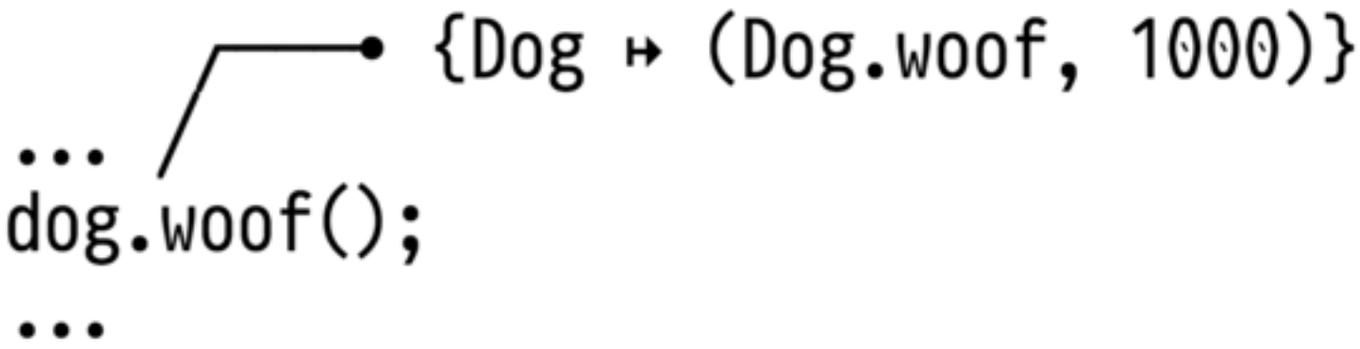

{Dog ↦ (Dog.woof, 4)}
dog.woof();
class Dog {
woof() ⇒ print("🐶🐶🐶");
}

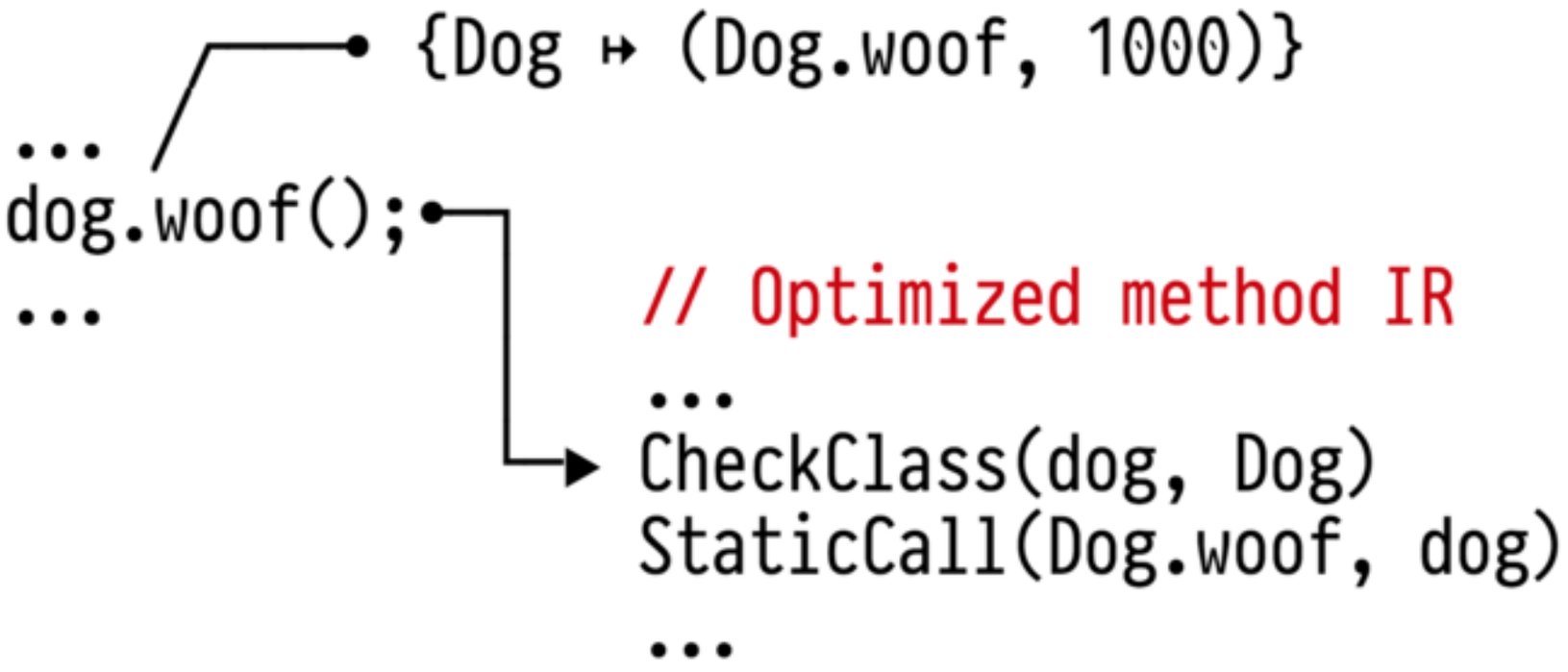

{Dog ↦ (Dog.woof, 4)}
dog.woof();
class Wolf {
woof() => print("🐺🐺🐺");
}

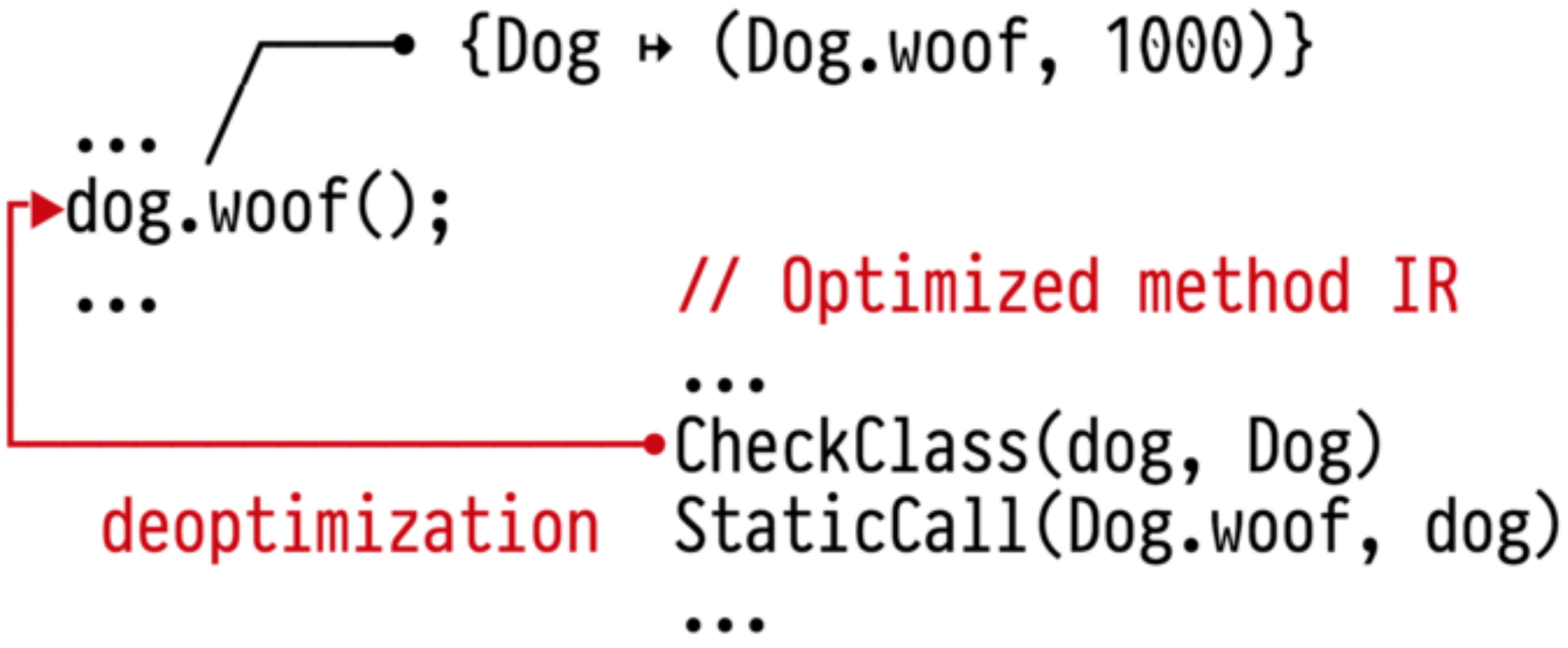

{Dog ↦ (Dog.woof, 4)}
↳ LookupAndCache() ⬆
dog.woof();
class Wolf {
woof() ⇒ print("🐺🐺🐺");
}


{Dog ↦ (Dog.woof, 4),
Wolf ↦ (Wolf.woof, 1)}
dog.woof();
class Wolf {
woof() ⇒ print("🐺🐺🐺");
}

type profiling


{Dog ↦ (Dog.woof, 1000)}
...
dog.woof();
...


{Dog ↦ (Dog.woof, 1000)}
...
dog.woof();
...
// Optimized method IR
...
CheckClass(dog, Dog)
StaticCall(Dog.woof, dog)
...


{Dog ↦ (Dog.woof, 1000)}
...
dog.woof();
...
// Optimized method IR
...
CheckClass(dog, Dog)
deoptimization
StaticCall(Dog.woof, dog)
...

classics

Published in ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991.

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle
Craig Chambers
David Ungar[†]

Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{urs,craig,ungar}@self.stanford.edu

Abstract: *Polymorphic inline caches* (PICs) provide a new way to reduce the overhead of polymorphic message sends by extending inline caches to include more than one cached lookup result per call site. For a set of typical object-oriented SELF programs, PICs achieve a median speedup of 11%.

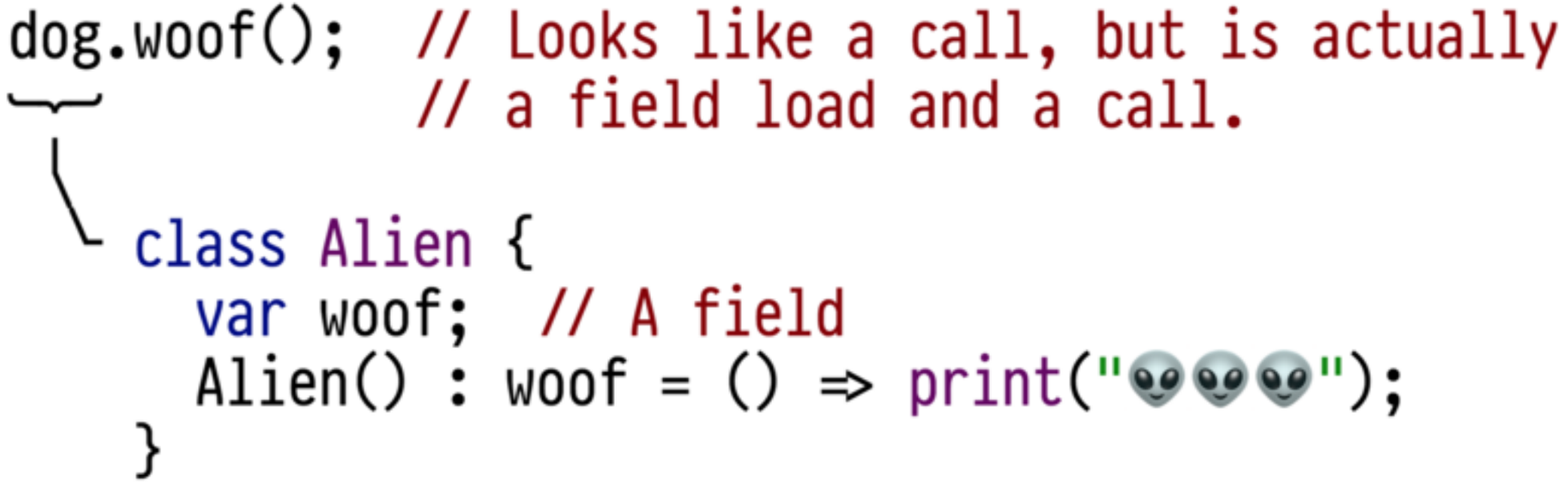
As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when *recompiling* a method. An experimental version of such a system achieves a median speedup of 27% for our set of SELF programs, reducing the number of non-inlined message sends by a factor of two.

Implementations of dynamically-typed object-oriented languages have been limited by the paucity of type information available to the compiler. The abundance of the type information provided by PICs suggests a new compilation approach for these languages, *adaptive compilation*. Such compilers may succeed in generating very efficient code for the time-critical parts of a program without incurring distracting compilation pauses.

1. Introduction

Historically, dynamically-typed object-oriented languages have run much slower than statically-typed languages. This disparity in performance stemmed largely from the relatively slow speed and high

suprisingly general

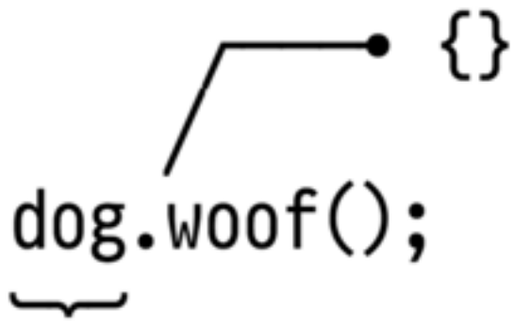

dog.woof(); // Looks like a call, but is actually
// a field load and a call.
class Alien {
var woof; // A field
Alien() : woof = () => print("👽👽👽");
}

```
//       (...) => dog.woof(...)
var f = dog.woof;
```

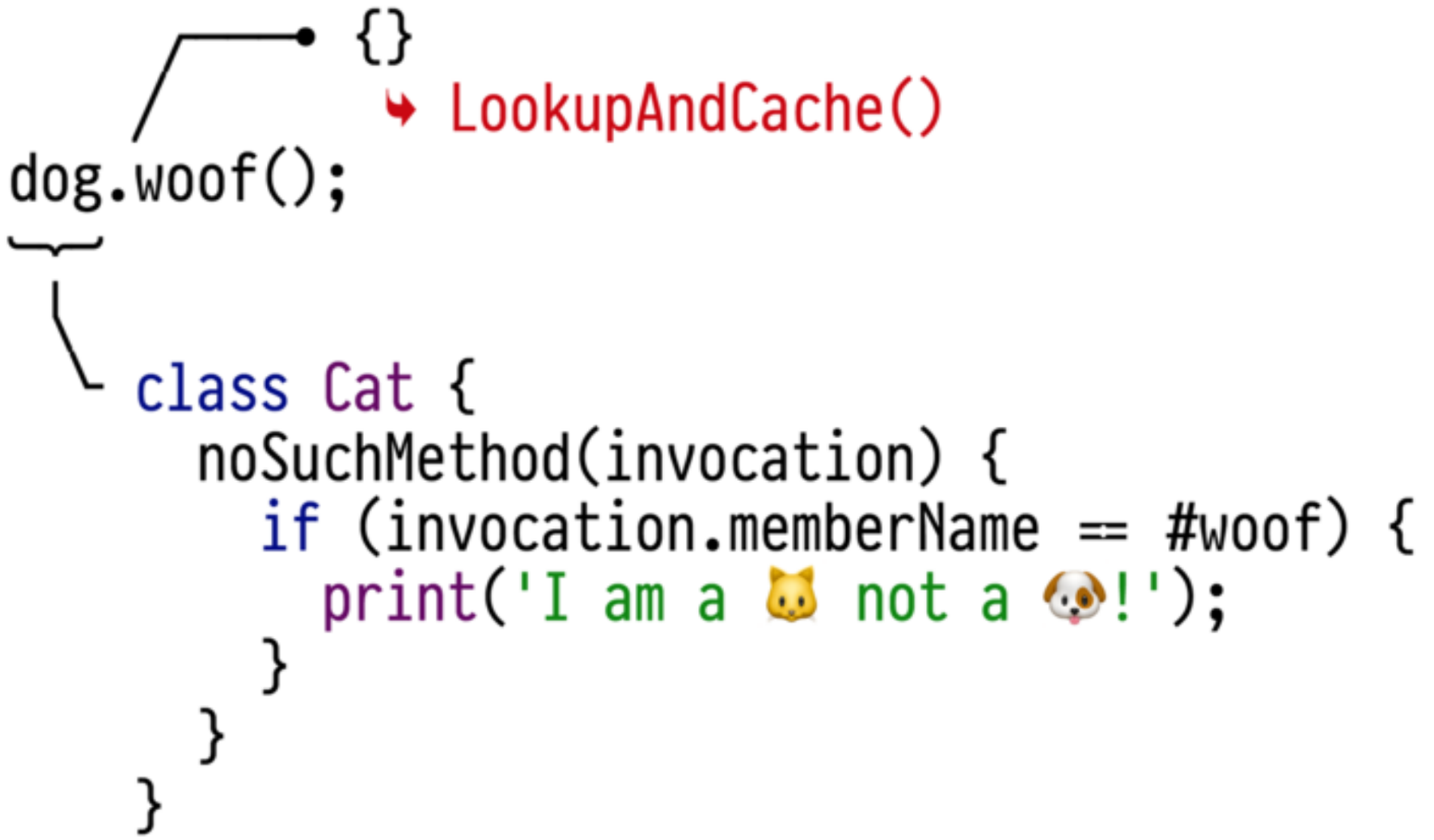
method tear-off

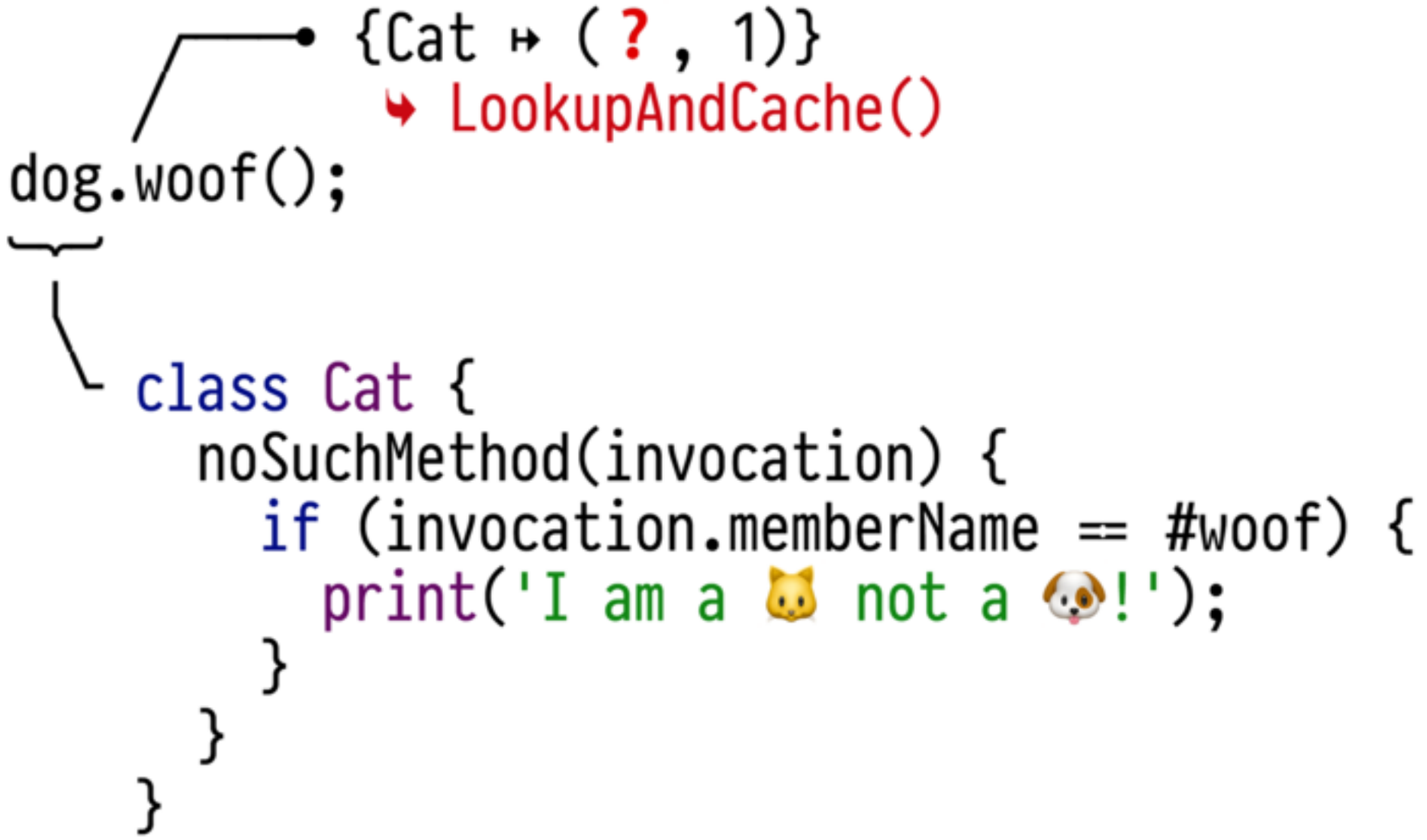
```
dog.woof();
```

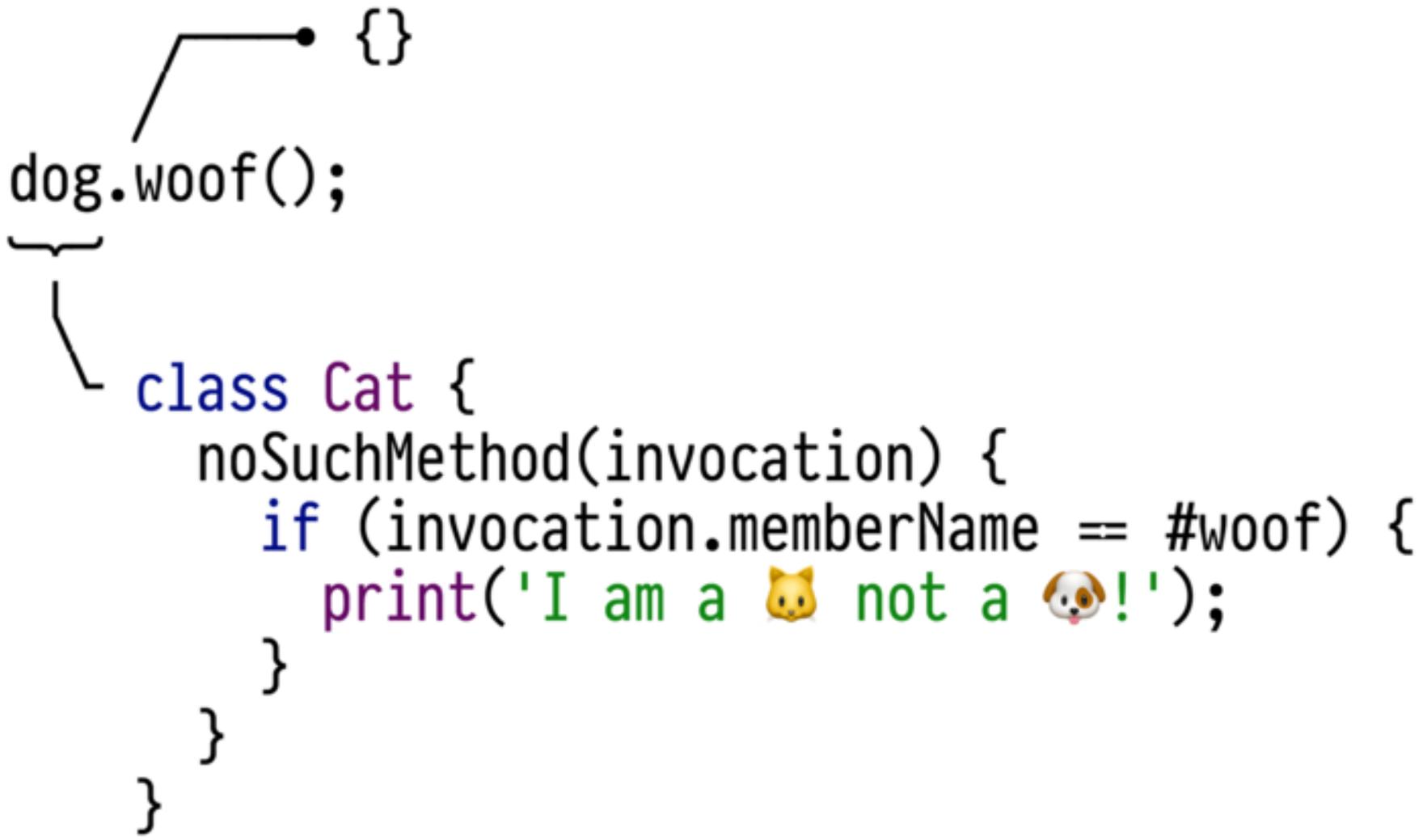
```
class Cat {
  noSuchMethod(invocation) {
    if (invocation.memberName == #woof) {
      print('I am a 🐱 not a 🐶!');
    }
  }
}
```

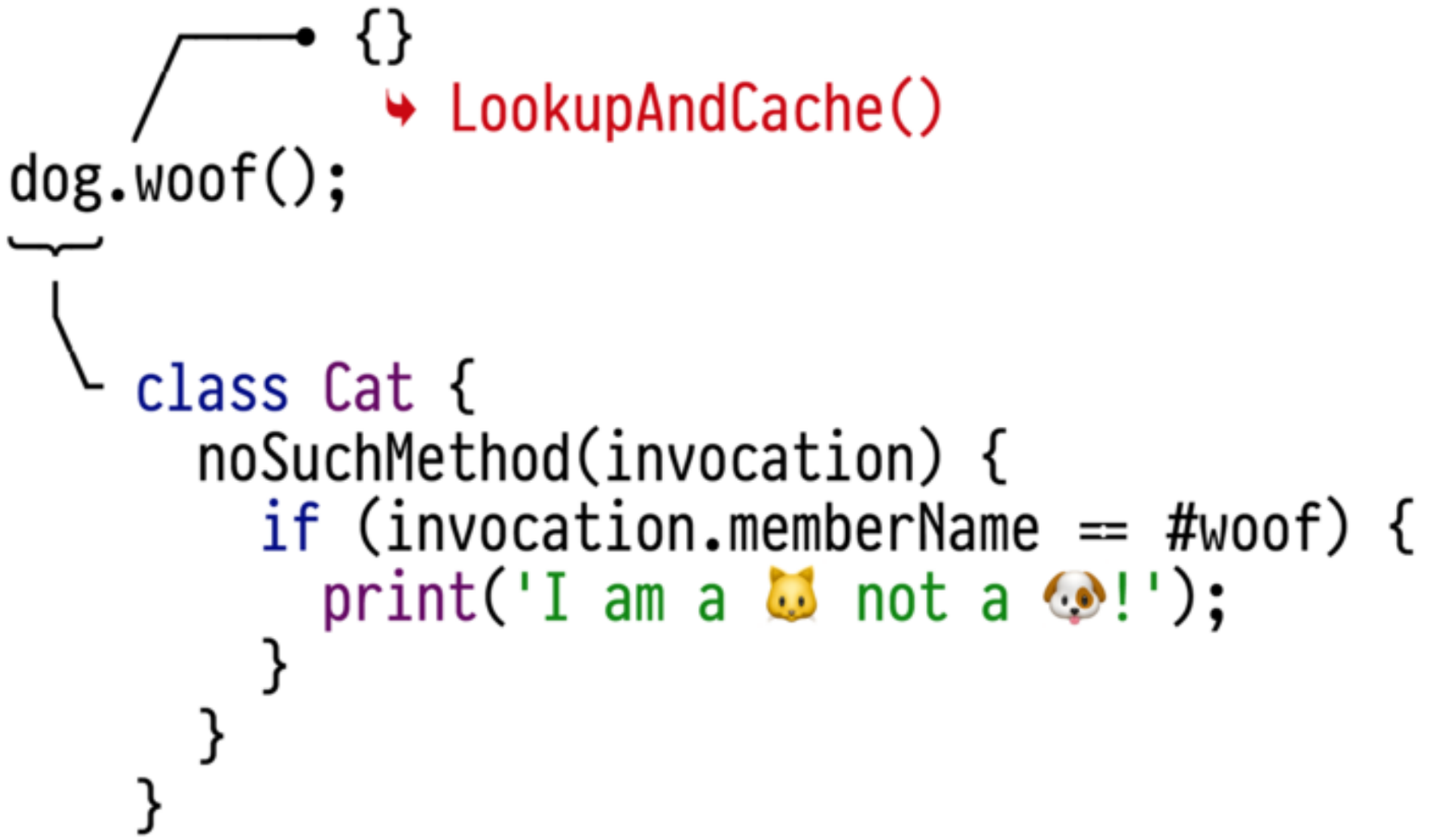


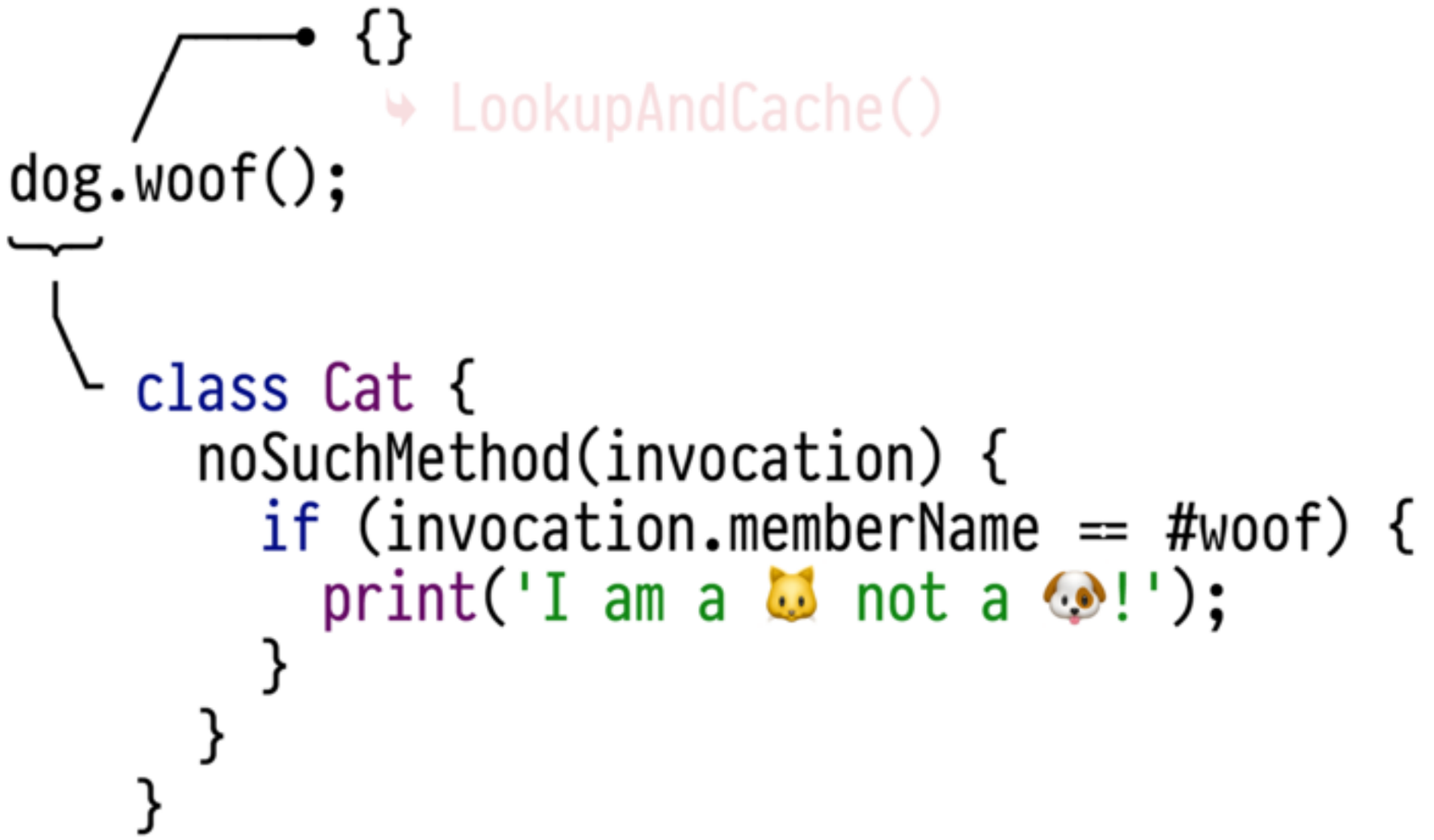

```
class Cat {
  noSuchMethod(invocation) {
    if (invocation.memberName == #woof) {
      print('I am a 🐱 not a 🐶!');
    }
  }
}
```

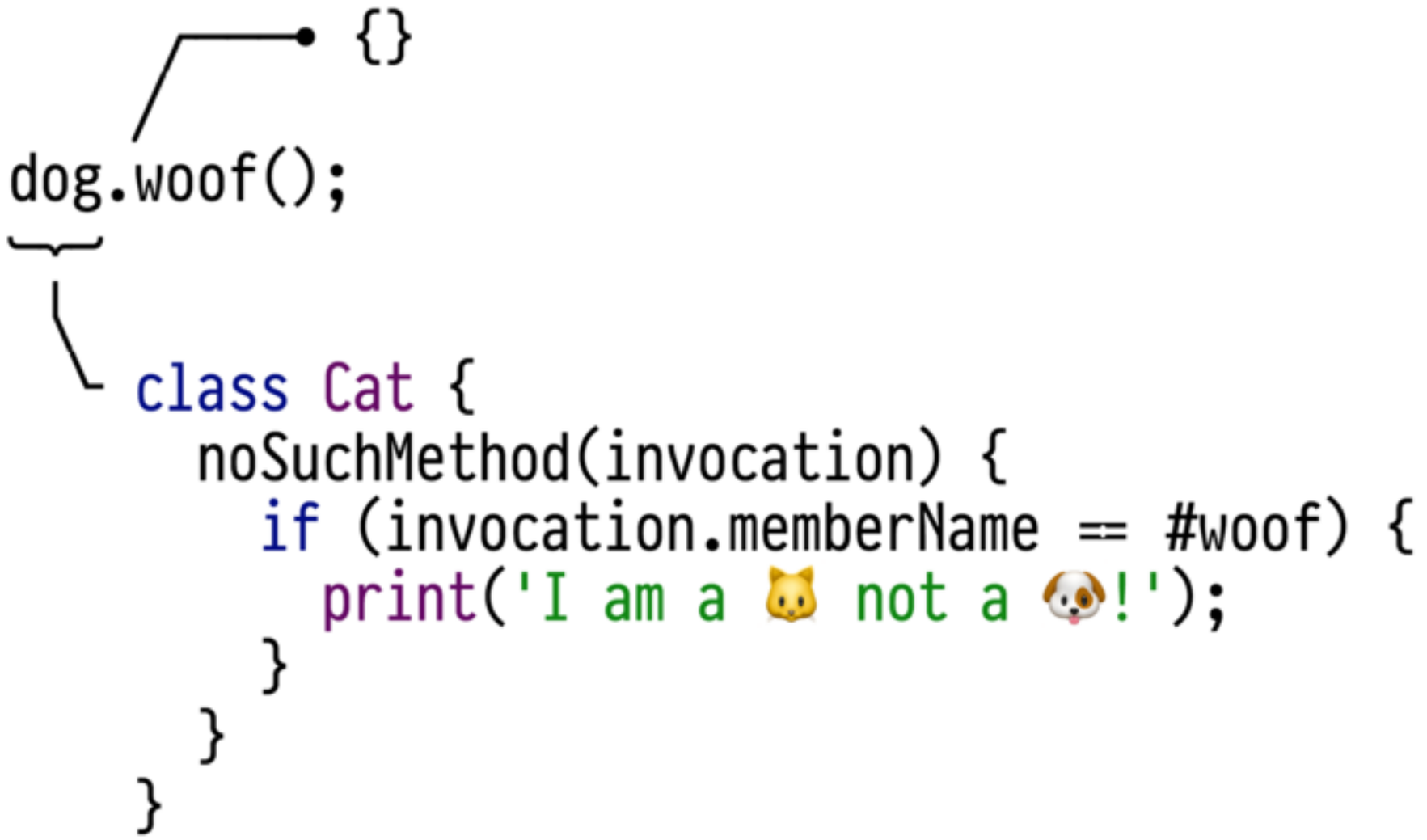

{}
LookupAndCache()
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}

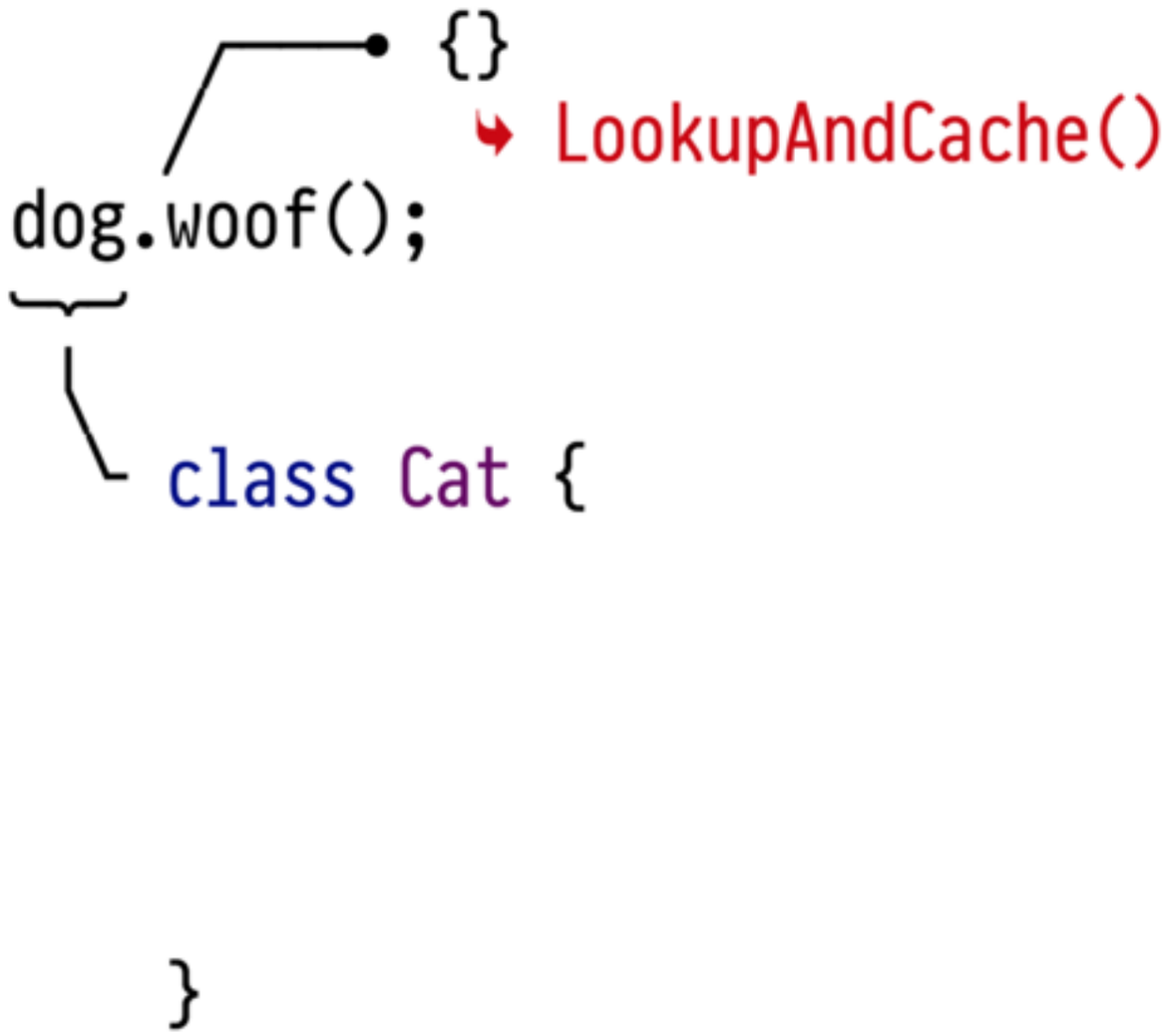

{Cat ↦ (?, 1)}
↳ LookupAndCache()
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}


{}
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}


{}
LookupAndCache()
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}

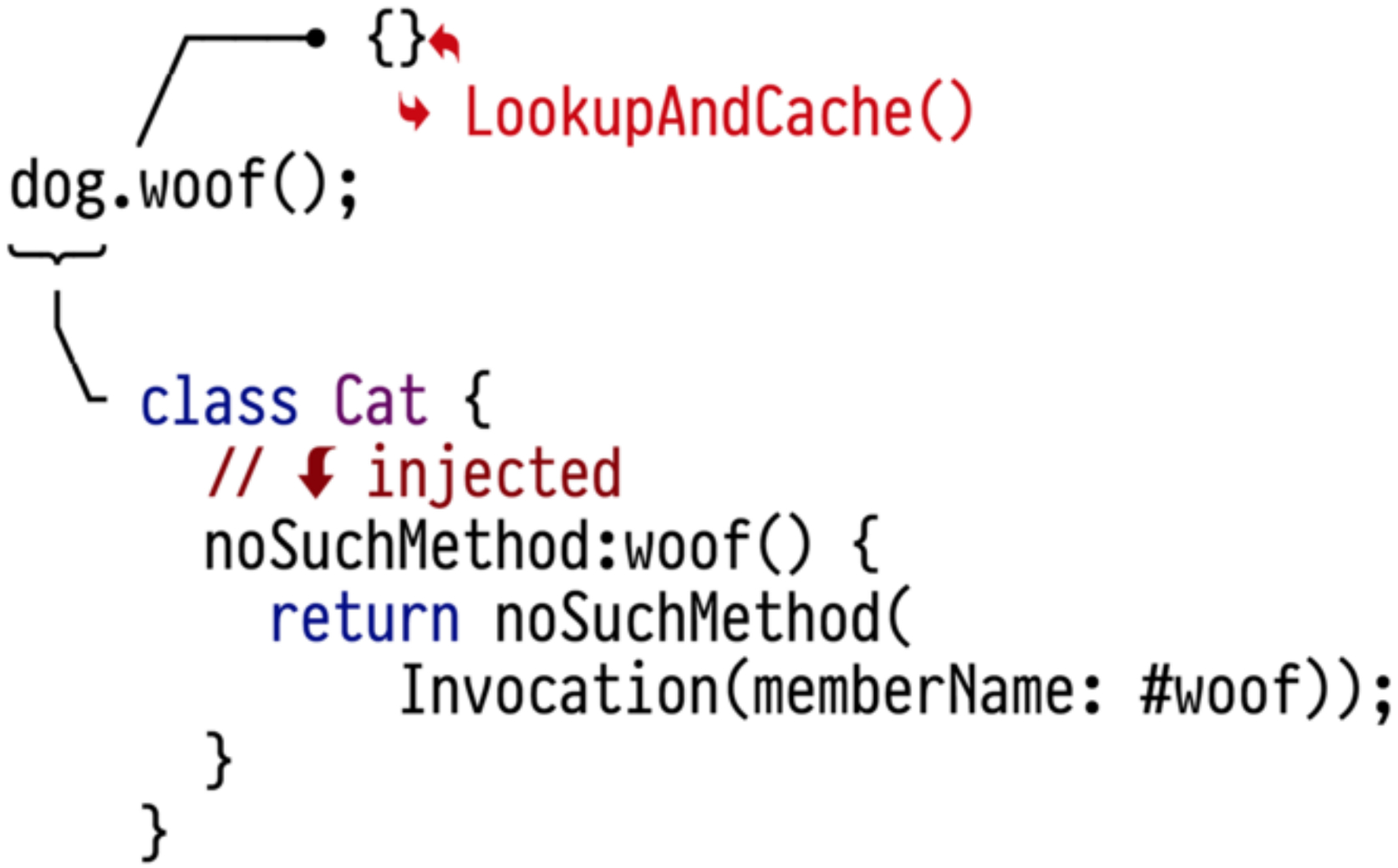

{}
↳ LookupAndCache()
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}

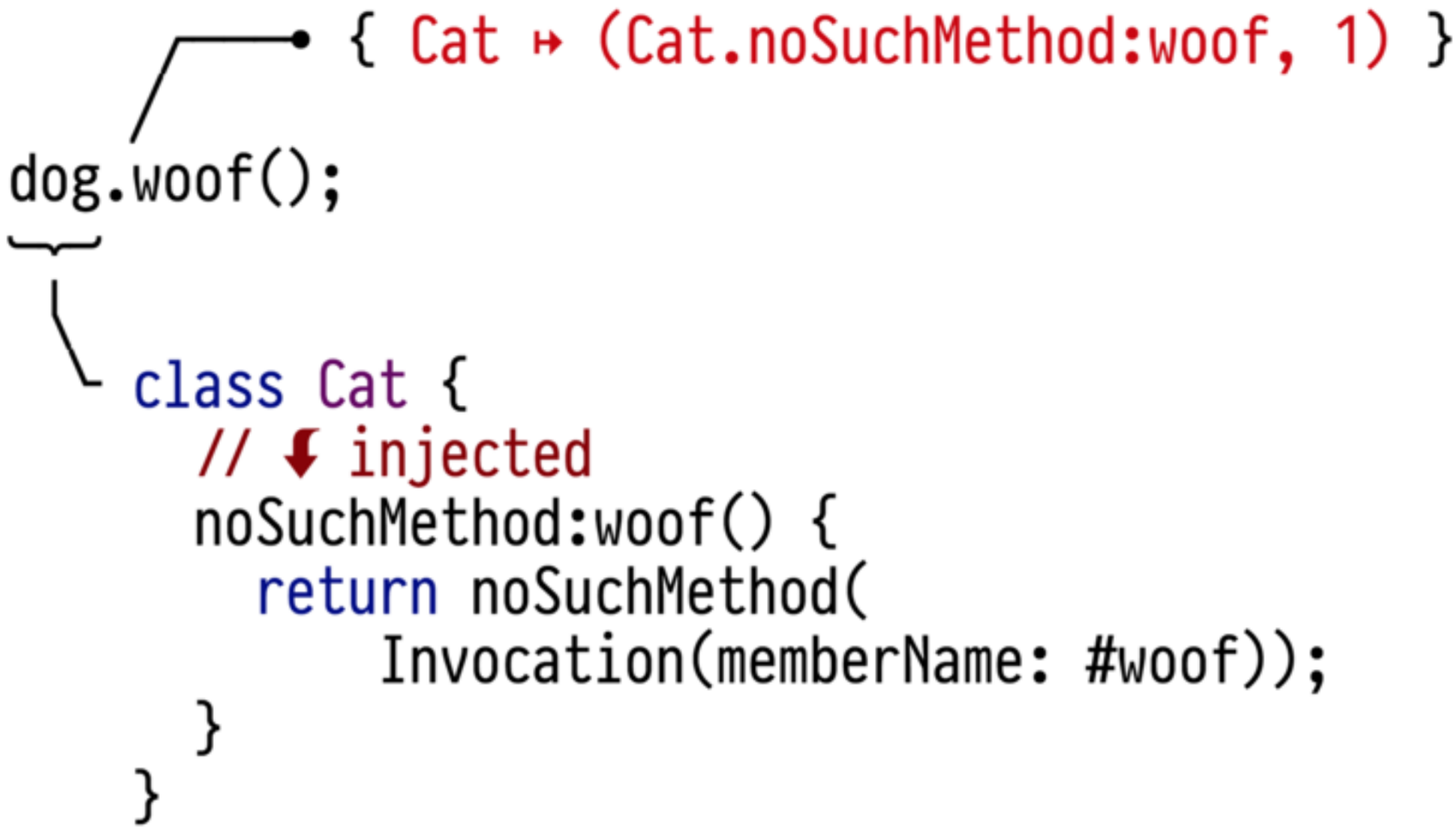

{}
dog.woof();
class Cat {
noSuchMethod(invocation) {
if (invocation.memberName == #woof) {
print('I am a 🐱 not a 🐶!');
}
}
}


{}
LookupAndCache()
dog.woof();
class Cat {
}


```
dog.woof();
```

```
class Cat {
  // ⬇ injected
  noSuchMethod:woof() {
    return noSuchMethod(
        Invocation(memberName: #woof));
  }
}
```


{}
LookupAndCache()
dog.woof();
class Cat {
// injected
noSuchMethod:woof() {
return noSuchMethod(
Invocation(memberName: #woof));
}
}

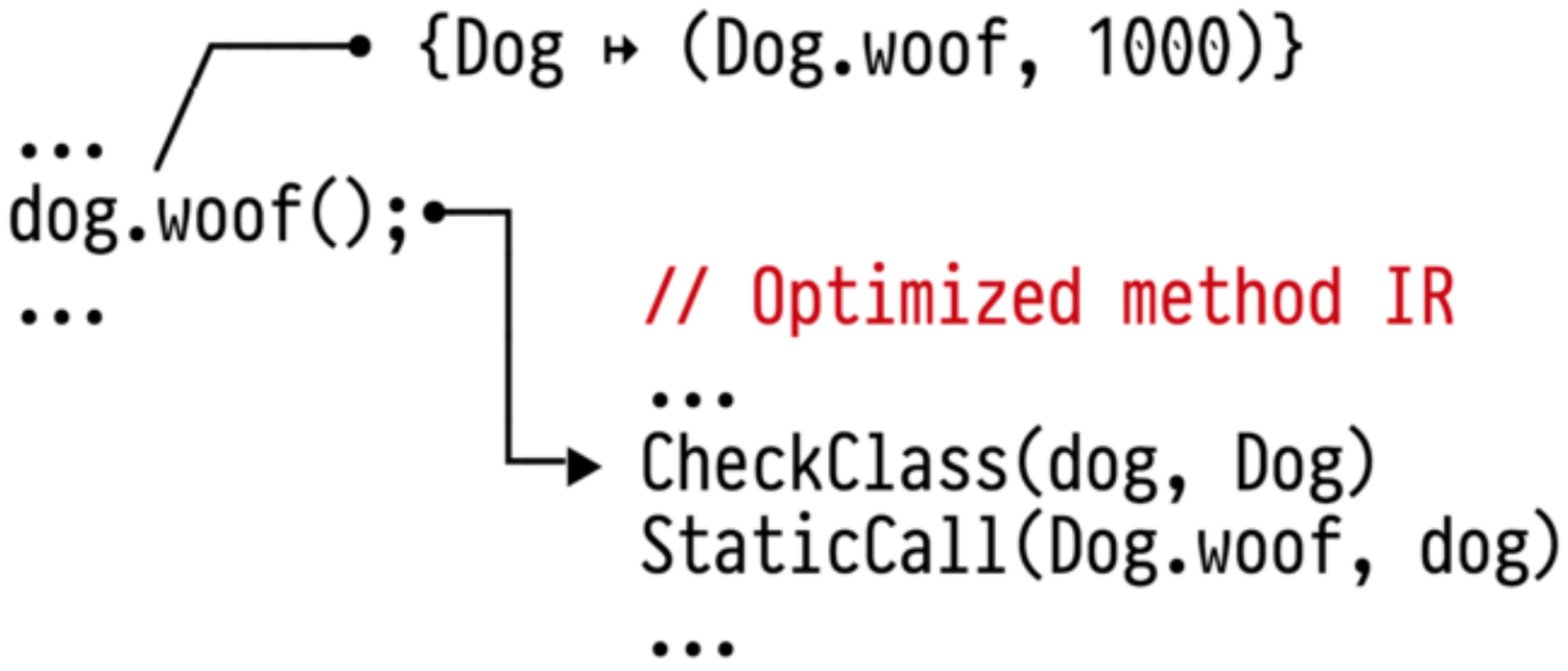

{ Cat ↦ (Cat.noSuchMethod:woof, 1) }
dog.woof();
class Cat {
// ⬇ injected
noSuchMethod:woof() {
return noSuchMethod(
Invocation(memberName: #woof));
}
}

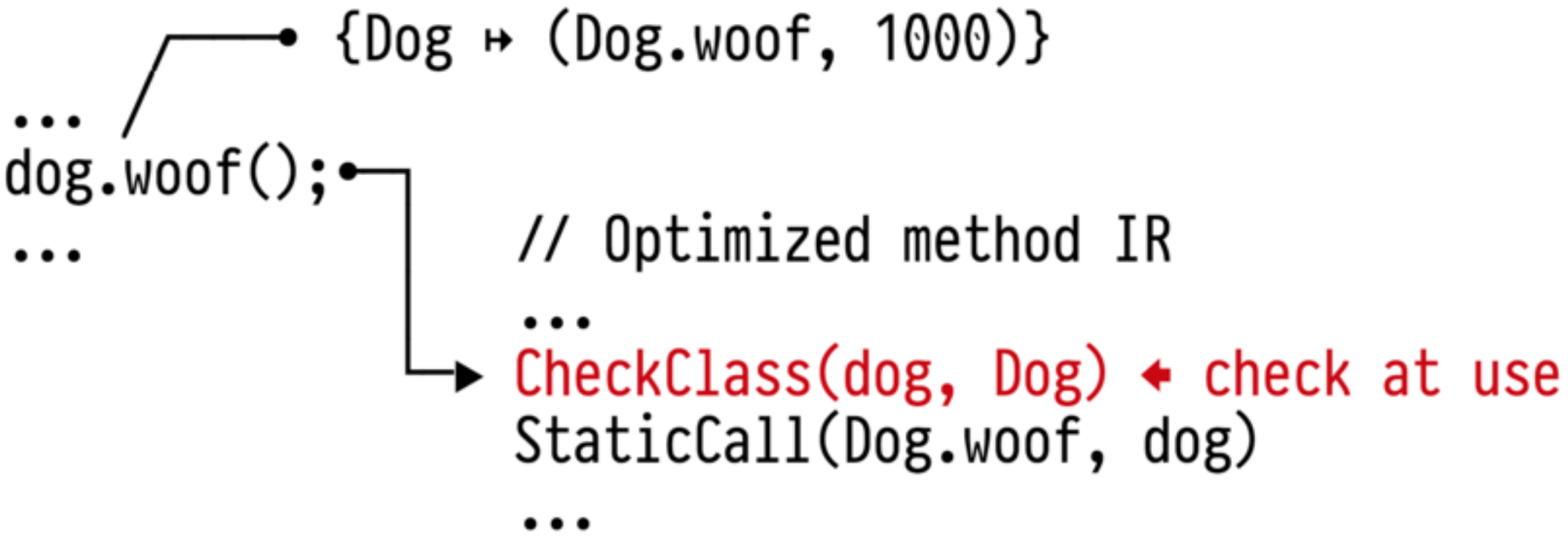
suddenly
all optimizations
just work

2012 - 2013

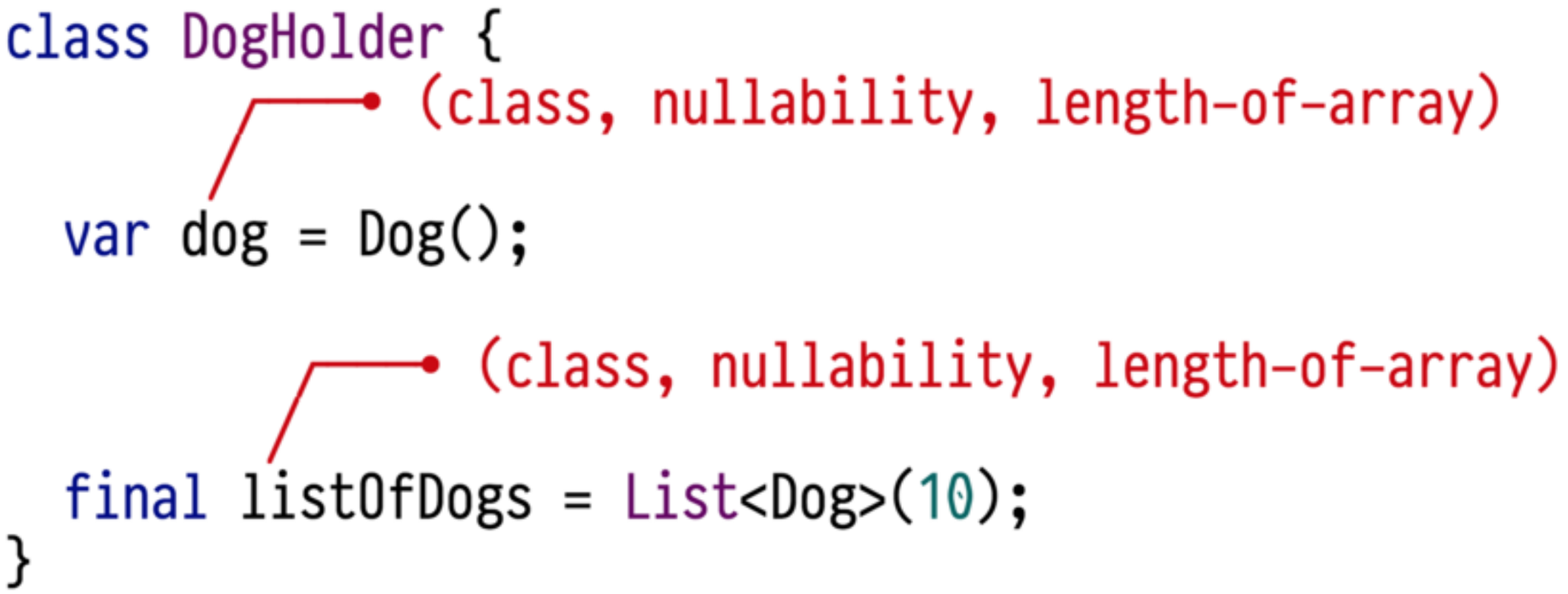
new JIT is "done"

non-classical


{Dog ↦ (Dog.woof, 1000)}
...
dog.woof();
...
// Optimized method IR
...
CheckClass(dog, Dog)
StaticCall(Dog.woof, dog)
...


{Dog ↦ (Dog.woof, 1000)}
...
dog.woof();
...
// Optimized method IR
...
CheckClass(dog, Dog) ← check at use
StaticCall(Dog.woof, dog)
...

Globally Tracked Field State



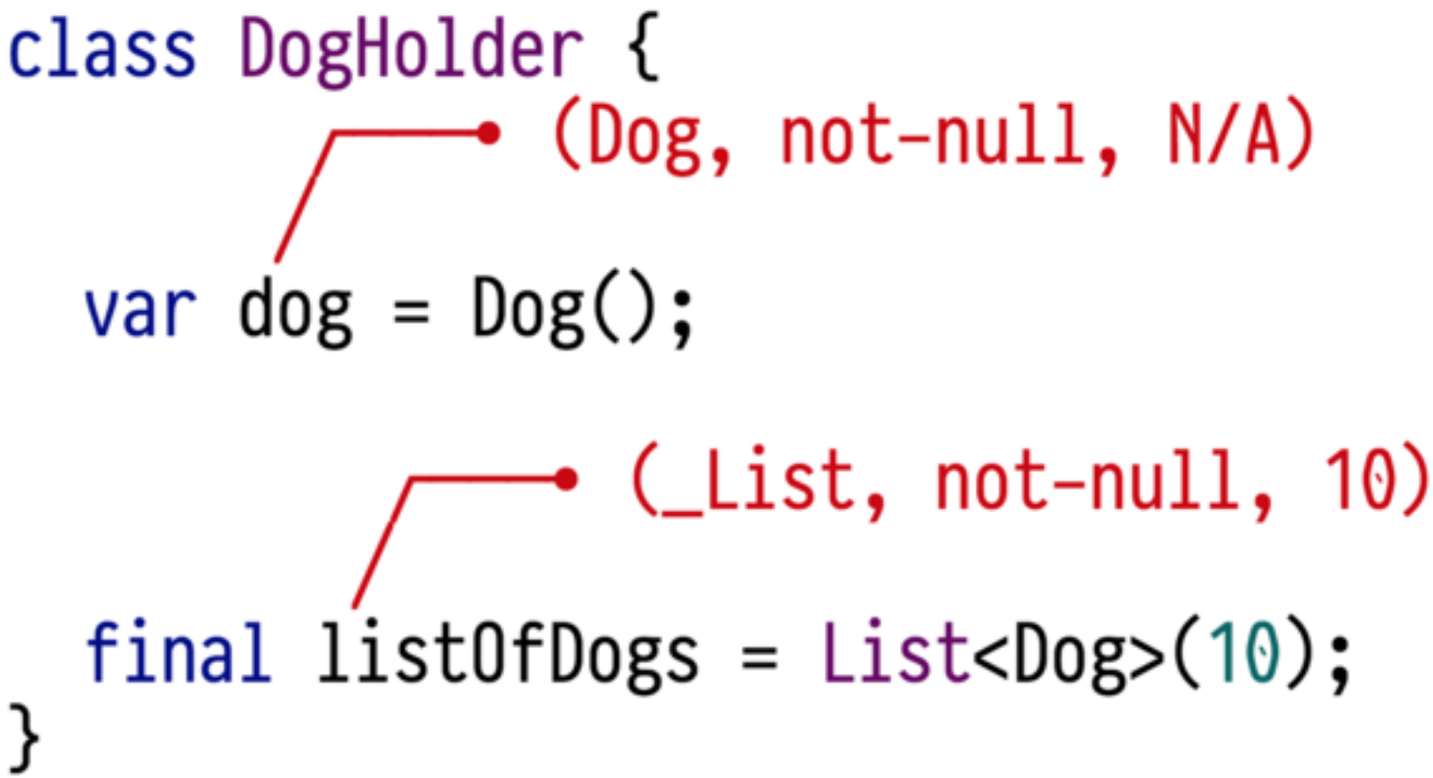
- Loads can *assume* the state

  [Concrete classes are used to eliminate `CheckClass`, length is is used to eliminate bounds checks.]

- Stores update the state

  [If incompatible change: optimized code depending on the assumption is deoptimized]

Globally Tracked Field State



Optimization shifts checks from uses to definitions

[where they often can be eliminated]

Strange Loop 2013

« Building an Optimizing Compiler for Dart »

2014

tweaking things

«  to hand-written as-sembly» effort

intrinsics rewritten in IL

[planned to move stubs too, still not started on that]

irregexp port

- irregexp is a V8's JITing RE engine
- in V8 it has platform specific backends
- in Dart we just generate IL from RegExp-s

« Wait, I thought Dart is supposed to be a Web language 🤔 »

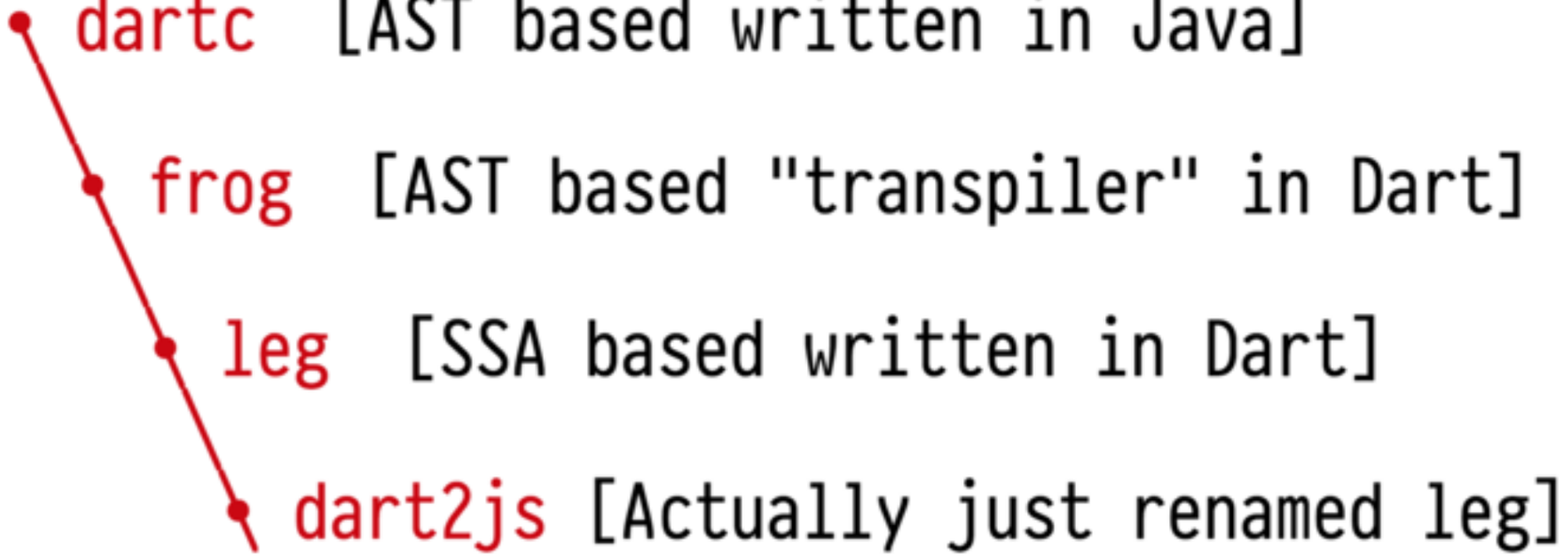
VM

dart2js

VM

dart2js

Parallel evolution of Dart-to-JS compilers



AOT compilation of Dart to JS is hard

Object model mismatch

can not completely erase Dart object model in JS

representation only as efficient as JS engine allows it

Open the Sky

Browse files

master fitness_apk_4 ··· demo_apk_22

abarth committed on Oct 23, 2014 1 parent 20cc569 commit ae72930937b1a1522d1919a82ddc21da7d953229

Showing **4,395 changed files** with **688,796 additions** and **0 deletions.**

Unified | Split

The diff you're trying to view is too large. We only load the first 3000 changed files.

«What would happen if you take WebView and remove the constraint that it has to render web-pages?»

— Adam Barth at FLOSS Weekly 439

Initially Sky started with JS

but later moved to Dart *entirely*

[with C++ for lowest *engine* layers]

and renamed

Flutter

talk from Strange Loop 2017

« Flutter: How we're building a UI framework for tomorrow at Google» by Eric Seidel

Flutter

is a mobile framework

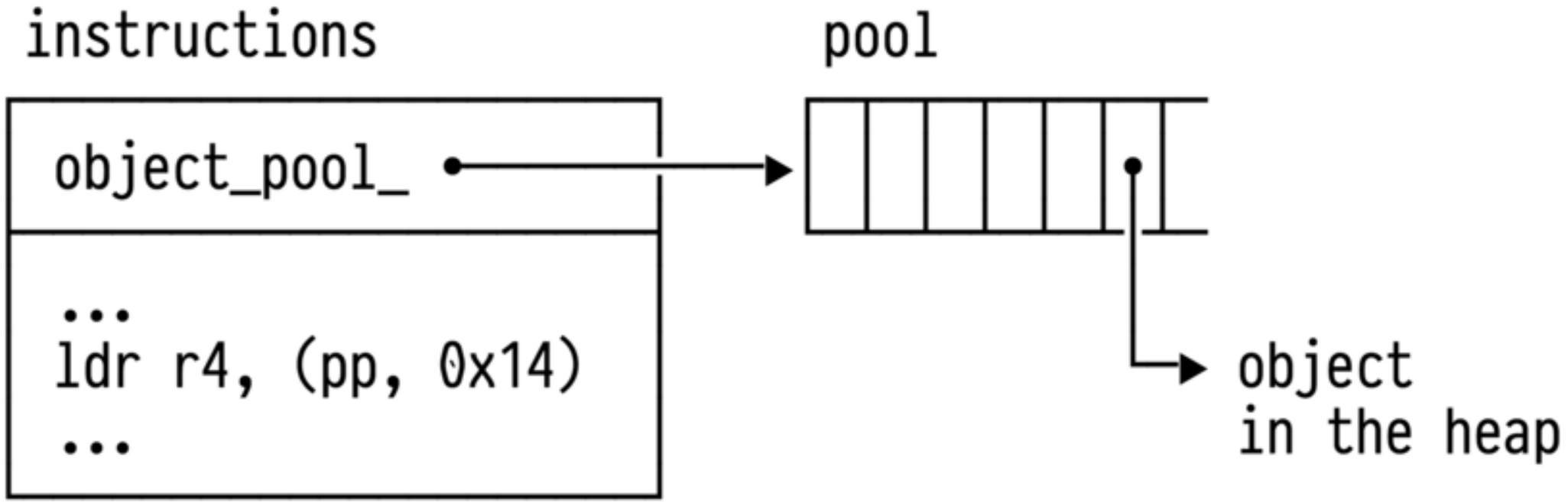
iOS does not allow JITing

[… unless you are Mobile Safari]

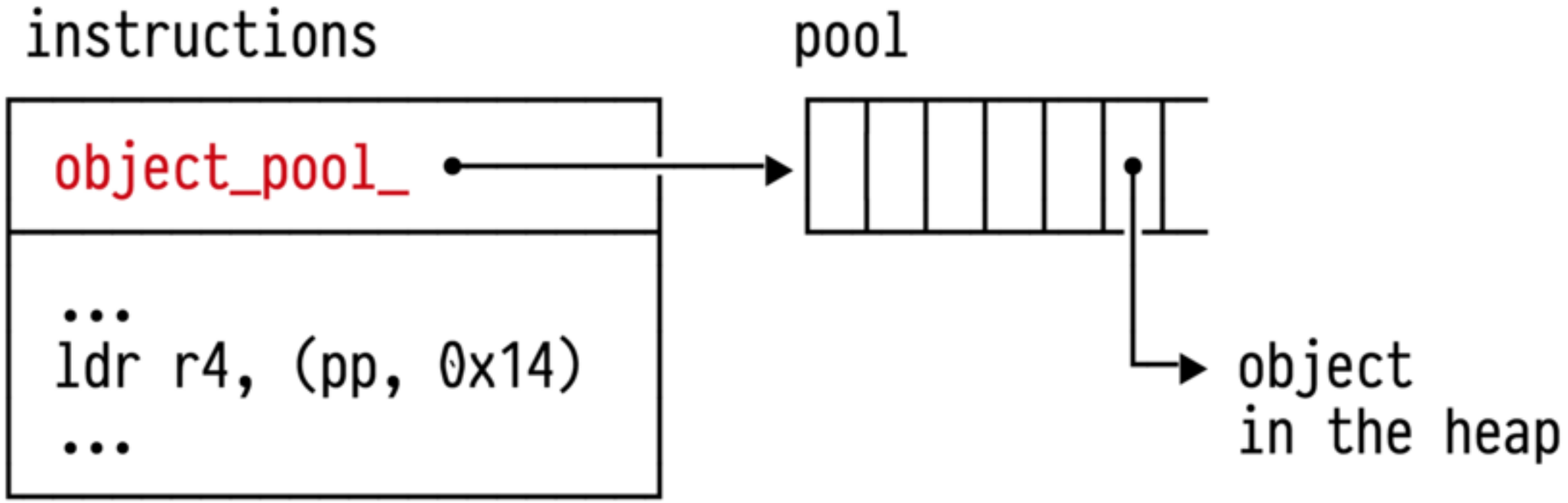

AOT

2015

started AOT prototype


instructions
pool
object_pool_
...
ldr r4, (pp, 0x14)
...
object
in the heap

JIT code is relocatable


instructions
pool
object_pool_
...
ldr r4, (pp, 0x14)
...
object
in the heap

had to change calling conventions

[want instructions to be RO and *isolate* independent, while pools are isolate dependent - so instructions can't contain pointer to the pool]

« don't want flags & no-body cares about JIT »

[changed calling conventions in a way that impacted JIT performance — in retrospect was a mistake]

then just
serialize it!

already had *snapshots*

[kinda like a heap dump that you can hydrate back]

how to get "JIT" code?

[real JIT does speculative opts!]

disable speculations then

force compile everything

DONE

[prototype took around 2-3 months]

"DONE"

[...]

"DONE"

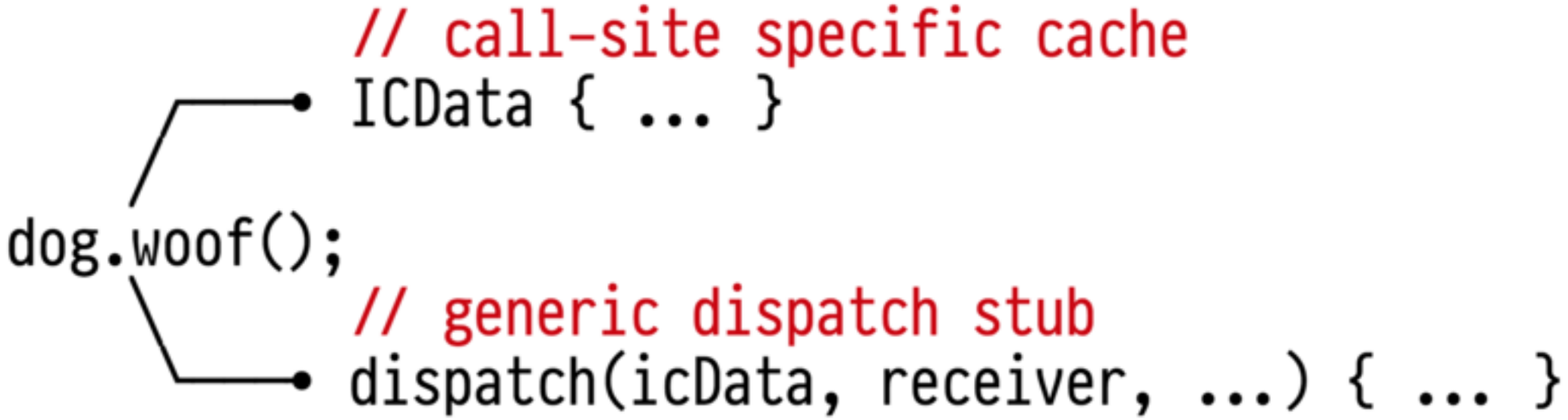
[...]

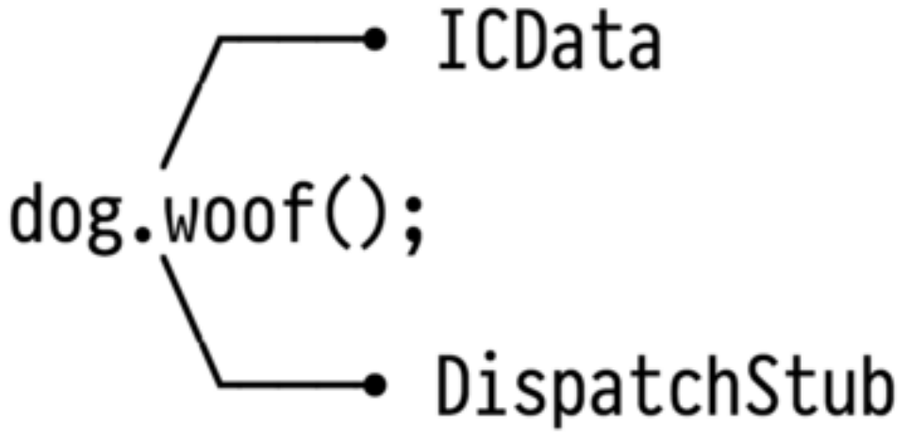
AOT compilation of Dart to JS is hard

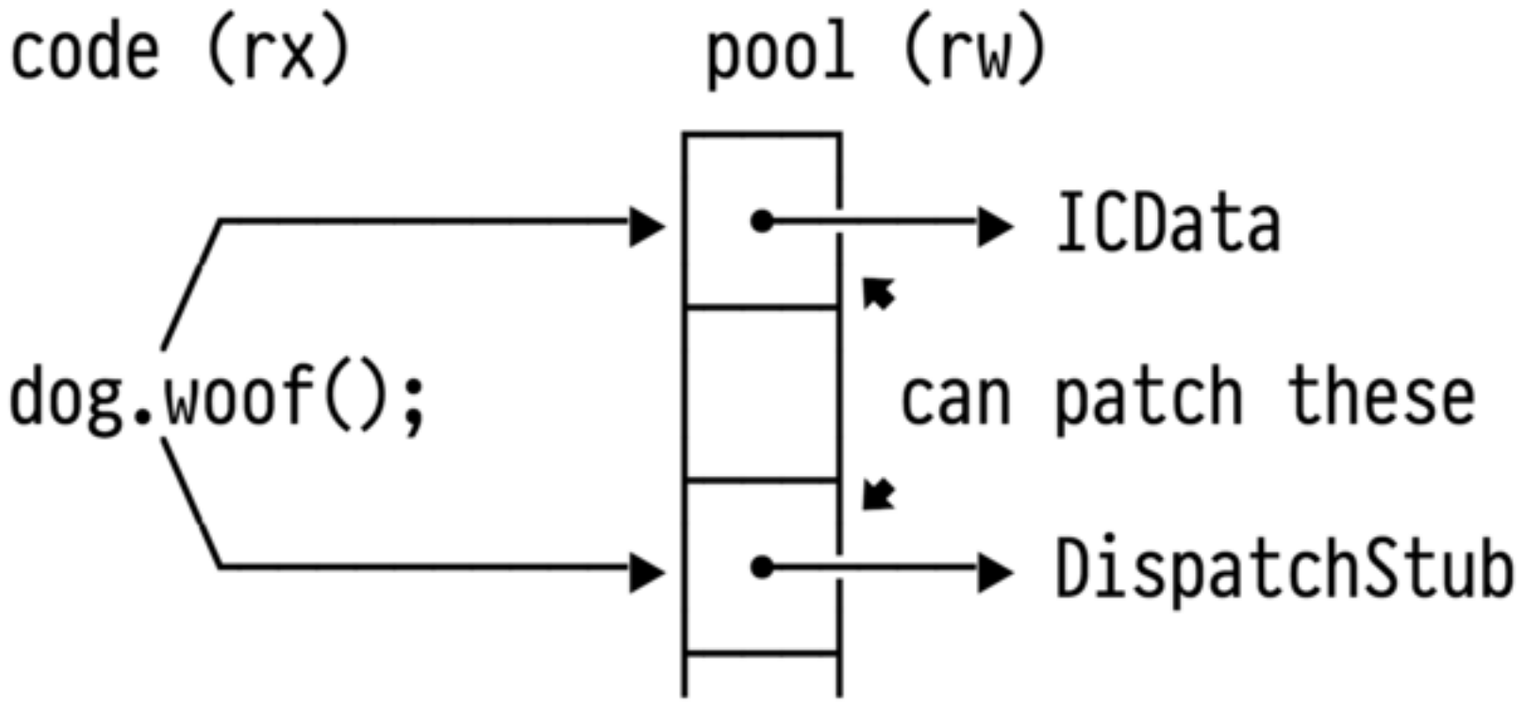
AOT compilation of Dart to [____] is nontrivial

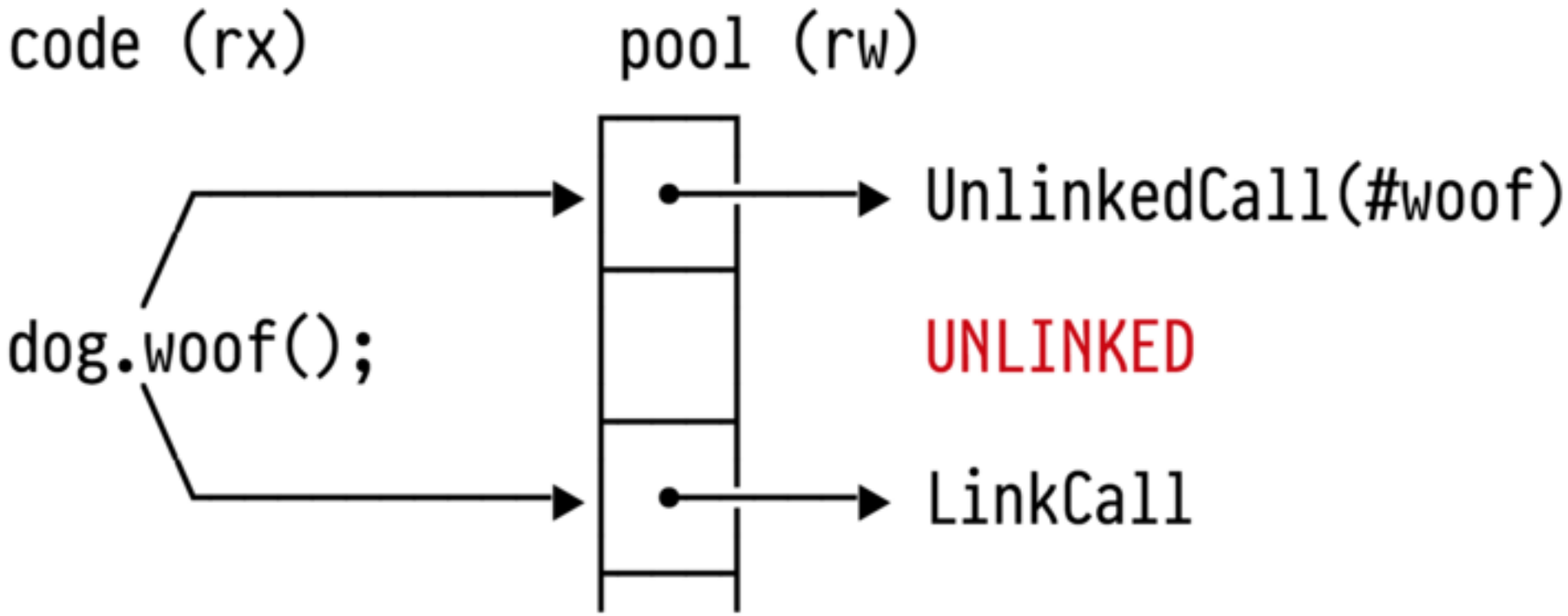
need replacement
for SPECULATIONS

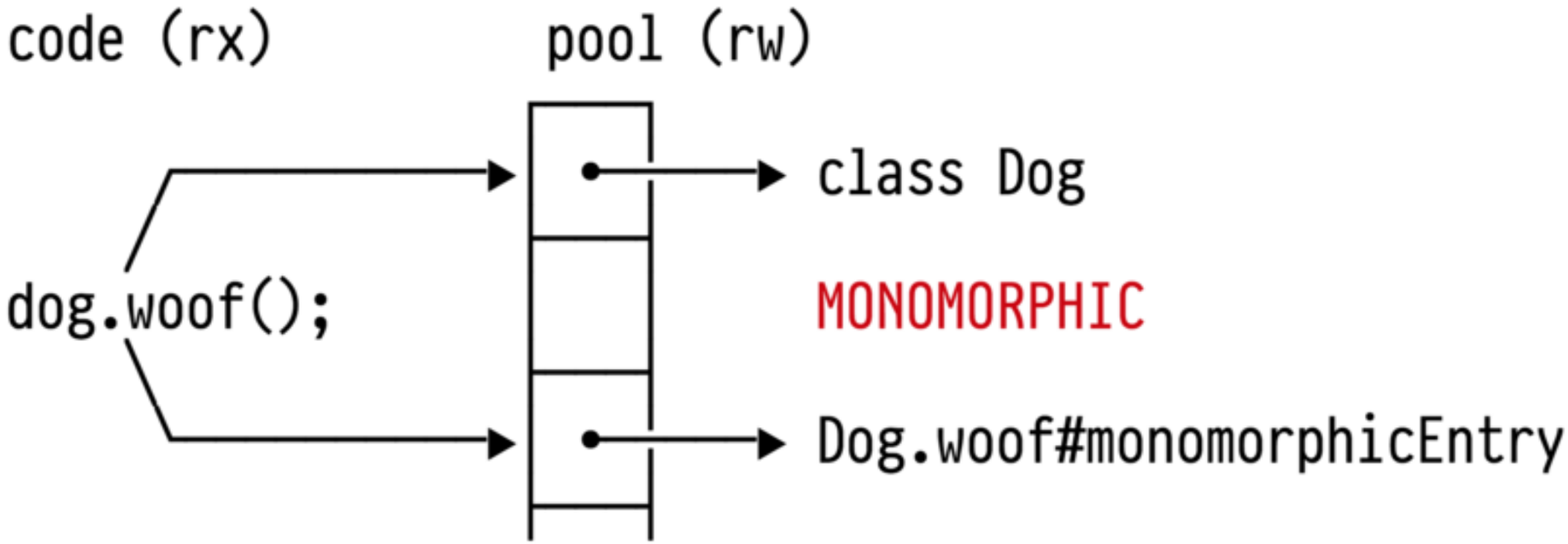
switchable calls

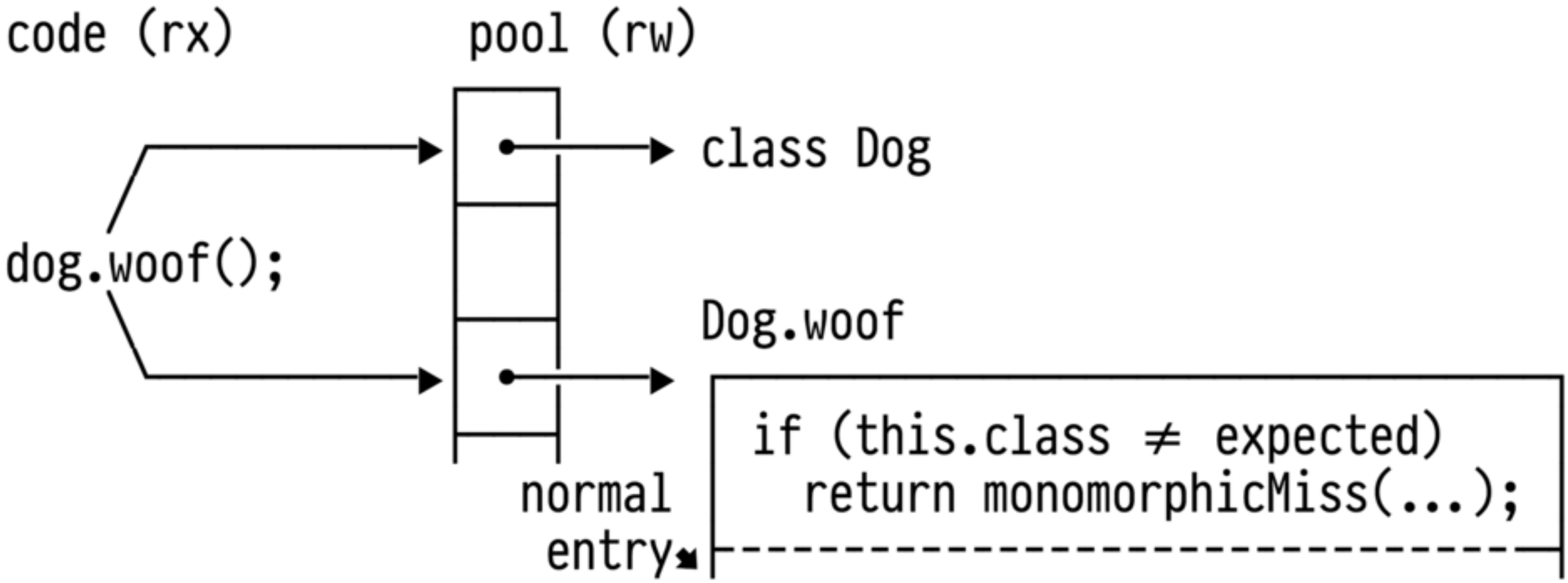

// call-site specific cache
ICData { ... }
dog.woof();
// generic dispatch stub
dispatch(icData, receiver, ...) { ... }

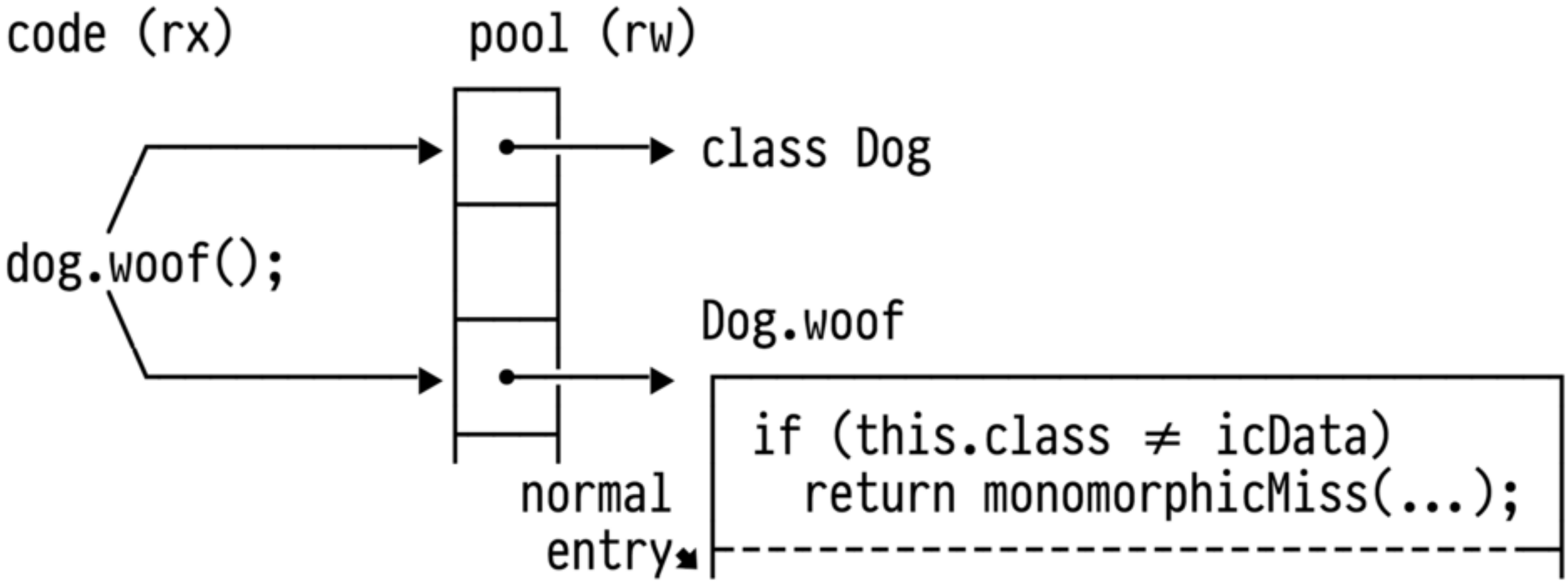

ICData
dog.woof();
DispatchStub

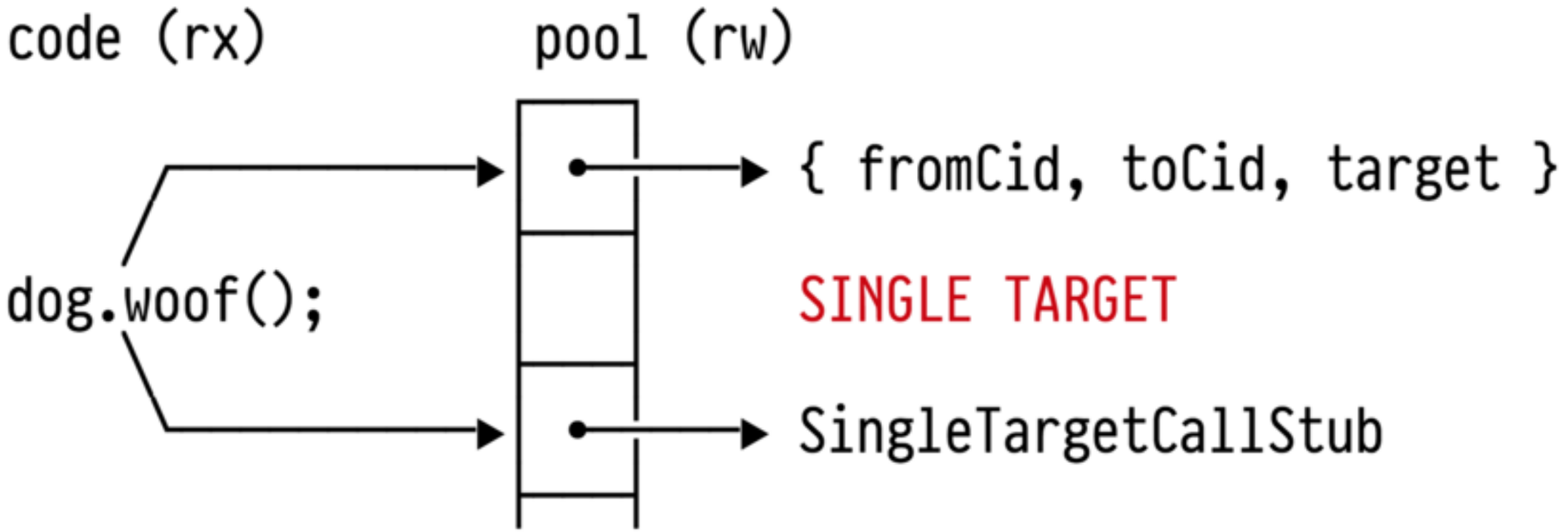

code (rx)
pool (rw)
ICData
dog.woof();
can patch these
DispatchStub

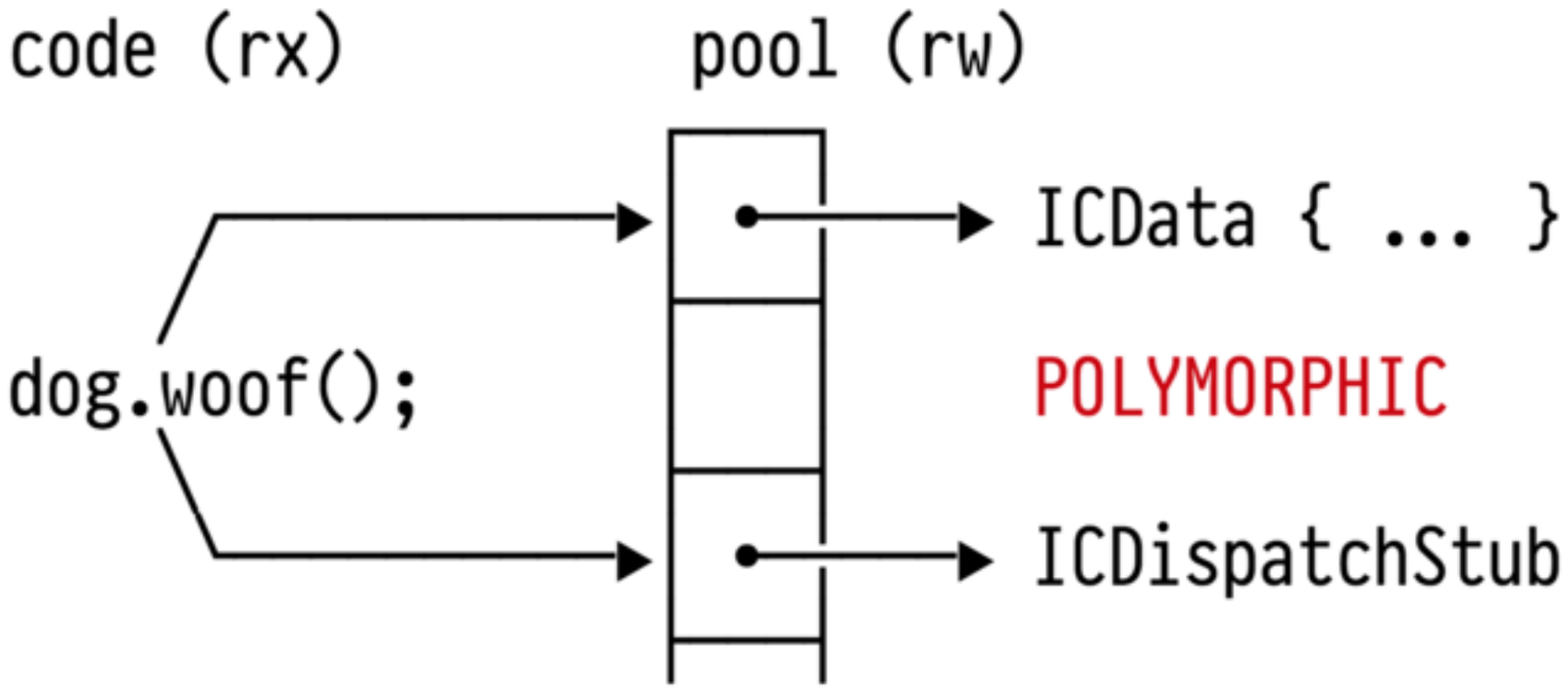

code (rx)
pool (rw)
UnlinkedCall(#woof)
dog.woof();
UNLINKED
LinkCall

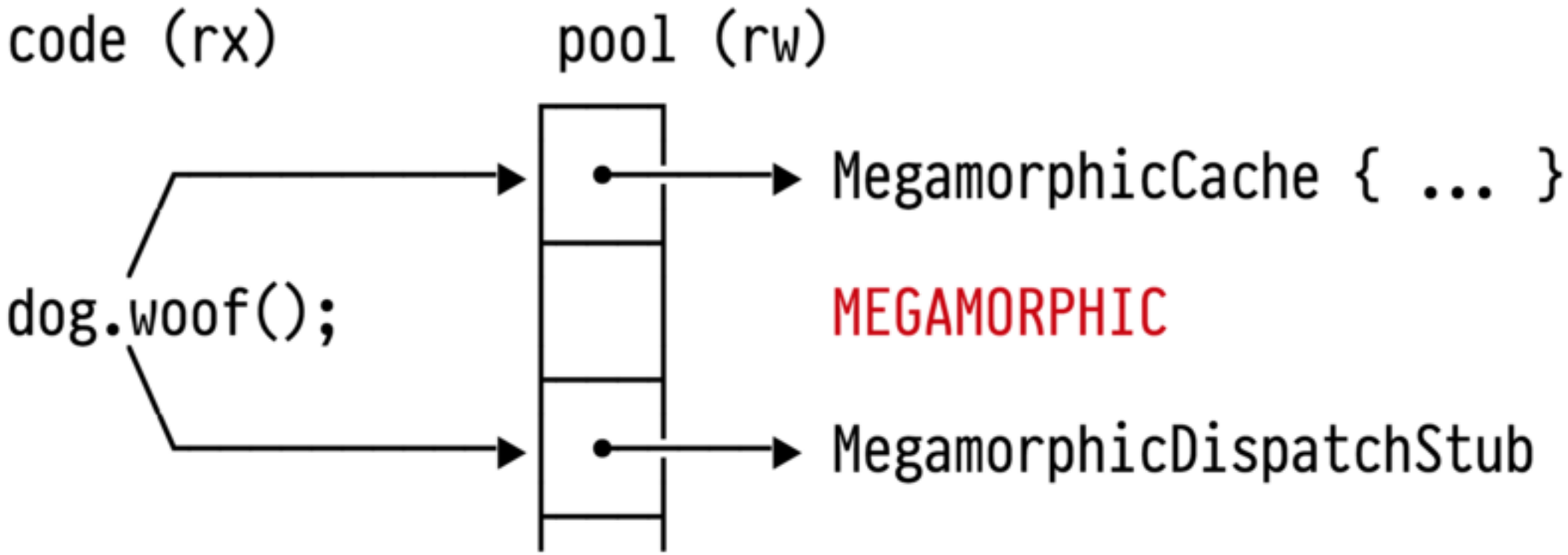

code (rx)
pool (rw)
class Dog
dog.woof();
MONOMORPHIC
Dog.woof#monomorphicEntry


code (rx)
pool (rw)
class Dog
dog.woof();
Dog.woof
if (this.class ≠ expected)
return monomorphicMiss(...);
normal
entry


code (rx)
pool (rw)
class Dog
dog.woof();
Dog.woof
if (this.class ≠ icData)
return monomorphicMiss(...);
normal
entry

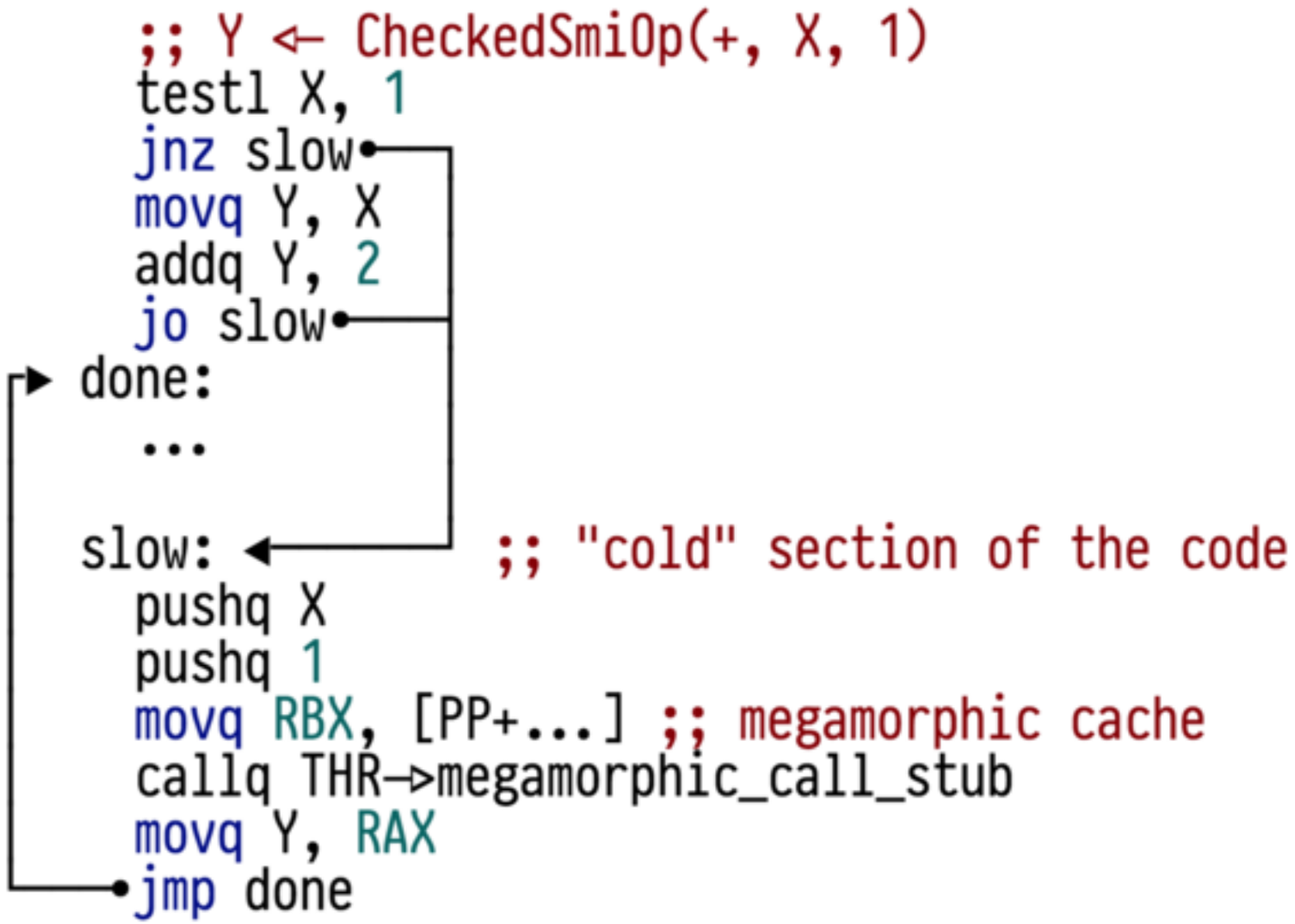

code (rx)
pool (rw)
{ fromCid, toCid, target }
dog.woof();
SINGLE TARGET
SingleTargetCallStub

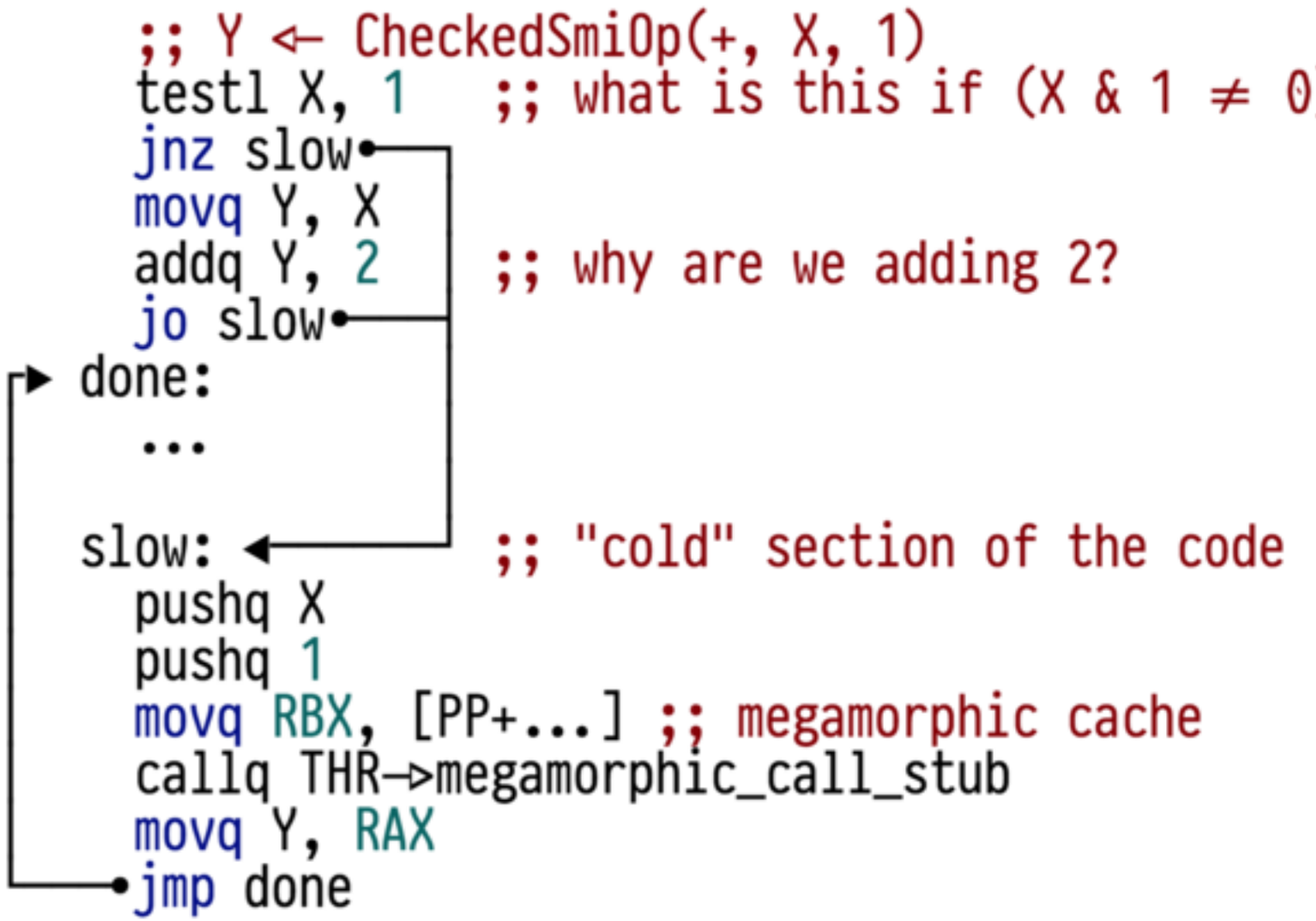

code (rx)
pool (rw)
dog.woof();
ICData { ... }
POLYMORPHIC
ICDispatchStub


code (rx)
pool (rw)
dog.woof();
MegamorphicCache { ... }
MEGAMORPHIC
MegamorphicDispatchStub

inline arithmetic fast-paths

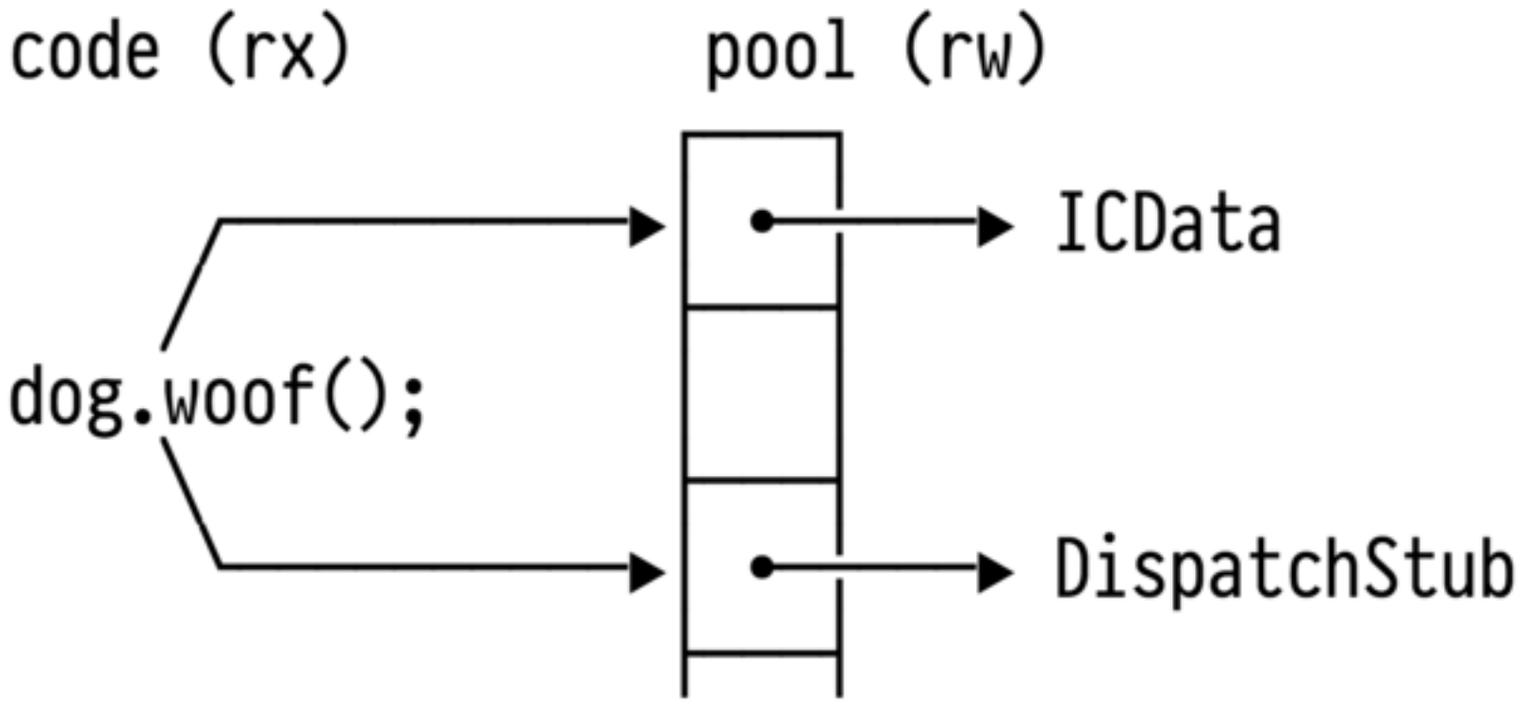
- operators can be overriden
- `int` is nullable in Dart
- `int` *was* arbitrary precision

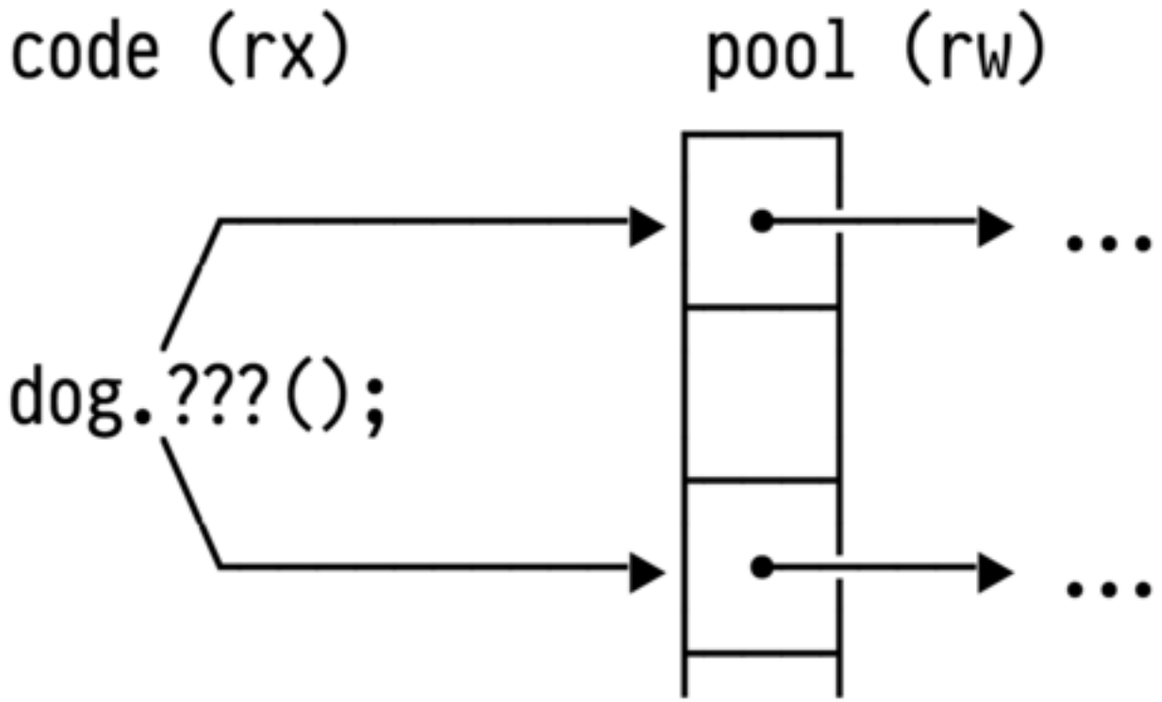

;; Y ← CheckedSmiOp(+, X, 1)
testl X, 1
jnz slow
movq Y, X
addq Y, 2
jo slow
done:
...
slow: ;; "cold" section of the code
pushq X
pushq 1
movq RBX, [PP+...] ;; megamorphic cache
callq THR->megamorphic_call_stub
movq Y, RAX
jmp done


;; Y ← CheckedSmiOp(+, X, 1)
testl X, 1 ;; what is this if (X & 1 ≠ 0) check?
jnz slow
movq Y, X
addq Y, 2 ;; why are we adding 2?
jo slow
done:
...
slow: ;; "cold" section of the code
pushq X
pushq 1
movq RBX, [PP+...] ;; megamorphic cache
callq THR->megamorphic_call_stub
movq Y, RAX
jmp done

plus some
closed-world
CHA opts

what about code size?

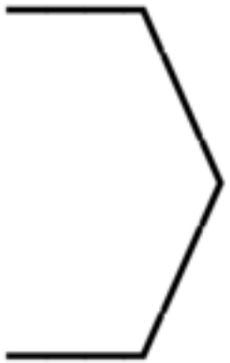

code (rx)
pool (rw)
ICData
dog.woof();
DispatchStub


code (rx)
pool (rw)
dog.???();
...
...

```
void doOne(x) {
  return x.foo().bar();
}

void doTwo(z) {
  return z.baz().quux();
}
```

```
void doOne(p0) {
  return p0.???().???();
}

void doTwo(p0) {
  return p0.???().???();
}
```

```
void doOne(p0) {
  return p0.???().???();
}

void doTwo(p0) {
  return p0.???().???();
}
```



same machine code
and can be shared

byproduct of AOT work *AppJIT snapshots*

[warmup the application - then serialize JIT code; massive reduction in warmup time — used by our tools now]

only so much can be done

[remember for example dynamic method injection?]

2016

the wind of changes

(1) dart2js is too slow for development

Dart Dev Compiler

has been cooking since 2014

Dart Dev Compiler

fast-modular compiles

Dart Dev Compiler

requires stronger type system

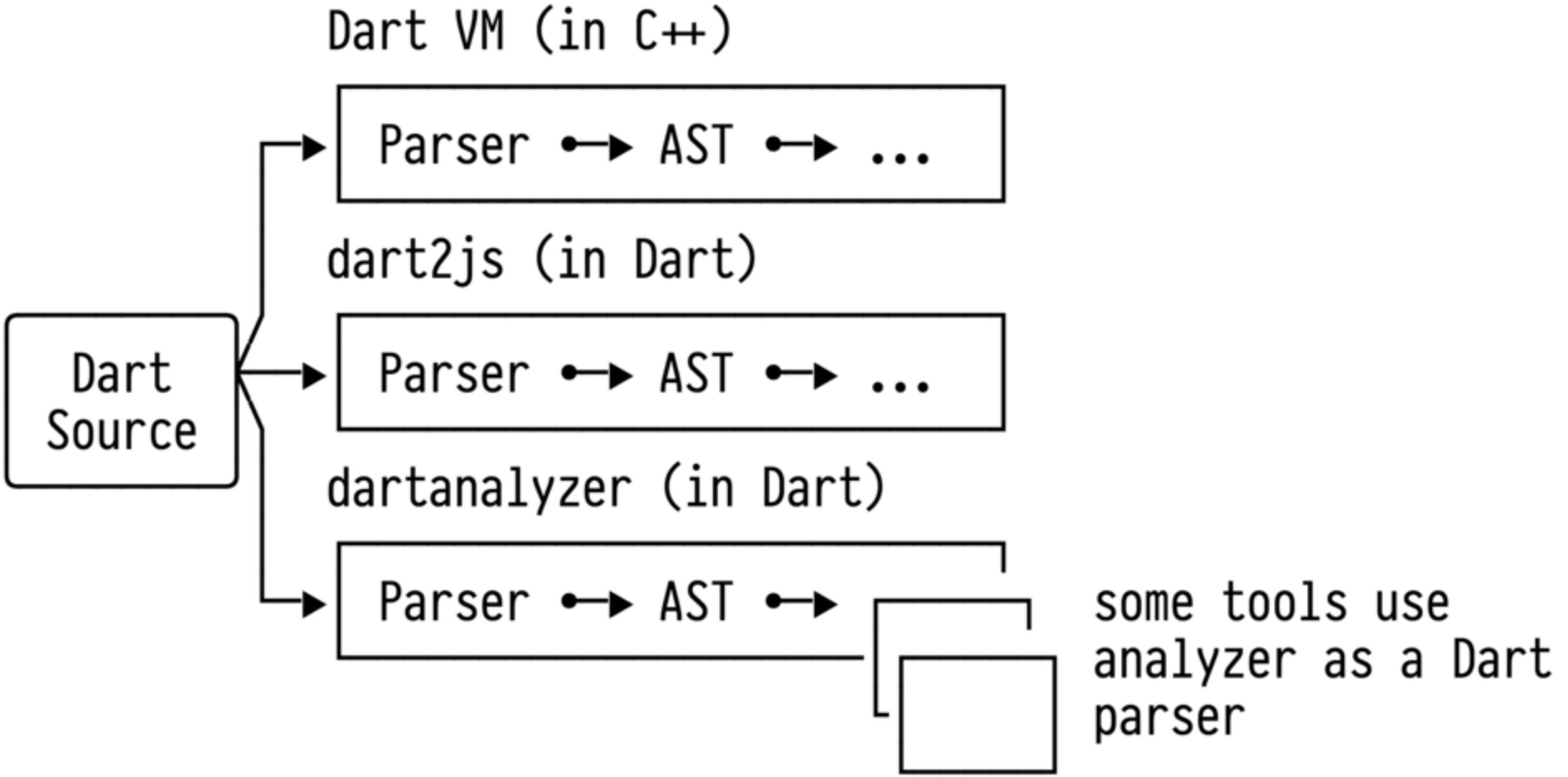
```
Dog d;  // is guaranteed to be a 🐶

List<Cat> x;
x[0];   // is guaranteed to be a 🐱
```

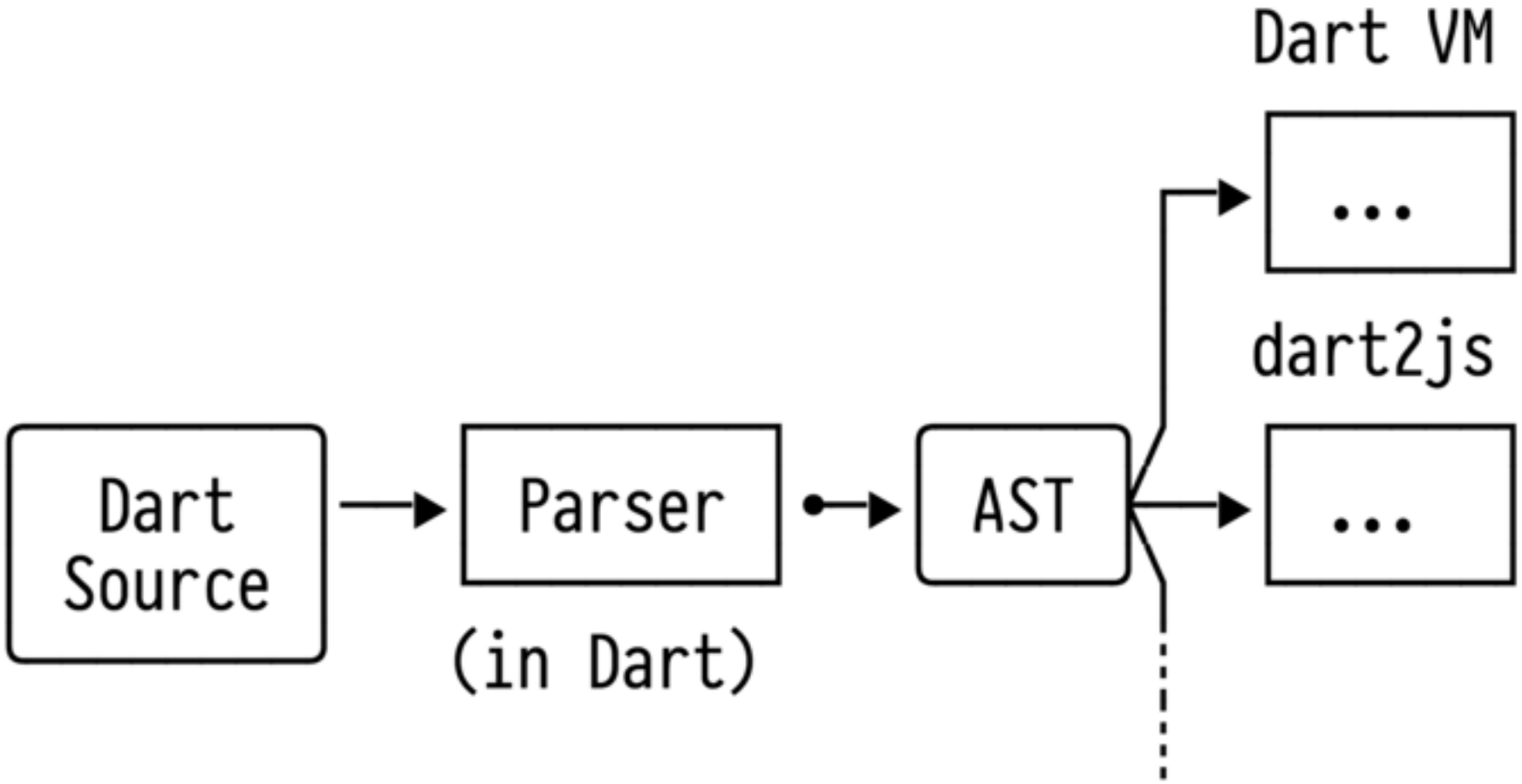
talk from Dart Conf 2018

« Evolving Dart: Leaving the ocean and learning to fly »

by Leaf Petersen

(2) every Dart tool has its own parser


Dart VM (in C++)
Parser → AST → ...
dart2js (in Dart)
Dart Source
Parser → AST → ...
dartanalyzer (in Dart)
Parser → AST →
some tools use analyzer as a Dart parser


Dart VM
...
dart2js
Dart Source
Parser
(in Dart)
AST
...

2016

new type system
& common front-end
migration starts

Aug 2018

Dart 2 shipped

tons of *gnarly* work

[cool stuff too: e.g. started implementing optimization passes in Dart]

is AOT trivial now?

- many important types are *interfaces*, e.g. `List<T>`
- `int` and `double` are still nullable
- `int` became 64-bit wrap around integer
- *soundness* requires runtime type checks

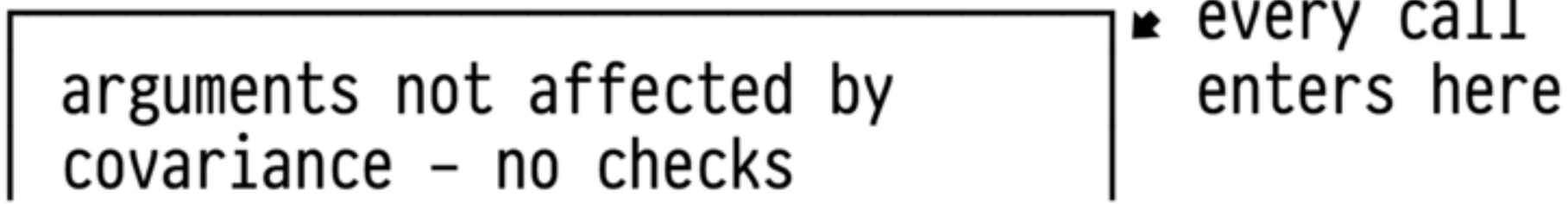
```dart
// In Dart all generics are covariant
var strings = ["a", "b"];  // static type List<String>
strings.add(10);           // compile time error

// Upcast List<String> to List<Object>
List<Object> objects = strings; // ok
objects.add(10);           // runtime error
                           // requires check somewhere
```

```dart
// In Dart all generics are covariant
var strings = ["a", "b"];  // static type List<String>
strings.add(10);           // compile time error

// Upcast List<String> to List<Object>
List<Object> objects = strings; // ok
objects.add(10);                // runtime error
                                // requires check somewhere

// Also Dart 2 still has dynamic
dynamic x = strings;
x.append("a");  // runtime error
x.add(10);      // runtime error
```

Current approach:

- track type arguments (statically in AOT, dynamically in JIT)
- bypass type checks when runtime type arguments are known to match static type arguments

List<T>.join(String sep)



```
List<T>.join(String sep)
```

arguments not affected by covariance – no checks

every call enters here except dynamic ones

```
dynamic x = [...];
x.join(10);
```

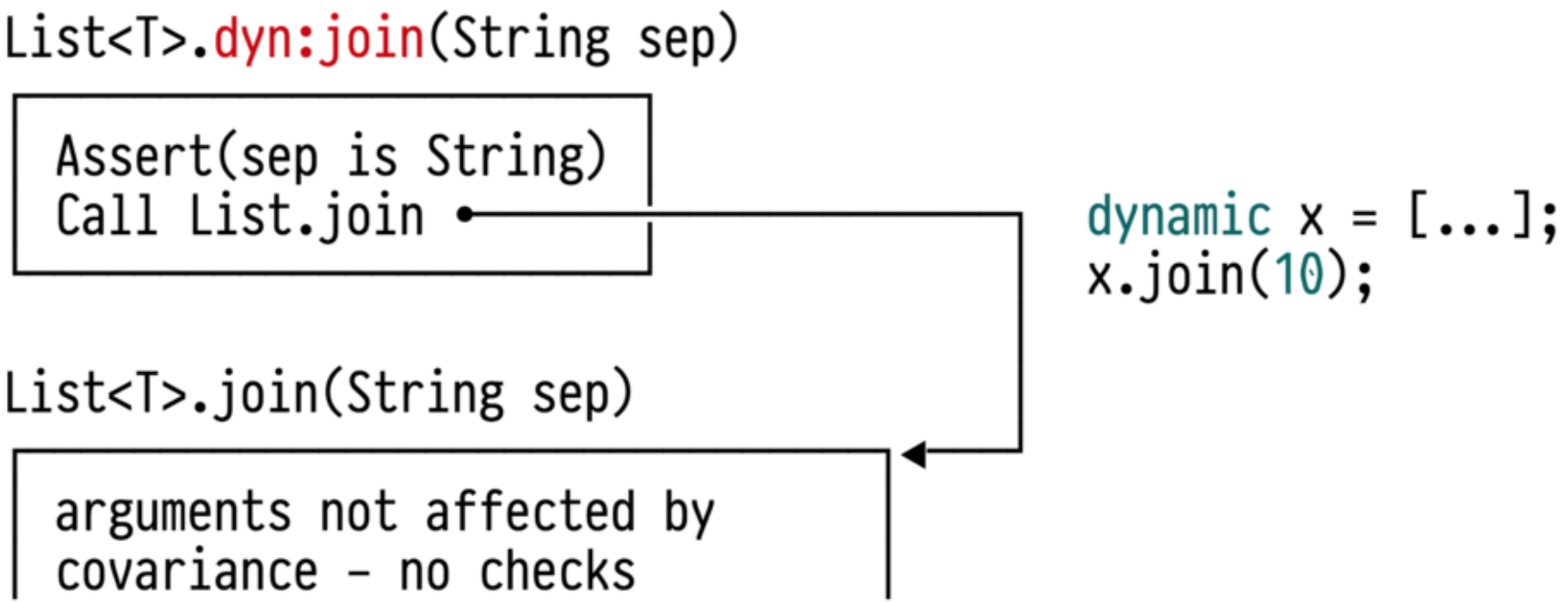
List<T>.join(String sep)

arguments not affected by covariance - no checks

```
dynamic x = [...];
x.dyn:join(10);
```

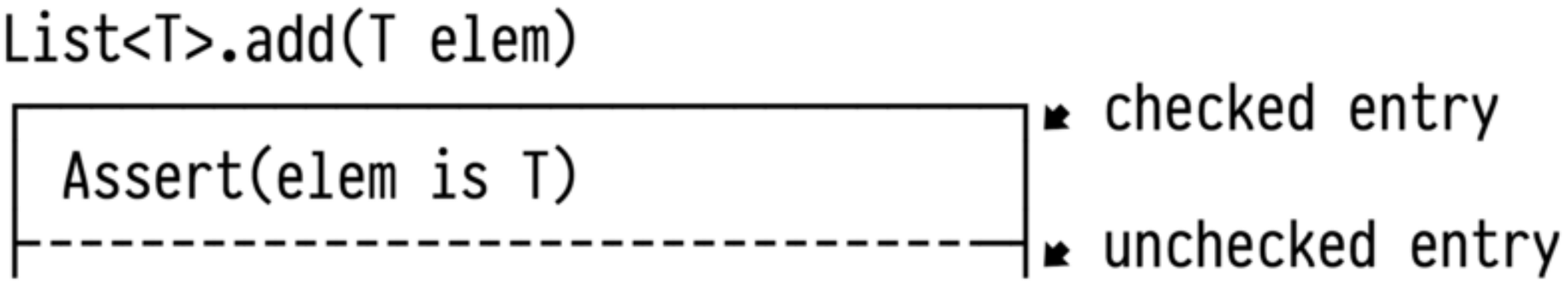
List<T>.join(String sep)

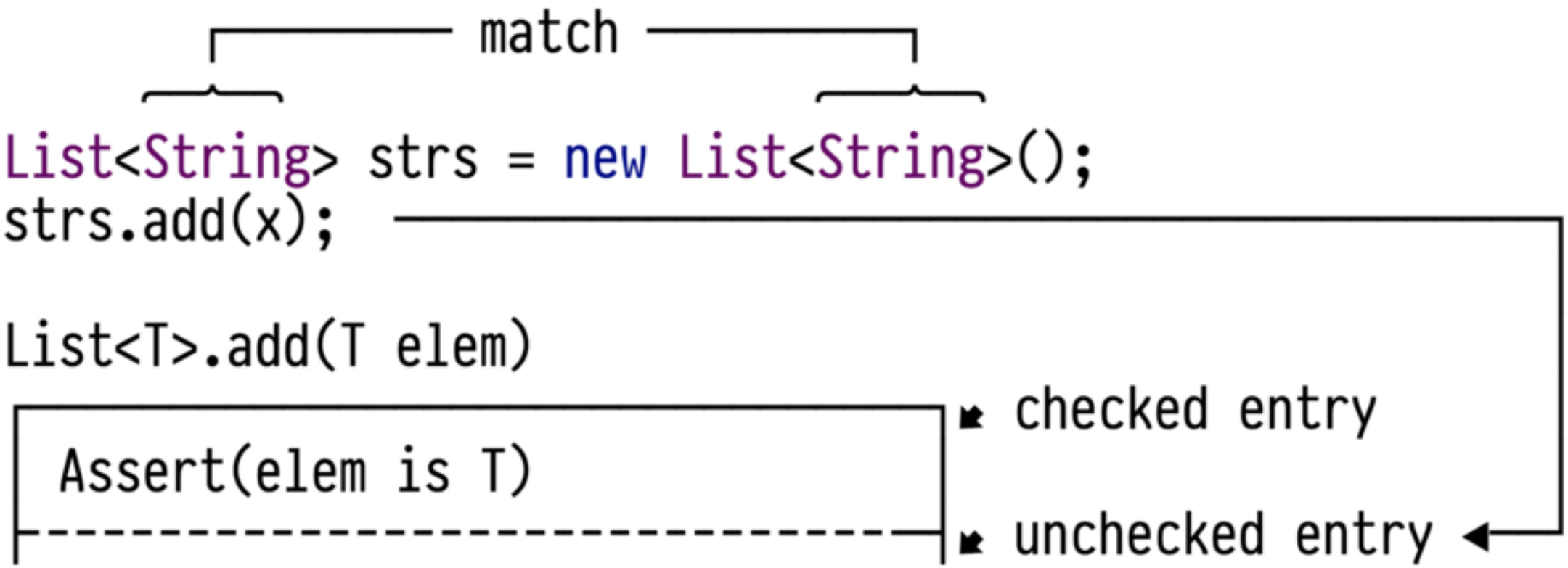
arguments not affected by covariance - no checks

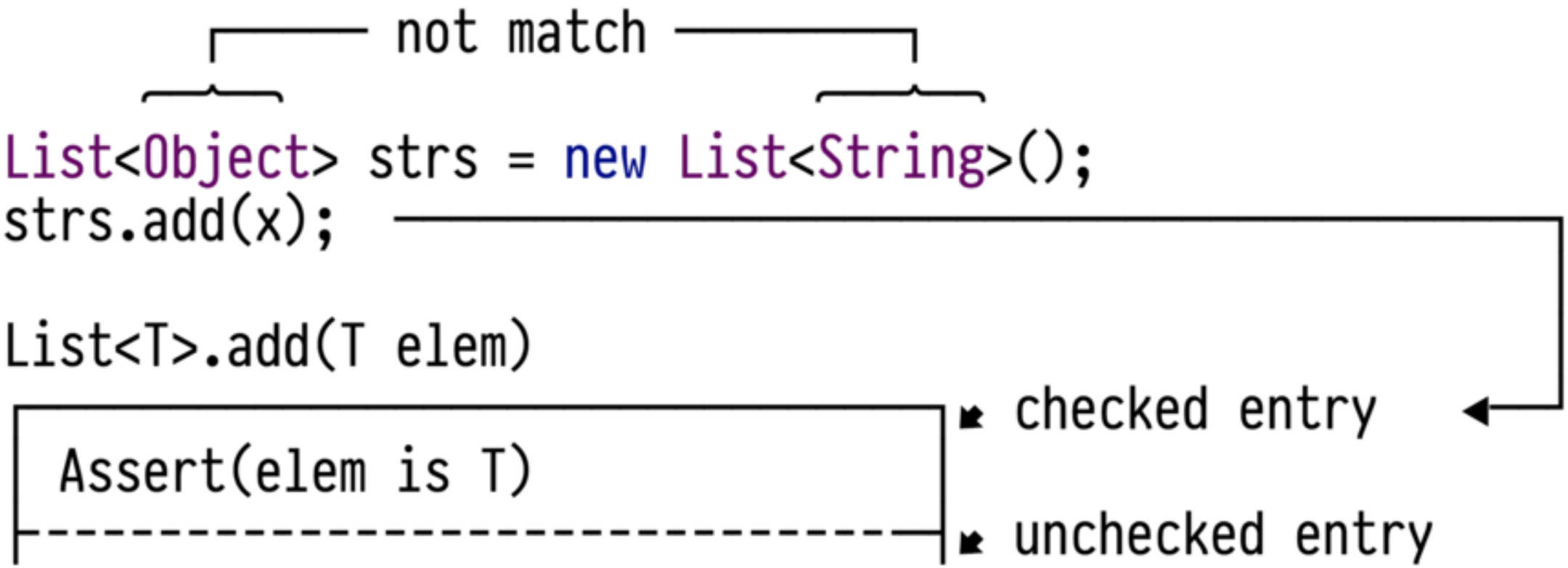

List<T>.dyn:join(String sep)
Assert(sep is String)
Call List.join
dynamic x = [...];
x.join(10);
List<T>.join(String sep)
arguments not affected by
covariance - no checks

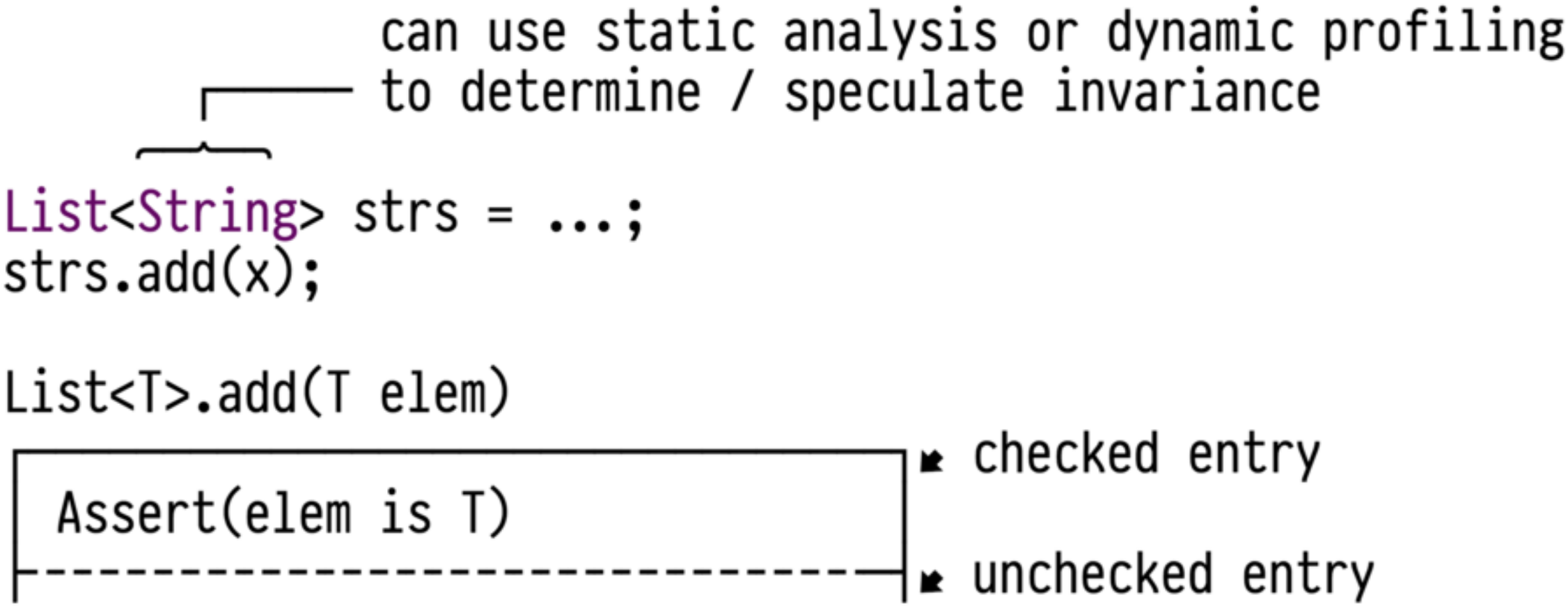
```
List<T>.add(T elem)
```

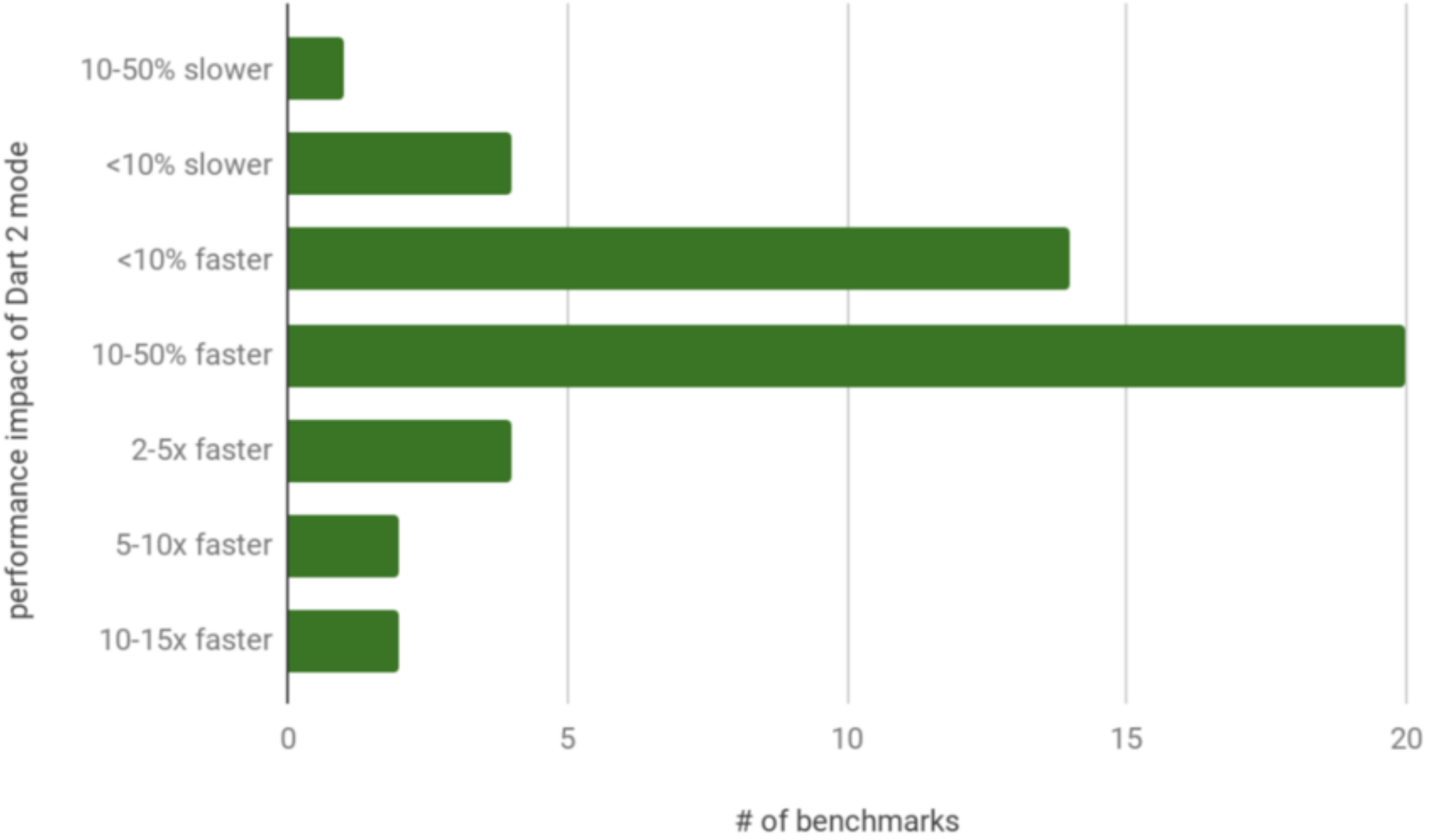
arguments are affected by covariance – need to check


List<T>.add(T elem)
Assert(elem is T)
checked entry
unchecked entry


match
List<String> strs = new List<String>();
strs.add(x);
List<T>.add(T elem)
Assert(elem is T)
checked entry
unchecked entry


not match
List<Object> strs = new List<String>();
strs.add(x);
List<T>.add(T elem)
Assert(elem is T)
checked entry
unchecked entry


can use static analysis or dynamic profiling
to determine / speculate invariance
List<String> strs = ...;
strs.add(x);
List<T>.add(T elem)
Assert(elem is T)
checked entry
unchecked entry


performance impact of Dart 2 mode
10-50% slower
<10% slower
<10% faster
10-50% faster
2-5x faster
5-10x faster
10-15x faster
0
5
10
15
20
# of benchmarks

architectural changes take forever

[choose the balance wisely]

parameterize your architecture

optimizations are more general than you think

[just need to tweak things a bit]

there are still interesting problems to solve

[code size for example]

if you have your own compiler use it more

[don't write assembly by hand]