

Ahead-of-time Compilation of JavaScript to WebAssembly via Static Type Inference

Chris Fallin, F5
VMIL 2026 (at SPLASH 2026)
Oct 8, 2026, Oakland, CA

Why JavaScript *in* WebAssembly?

- FaaS & network-dataplane (& embedded & ...) use-cases are *Wasm-first*
 - We cannot embed V8 or SpiderMonkey and have a first-class JS engine

Why JavaScript *in* WebAssembly?

- FaaS & network-dataplane (& embedded & ...) use-cases are *Wasm-first*
 - We cannot embed V8 or SpiderMonkey and have a first-class JS engine
- Yet customers want JavaScript (and have a lot of existing code)

Why JavaScript *in* WebAssembly?

- FaaS & network-dataplane (& embedded & ...) use-cases are *Wasm-first*
 - We cannot embed V8 or SpiderMonkey and have a first-class JS engine
- Yet customers want JavaScript (and have a lot of existing code)
- Why can't we use V8/SpiderMonkey?

Why not JIT?

- No minting of new code at runtime
 - Security — mprotect +X is heavily audited; CVE record of JS engines...
 - Latency — no optimizing compiler in the middle of the dataplane
 - Deterministic perf — no reliance on warmup to get to expected level

Why Wasm-first?

- Management + interface uniformity
 - A Wasm module (component) is self-contained, declares its interfaces, and supports every language that we need to support
 - Top-level language runtime is N x M integration task with dataplane
 - Bring-your-own-engine allows easier in-field upgrades — more control to user

Existing Approaches to JS in Wasm

- circa 2024: *Interpreter-in-a-box* (SpiderMonkey, QJS, ...)
 - Performance ~30x slower than native (10-100x range)
 - Substantially changes the *kind of workloads* that are even feasible

Existing Approaches to JS in Wasm

- circa 2024: *Interpreter-in-a-box* (SpiderMonkey, QJS, ...)
 - Performance ~30x slower than native (10-100x range)
 - Substantially changes the *kind of workloads* that are even feasible
- weval (partial eval) + Portable Baseline Interpreter in SpiderMonkey (PLDI'25)
 - Equivalent to native baseline compiler: AOT, dynamic types —> late binding
 - 1.5-3x speedups; still ~20x slower than native

Existing Approaches to JS in Wasm

- circa 2024: *Interpreter-in-a-box* (SpiderMonkey, QJS, ...)
 - Performance ~30x slower than native (10-100x range)
 - Substantially changes the *kind of workloads* that are even feasible
- weval (partial eval) + Portable Baseline Interpreter in SpiderMonkey (PLDI'25)
 - Equivalent to native baseline compiler: AOT, dynamic types —> late binding
 - 1.5-3x speedups; still ~20x slower than native
- *Why is it still so slow? What is the gap?*

Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

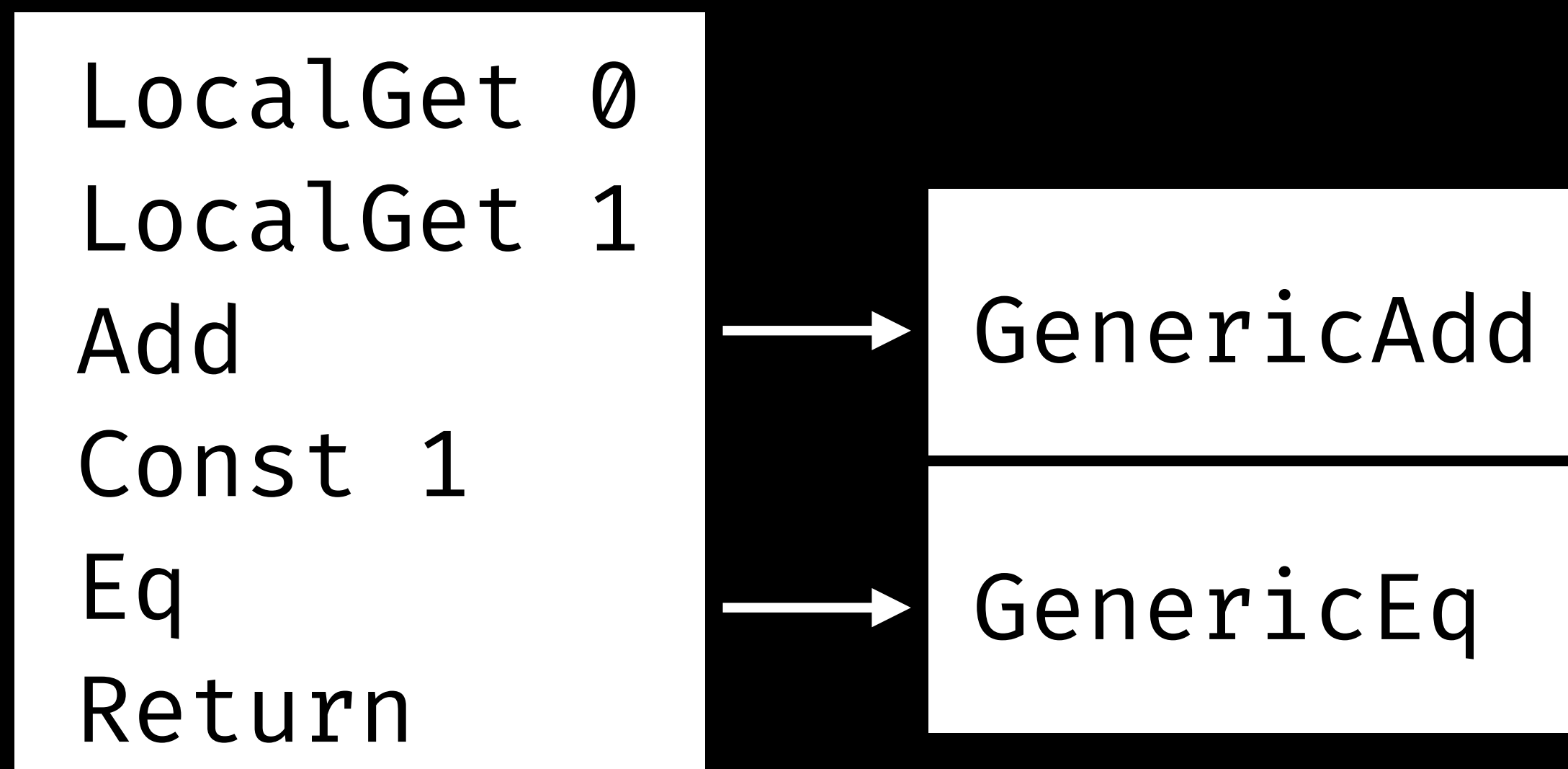
function(x, y) { return (x + y) == 1; }

```
LocalGet 0  
LocalGet 1  
Add  
Const 1  
Eq  
Return
```

Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

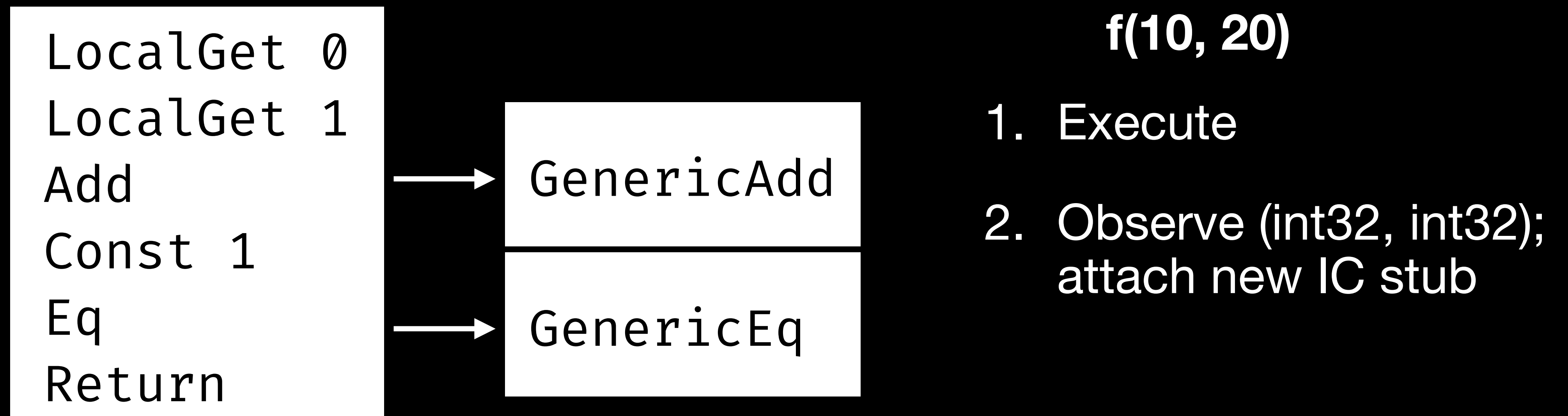
function(x, y) { return (x + y) == 1; }



Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

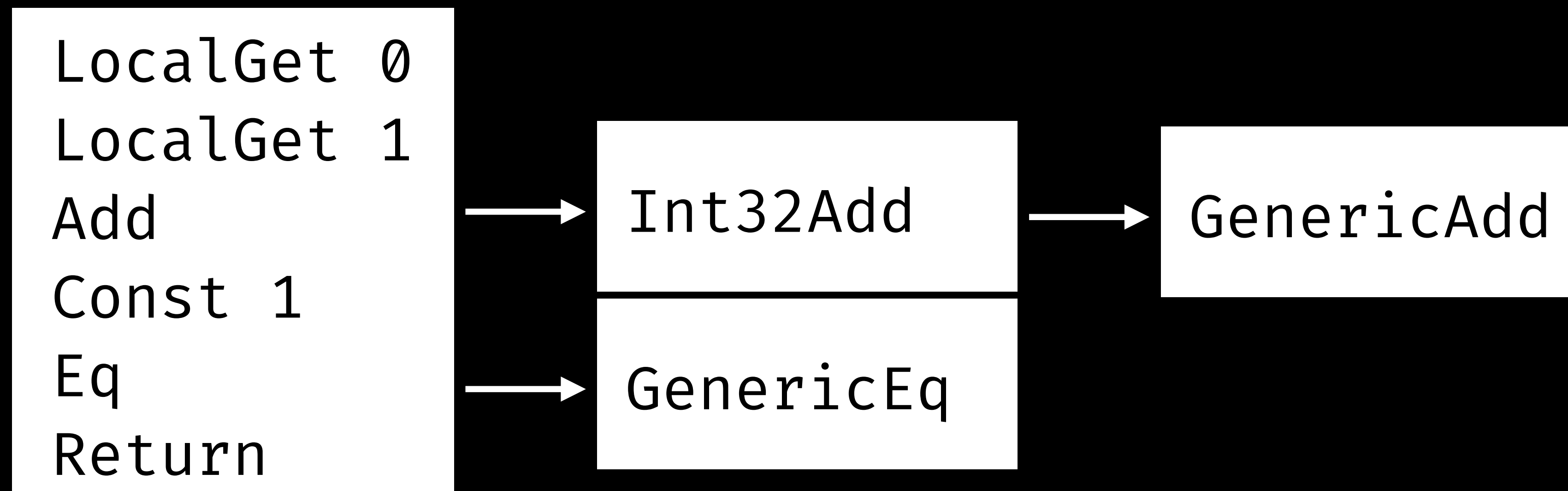
function(x, y) { return (x + y) == 1; }



Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

function(x, y) { return (x + y) == 1; }



Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

function(x) { return x.prop; }

LocalGet 0
PropGet 'x'
Return



GenericPropGet

Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

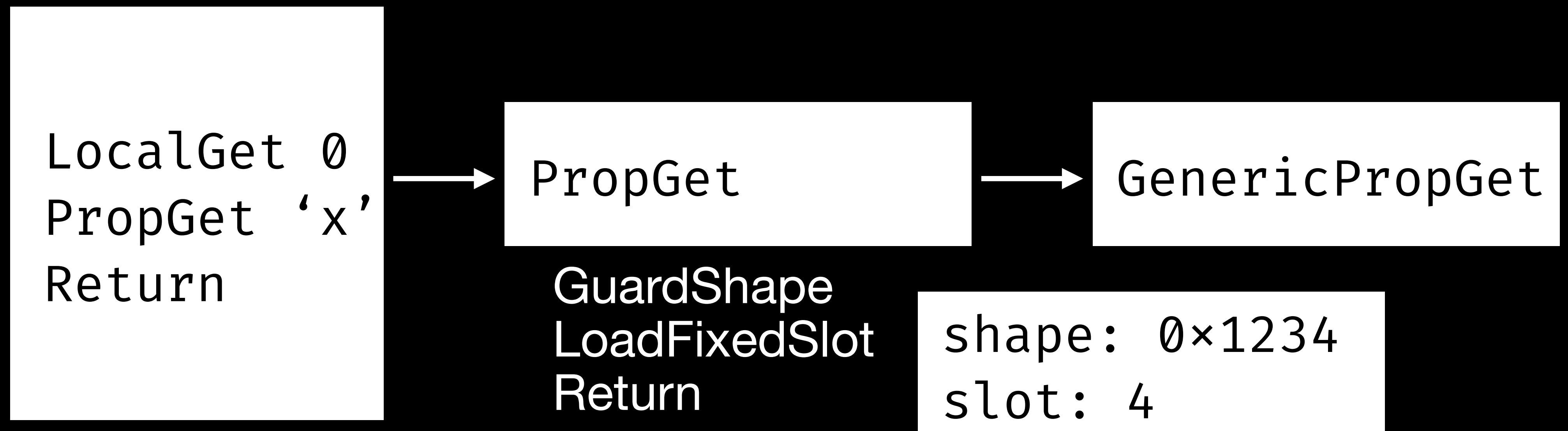
function(x) { return x.prop; }



Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...

```
function(x) { return x.prop; }
```



Recap: “Baseline-level” Compilation

- JS source compiles to JS bytecode 1-to-1
 - Operators with JS semantics: ad-hoc polymorphism, props+prototypes, ...
- This *permits AOT* (embed all common stub sequences in engine)

GuardInt32
GuardInt32
Int32Add
Return

GuardString
GuardString
StringConcat
Return

GuardShape
LoadFixedSlot
Return ...

What Can't Baseline Compilation Achieve?

```
function(x) { return x.a + x.b + 1; }
```

What Can't Baseline Compilation Achieve?

```
function(x) { return x.a + x.b + 1; }
```

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```

What Can't Baseline Compilation Achieve?

```
function(x) { return x.a + x.b + 1; }
```

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```



PropGet



...



PropGet



...



Int32Add



...

What Can't Baseline Compilation Achieve?

```
function(x) { return x.a + x.b + 1; }
```

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```

→ PropGet → ...

→ PropGet → ...

→ Int32Add → ...

```
GuardShape  
LoadFixedSlot  
GuardShape  
LoadFixedSlot
```

Const

```
GuardInt32  
GuardInt32  
Int32Add  
BoxInt32
```

```
GuardInt32  
GuardInt32  
Int32Add  
BoxInt32
```

What Can't Baseline Compilation Achieve?

Redundant guards

```
function(x) { return x.a + x.b + 1; }
```

GuardShape
LoadFixedSlot
GuardShape
LoadFixedSlot

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```

→ PropGet → ...

→ PropGet → ...

→ Int32Add → ...

Const

GuardInt32
GuardInt32
Int32Add
BoxInt32

GuardInt32
GuardInt32
Int32Add
BoxInt32

What Can't Baseline Compilation Achieve?

Redundant guards

```
function(x) { return x.a + x.b + 1; }
```

GuardShape
LoadFixedSlot
GuardShape
LoadFixedSlot

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```

PropGet

...

PropGet

...

Int32Add

...

Trivially-true guard

Const

GuardInt32
GuardInt32
Int32Add
BoxInt32

GuardInt32
GuardInt32
Int32Add
BoxInt32

What Can't Baseline Compilation Achieve?

Redundant guards

```
function(x) { return x.a + x.b + 1; }
```

GuardShape
LoadFixedSlot
GuardShape
LoadFixedSlot

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
Const 1  
Add  
Add  
Return
```

PropGet

...

PropGet

...

Int32Add

...

Trivially-true guard

Redundant boxing

Trivially-true guard

Const

GuardInt32
GuardInt32
Int32Add
BoxInt32

GuardInt32
GuardInt32
Int32Add
BoxInt32

What Can't Baseline Compilation Achieve?

1. Only a few guards actually check "external conditions"
 - x's shape
 - x.a's type
 - x.b's type

Redundant guards

+ x.b + 1; }

GuardShape

LoadFixedSlot

GuardShape

LoadFixedSlot

Const

trivially-true guard

GuardInt32

GuardInt32

Int32Add

redundant boxing

BoxInt32

trivially-true guard

GuardInt32

GuardInt32

Int32Add

BoxInt32

What Can't Baseline Compilation Achieve?

1. Only a few guards actually check
“external conditions”

- x's shape
- x.a's type
- x.b's type

The rest of the guards check conditions that
are always true, but established by
other operators

Redundant guards

+ x.b + 1; }

GuardShape

LoadFixedSlot

GuardShape

LoadFixedSlot

Const

...

trivially-true guard

GuardInt32

GuardInt32

Int32Add

redundant boxing

BoxInt32

...

trivially-true guard

GuardInt32

GuardInt32

Int32Add

BoxInt32

What Can't Baseline Compilation Achieve?

1. Only a few guards actually check “external conditions”

- x's shape
- x.a's type
- x.b's type

The rest of the guards check conditions that are always true, but established by *other operators*

2. (Not shown) values flow through an *actual operand stack in memory*

Redundant guards

+ x.b + 1; }

GuardShape

LoadFixedSlot

GuardShape

LoadFixedSlot

Const

...

trivially-true guard

GuardInt32

GuardInt32

Int32Add

redundant boxing

BoxInt32

...

trivially-true guard

GuardInt32

GuardInt32

Int32Add

BoxInt32

What Can't Baseline Compilation Achieve?

Baseline *cannot optimize across ICs*

Redundant guards

.b + 1; }

GuardShape
LoadFixedSlot
GuardShape
LoadFixedSlot

Const

-true guard

ant boxing

-true guard

GuardInt32
GuardInt32
Int32Add
BoxInt32

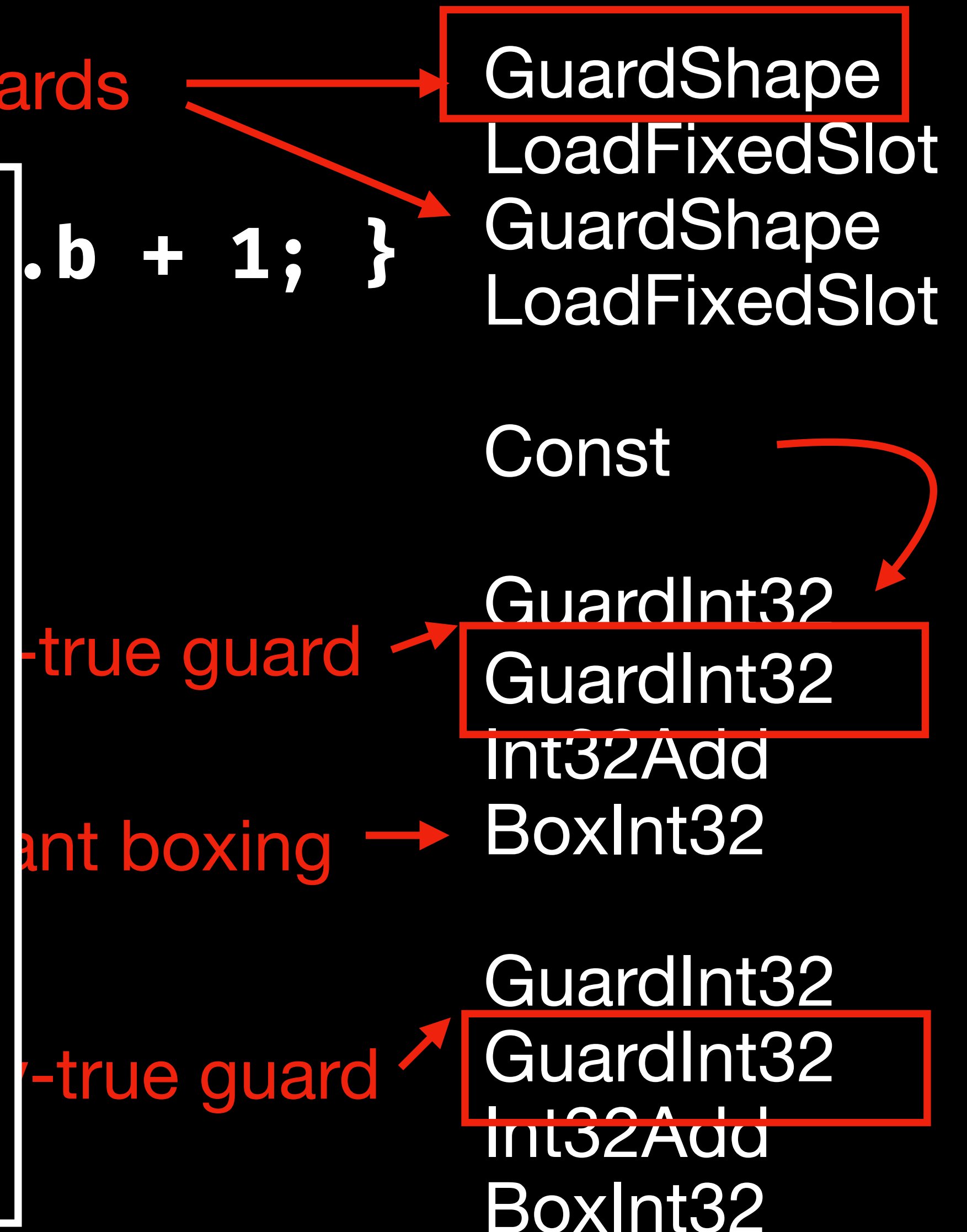
GuardInt32
GuardInt32
Int32Add
BoxInt32

What Can't Baseline Compilation Achieve?

Baseline *cannot optimize across ICs*

- Interface at JS-operator boundary is fully-boxed tagged values + strict operand-stack virtual machine model

Redundant guards



What Can't Baseline Compilation Achieve?

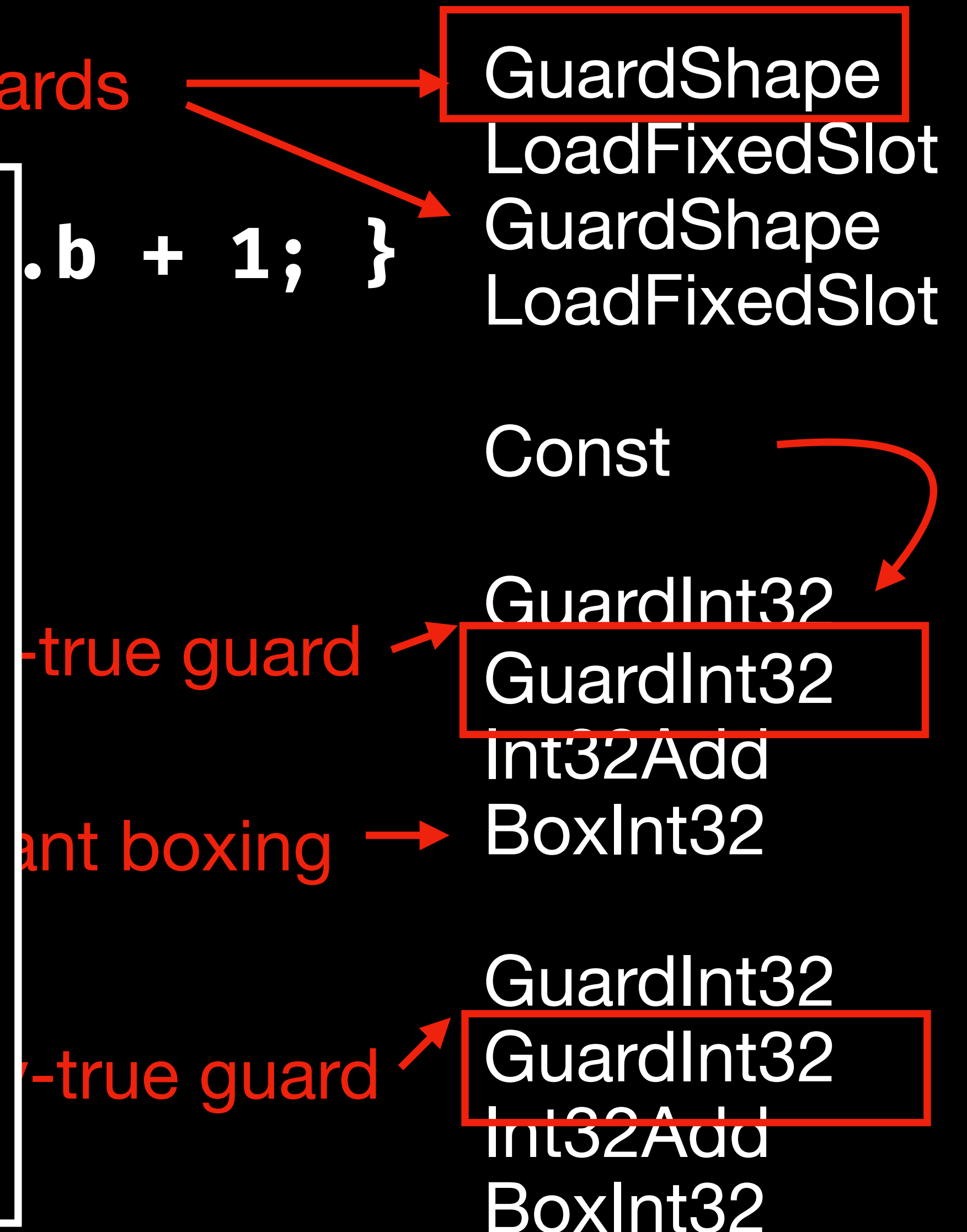
Baseline *cannot optimize across ICs*

- Interface at JS-operator boundary is fully-boxed tagged values + strict operand-stack virtual machine model

Baseline *achieves late binding* (permitting compilation without types!) by enforcing this interface

... but it gives up function-scope type specialization!

Redundant guards



How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
...
```



```
PropGet
```



```
PropGet
```

```
LocalGet 0  
GuardShape  
LoadFixedSlot  
LocalGet 0  
GuardShape  
LoadFixedSlot  
...
```

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code

```
LocalGet 0  
PropGet 'x'  
LocalGet 0  
PropGet 'y'  
...
```



```
PropGet
```



```
PropGet
```

```
LocalGet 0  
GuardShape  
LoadFixedSlot  
LocalGet 0  
GuardShape  
LoadFixedSlot  
...
```

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code

```
function(x) { return x.a + x.b; }
```

```
GuardShape  
LoadFixedSlot  
LoadFixedSlot  
GuardInt32  
GuardInt32  
Int32Add  
Return
```

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code
- *Note:* IC inlining + guard hoisting is WarpMonkey (IonMonkey)'s approach; other JITs may use explicit type feedback vectors. Principle is the same.

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- So: observe types during warmup, then generate type-specialized code
- *Note:* IC inlining + guard hoisting is WarpMonkey (IonMonkey)'s approach; other JITs may use explicit type feedback vectors. Principle is the same.
- Some work on computing downstream types from observed + guarded field types, with tripwires on changes to those fields (Hackett/Guo PLDI'12). Still depends on runtime observations.

How do Native JS Engines Achieve Speedups?

- The JIT hypothesis: types are mostly stable once observed (*or we can detect+avoid megamorphic cases*)
- *What if we can't profile the code?*

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)
- Have a control-plane do it automatically? (profile service-wide + recompile)

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)
- Have a control-plane do it automatically? (profile service-wide + recompile)
 - Conway's Law: "we are a toolchain team and that's a control-plane issue"

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)
- Have a control-plane do it automatically? (profile service-wide + recompile)
 - Conway's Law: "we are a toolchain team and that's a control-plane issue"
 - Also has the *warmup problem* on newly-deployed code; need a traffic-mirroring approach

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)
- Have a control-plane do it automatically? (profile service-wide + recompile)
 - Conway's Law: "we are a toolchain team and that's a control-plane issue"
 - Also has the *warmup problem* on newly-deployed code; need a traffic-mirroring approach
- Can we *stare at the code and infer the types*?

Why Can't We Profile the Code?

- We may not be able to run the prod workload in dev environment
 - (lack of APIs, endpoints, prod datasets, ...)
- Have a control-plane do it automatically? (profile service-wide + recompile)
 - Conway's Law: "we are a toolchain team and that's a control-plane issue"
 - Also has the *warmup problem* on newly-deployed code; need a traffic-mirroring approach
- Can we *stare at the code** and infer the types?

* *do a static analysis*

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

Global lookup (implicit globalThis, ...)

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

Property access (through prototype)

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```



Devirtualization

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```

Property access:
- slot lookup
- type from ctor

Interprocedural Type Inference?

- Consider:

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

Finally can see
i32 + i32 types

```
function main() {  
    let v1 = new Vector(1, 2, 3);  
    let v2 = new Vector(4, 5, 6);  
    let v3 = v1.add(v2);  
}
```

Interprocedural Type Inference?

- So we need:
 - Global points-to analysis tracking types per field on heap abstractions
 - Tracking of functions as concrete heap abstractions + devirtualization
 - Modeling of prototype-object chains
 - Probably some context-sensitivity
- Just to get a fast integer add!

Has This Been Done Before?

- IBM WALA
- Type Analysis for JavaScript (TAJS)

Has This Been Done Before?

- IBM WALA
- Type Analysis for JavaScript (TAJS)
- Key ideas:
 - Points-to analysis (heap abstractions) in fixpoint loop with callgraph analysis
 - Abstract interpretation of each JS operator

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

.x

{abs2}

.y

{int32, string}

Has This Been Done Before?

- IBM WALA
- Type Analysis for JavaScript (TAJS)
- Key ideas:
 - Points-to analysis (heap abstractions) in fixpoint loop with callgraph analysis
 - Abstract interpretation of each JS operator
 - Conservative: meet-over-prototypes, all fallback paths (.toString() / .valueOf()), ...

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

.x

{abs2}

.y

{int32, string}

Has This Been Done Before?

- IBM WALA
- Type Analysis for JavaScript (TAJS)
- Key ideas:
 - Points-to analysis (heap abstractions) in fixpoint loop with callgraph analysis
 - Abstract interpretation of each JS operator
 - Conservative: meet-over-prototypes, all fallback paths (.toString() / .valueOf()), ...
- *Extraordinarily* unscalable: targeting static-analysis tools; minutes...hours runtimes
 - TAJs: flow-sensitive *image of entire heap at every point (!!!)* + *callsite sensitivity*

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

.x
.y

{abs2}
{int32, string}

Summary: What we Have

Approach	Runtime Perf	Needs Warmup?	Scalable to Large Programs?
Native JIT	Yes	Yes	Yes

Summary: What we Have

Approach	Runtime Perf	Needs Warmup?	Scalable to Large Programs?
Native JIT	Yes	Yes	Yes
Baseline AOT	~Mid	No	Yes

Summary: What we Have

Approach	Runtime Perf	Needs Warmup?	Scalable to Large Programs?
Native JIT	Yes	Yes	Yes
Baseline AOT	~Mid	No	Yes
Whole-program AOT Analysis	Yes (brittle)	No	No!!!

Summary: What we Have

Approach	Runtime Perf	Needs Warmup?	Scalable to Large Programs?
Native JIT	Yes	Yes	Yes
Baseline AOT	~Mid	No	Yes
Whole-program AOT Analysis	Yes (brittle)	No	No!!!
???	Yes	No	Yes

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes
- It works!

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes
- It works! One one benchmark!

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes
- It works! One one benchmark!
 - Richards (Octane): interp ~340; weval ~1k; SPARTA ~17k; native JIT ~28k

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes
- It works! One one benchmark!
 - Richards (Octane): interp ~340; weval ~1k; SPARTA ~17k; native JIT ~28k
- Mandreel (Octane): ~350kLoC JS; OOMs on 128GiB workstation

An Earnest Attempt: Sound (but Partial) Analysis

- SPARTA: Sound, Partial Analysis for the Recovery of Type Abstractions
- Basically the above, minus flow-sensitivity and callsite-sensitivity
 - + Polymorphic callsites with receiver+arg correlation
 - + Definite-defined analysis to restrict meet-over-prototypes
- It works! One one benchmark!
 - Richards (Octane): interp ~340; weval ~1k; SPARTA ~17k; native JIT ~28k
- Mandreel (Octane): ~350kLoC JS; OOMs on 128GiB workstation
- box2d (Octane): too-clever object framework; “leakage” of all objects to all typesets

Why does Static Analysis “Fail”?

- Abstract interpretation can derive many facts about a program
- But *Rice’s theorem* mirrors undecidability into the analysis domain:
 - ***we can never resolve everything***
- And JS programmers are “*extremely clever*”

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”:
 - Arbitrary computation to create prototype-object classes
 - box2d:

```
function K { ... };  
function inherit(y,w) {  
    K.prototype = w.prototype; y.prototype = new K;  
y.prototype.constructor = y; }
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”:
 - Arbitrary computation to create prototype-object classes
 - box2d:

```
function K { ... };  
function inherit(y,w) {  
    K.prototype = w.prototype; y.prototype = new K;  
y.prototype.constructor = y; }                               ?!?!  
- need flow sensitivity  
  to abstract-interpret  
  and recover protos
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”
- And JS is extremely weird:

```
let x = { a: A, b: B, c: C };  
let y = x[index];    // ← points-to set: { A, B, C }
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”
- And JS is extremely weird:

```
let x = { a: A, b: B, c: C };  
let y = x[index];      // ← pts: { A, B, C, __proto__ }
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”
- And JS is extremely weird:

```
let x = { a: A, b: B, c: C };  
let y = x[“__proto__”];
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”
- And JS is extremely weird:

```
let x = { a: A, b: B, c: C };  
let y = x[“__proto__” + “to__”];
```

Why does Static Analysis “Fail”?

- JS programmers are “*extremely clever*”
- And JS is extremely weird:

```
let x = { a: A, b: B, c: C };  
let y = x[“__proto__” + “to__”];
```

?!?!

Do we need a
“definitely not __proto__”
analysis on strings?

Insight #1: Snapshots + Abstract Interpretation

- Abstract interpretation can derive many facts about a program
- But *Rice's theorem* mirrors undecidability into the analysis domain:
 - ***we can never resolve everything***

Insight #1: Snapshots + Abstract Interpretation

- Abstract interpretation can derive many facts about a program
- But *Rice's theorem* mirrors undecidability into the analysis domain:
 - ***we can never resolve everything***
- Instead, seed a points-to with *concrete heap* + analyze from entry-point

Insight #1: Snapshots + Abstract Interpretation

- Abstract interpretation can derive many facts about a program
- But *Rice's theorem* mirrors undecidability into the analysis domain:
 - ***we can never resolve everything***
- Instead, seed a points-to with *concrete heap* + analyze from entry-point
i.e., do **concrete interp** on init phase and **abstract interp** on run phase

Insight #1: Snapshots + Abstract Interpretation

- Abstract interpretation can derive many facts about a program
- But *Rice's theorem* mirrors undecidability into the analysis domain:
 - ***we can never resolve everything***
- Instead, seed a points-to with *concrete heap* + analyze from entry-point

We are pushing the “we can't profile” boundary:
- *we can't run the main entry point (e.g. HTTP handler)*
- *but we **can** run module-load, handler registration, ...*

*Many meta-programming shenanigans happen here
and we can “see through them” now*

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- We are working too hard to discover that

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

means “method add on class Vector”

(by soundly tracking prototypes, devirtualizing calls, passing the `this` receiver type interprocedurally, ...)

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- We are working too hard to discover that

```
Vector.prototype.add = function(other) {  
    return new Vector(this.x + other.x, ...);  
};
```

means “method add on class Vector”

(by soundly tracking prototypes, devirtualizing calls, passing the `this` receiver type interprocedurally, ...)

- Let’s just infer what the human does: “Vector is a class with an add method”

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- The program *could*:
 - monkeypatch all of the core types
 - use prototypes cleverly
 - add proxy hooks
 - overflow every i32 arith op

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- The program *could*:
 - monkeypatch all of the core types
 - use prototypes cleverly
 - add proxy hooks
 - overflow every i32 arith op
 - autonomously lobby the EcmaScript committee for even more weird behavior

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- The program *could*:
 - monkeypatch all of the core types
 - use prototypes cleverly
 - add proxy hooks
 - overflow every i32 arith op
 - ~~autonomously lobby the EcmaScript committee for even more weird behavior~~

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- The program *could*:
 - monkeypatch all of the core types
 - use prototypes cleverly
 - add proxy hooks
 - overflow every i32 arith op
 - ~~autonomously lobby the EcmaScript committee for even more weird behavior~~
- **But it probably won't do these things (or not after initialization)**

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- Instead, let's analyze “likely types”
- And then catch the unsound cases via guards at runtime

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- Instead, let’s analyze “likely types”
- And then catch the unsound cases via guards at runtime
- This is an interesting new design-space!
 - Nontrivial tradeoff: decide where to draw the “fastpath boundary”
 - Include more possibilities—> more code runs fastpath, but fastpath is slower
 - We still need a “guard soundness” strategy for each optimistic assumption

Insight #2: Unsound (Optimistic) Analysis

aka “the guards will keep us safe”

- What do we “ignore”? (*incomplete list*)
 - int32 overflow, i53 overflow, negative zero, ...
 - drop property writes when field name (f in $x[f] = a$) or receiver is any/unknown
 - assume globals are static-type after init snapshot
 - ignore undef/null in a lot of places; ignore array holes
 - assume that functions on prototype objects take that proto as **this** type
 - assume builtins, functions, methods are *not* dynamically rebound/monkeypatched

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

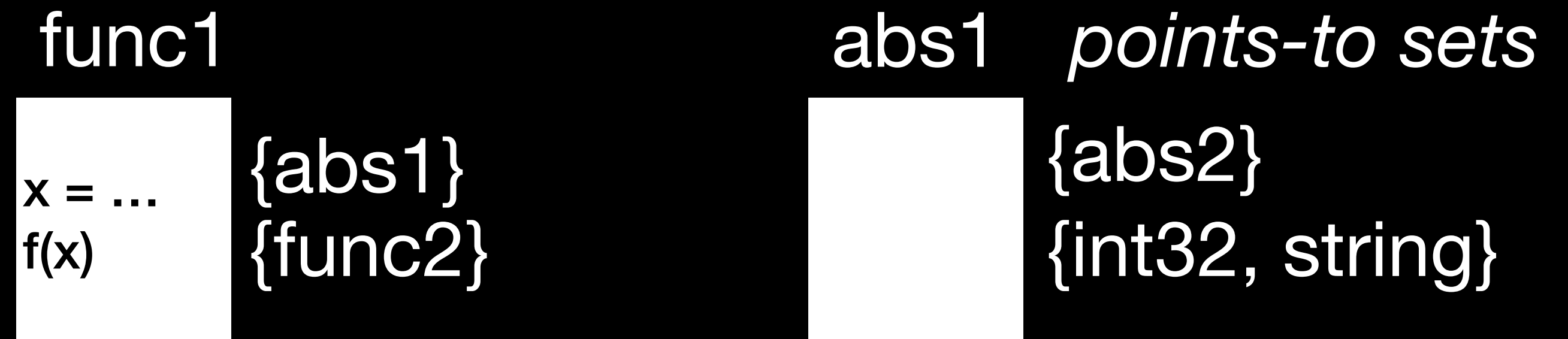
{abs1}
{func2}

abs1 *points-to sets*

{abs2}
{int32, string}

- We need points-to analysis to understand dataflow through the heap
 - without knowledge of object fields, type inference is *very* local + limited

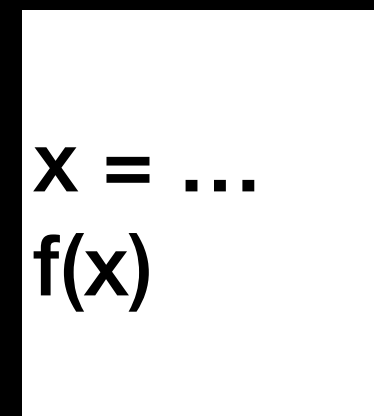
Insight #3: Build a *Fast* Points-to + Callgraph Analysis



- We need points-to analysis to understand dataflow through the heap
 - without knowledge of object fields, type inference is *very* local + limited
- Mutual fixpoint with callgraph analysis:
 - We need the callgraph to understand the *whole-program* flow of heap abstractions
 - The callgraph needs heap abstractions to understand how first-class functions reach callsites

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1



{abs1}
{func2}

abs1 *points-to sets*



{abs2}
{int32, string}

- The standard in static analysis is *Andersen points-to*
 - Keep a points-to set (or type set) for every storage cell
 - “Inclusion constraints”: $a := b \Rightarrow \text{pts}(b) \subseteq \text{pts}(a)$

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

{abs2}
{int32, string}

- The standard in static analysis is *Andersen points-to*
 - Keep a points-to set (or type set) for every storage cell
 - “Inclusion constraints”: $a := b \Rightarrow \text{pts}(b) \subseteq \text{pts}(a)$
- *Extremely* slow; ~cubic in program size in worst case; huge memory footprint

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

{abs2}
{int32, string}

- An alternative is *Steensgaard* points-to
 - Keep a global union-find data structure for all heap abstractions
 - “Union constraints”: $a := b \Rightarrow \text{union}(a, b)$

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

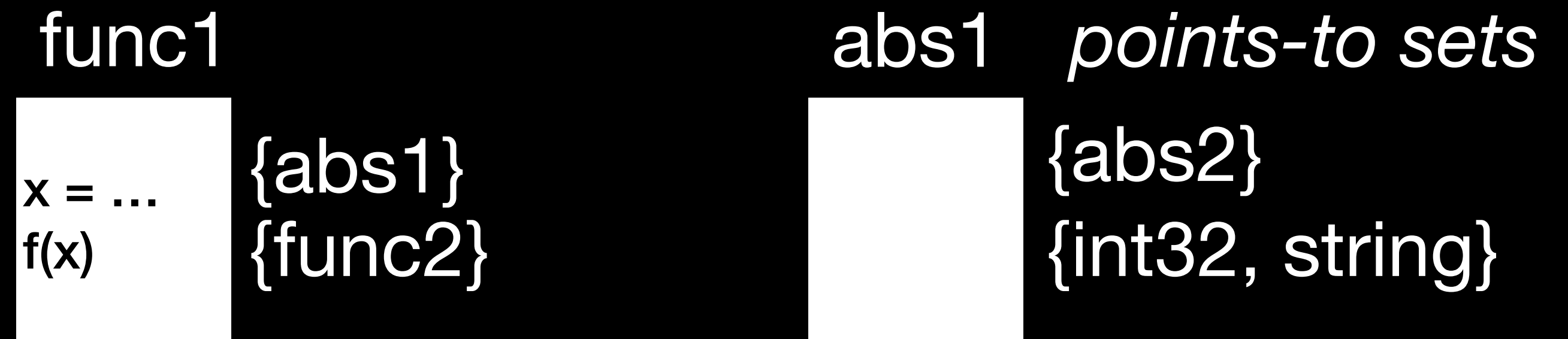
{abs1}
{func2}

abs1 *points-to sets*

{abs2}
{int32, string}

- An alternative is *Steensgaard* points-to
 - Keep a global union-find data structure for all heap abstractions
 - “Union constraints”: $a := b \Rightarrow \text{union}(a, b)$
- *Very fast*: ~linear in program size

Insight #3: Build a *Fast* Points-to + Callgraph Analysis



- An alternative is *Steensgaard* points-to
 - Keep a global union-find data structure for all heap abstractions
 - “Union constraints”: $a := b \Rightarrow \text{union}(a, b)$
- *Very fast*: ~linear in program size
- *Horribly inaccurate*: Many real programs congeal into a few giant blobs

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

{abs1}
{func2}

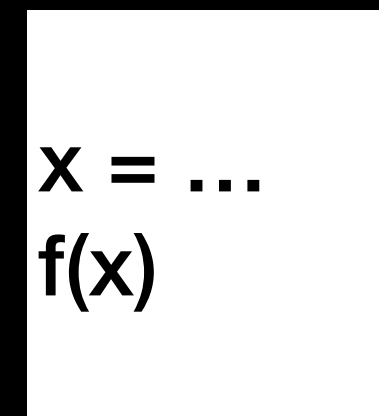
abs1 *points-to sets*

{abs2}
{int32, string}

- Why not best of both?
 - We like Andersen for its inclusion semantics: “aware of dataflow direction”
 - We like Steensgaard for its “size cap”: one cell cannot hold arbitrary set

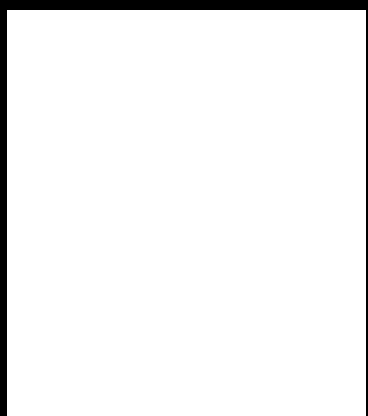
Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1



{abs1}
{func2}

abs1 *points-to sets*



{abs2}
{int32, string}

- Why not best of both?
 - We like Andersen for its inclusion semantics: “aware of dataflow direction”
 - We like Steensgaard for its “size cap”: one cell cannot hold arbitrary set
- We don’t do much with a multi-entry points-to set: e.g. no polymorphic dispatch. Why not coarsen?

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

{abs2}
{int32, string}

- Abstract domain: “one allocation site” < “union-find clump” < “any object”
 - i.e., we still do *inclusion*, but if a set grows, we instead union the elements
 - This means that the lattice meet *mutates the domain* (!!)
 - Nevertheless it seems to work!

Insight #3: Build a *Fast* Points-to + Callgraph Analysis

func1

x = ...
f(x)

{abs1}
{func2}

abs1 *points-to sets*

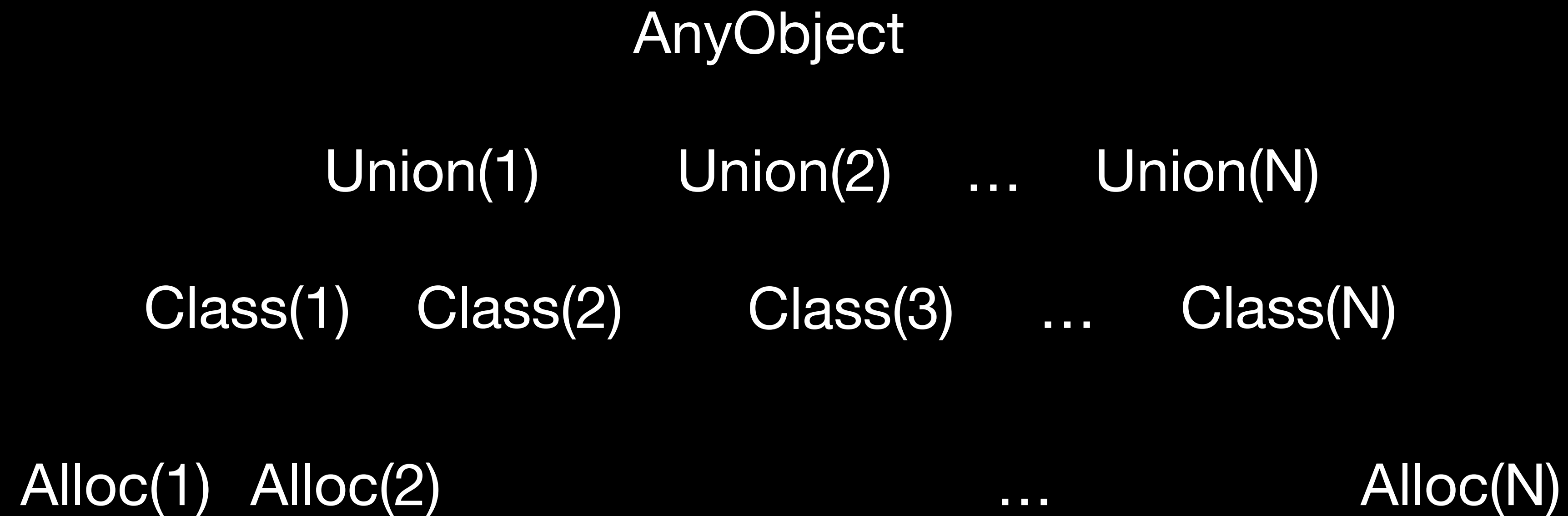
{abs2}
{int32, string}

- “one allocation site” < “class” < “union-find clump” < “any object”



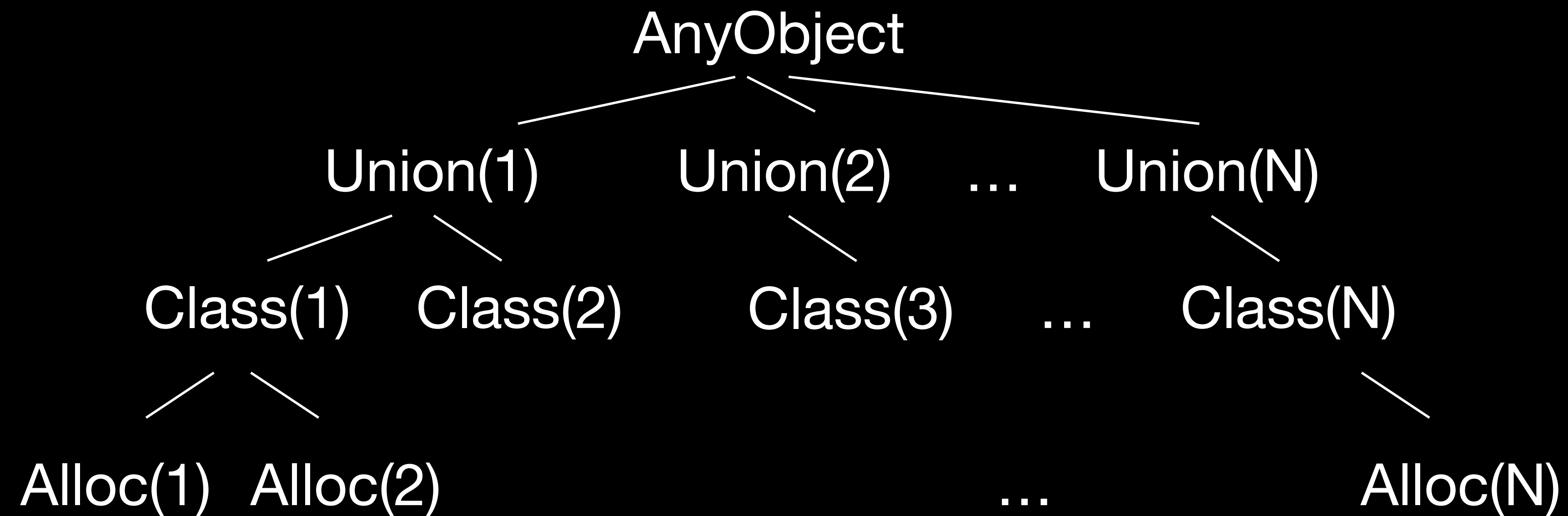
prototype object

Insight #3: Build a *Fast* Points-to + Callgraph Analysis



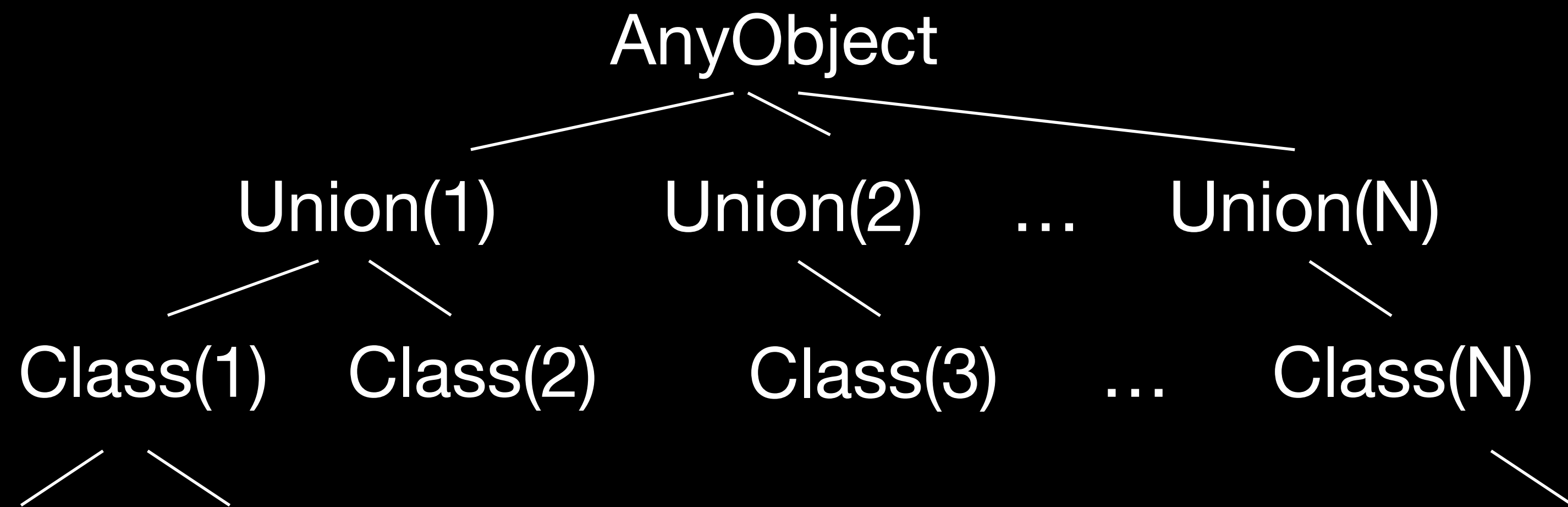
Three-level lattice

Insight #3: Build a *Fast* Points-to + Callgraph Analysis



Three-level lattice
... in fact, a tree!

Insight #3: Build a *Fast* Points-to + Callgraph Analysis



Together, these form the “likely-types analysis”

- Decorate the JS bytecode with types
- Then compile using these types, as if they were IC-derived type observations

Codegen: A Tale of Two Tiers

- Classical approach (IonMonkey, V8, ...):
 - *Optimistic* highest tier with fully type-specialized code
 - *Generic* lower (but still compiled) “baseline” tier
 - Possibility of de-opt at any guard
 - Possibility of “tier up” at designated points (usually func + loop headers)

Codegen: A Tale of Two Tiers

- Classical approach (IonMonkey, V8, ...):
 - *Optimistic* highest tier with fully type-specialized code
 - *Generic* lower (but still compiled) “baseline” tier
 - Possibility of de-opt at any guard
 - Possibility of “tier up” at designated points (usually func + loop headers)
- This turns out to be a good idea!

Codegen: A Tale of Two Tiers

- Classical approach (IonMonkey, V8, ...):
 - *Optimistic* highest tier with fully type-specialized code
 - *Generic* lower (but still compiled) “baseline” tier
 - Possibility of de-opt at any guard
 - Possibility of “tier up” at designated points (usually func + loop headers)
- This turns out to be a good idea!
 - Initially tried *basic block versioning* but very hard to avoid blowup

Codegen: A Tale of Two Tiers

- MIR (mid-IR): a *typed IR*; lower JS bytecode + likely-types into MIR
 - Guards are explicit (conditional branches); exits are explicit
 - Representation decisions (e.g. unboxed i32) are explicit
 - Field accesses are *still* at JS-semantics level; critical for analyzing effects

Codegen: A Tale of Two Tiers

- MIR (mid-IR): a *typed IR*; lower JS bytecode + likely-types into MIR
 - Type system can represent:
 - Specific object classes/layouts, and partial initialization
 - Known specific function objects
 - Known integer ranges

Codegen: A Tale of Two Tiers

```
TaskControlBlock.prototype.isHeldOrSuspended = function () {
    return (this.state & STATE_HELD) ≠ 0 || (this.state = STATE_SUSPENDED);
};

func @s194 (formals=0, locals=1) {
b0(v0: obj{Function(s194)}, v1: val):                ; v1 = this, untyped
    guard.tags v1 {object} → ok b5(v8), fail b4
b5(v8: val{object}):
    v53 = unbox.obj v8
    guard.layout v53 L5 types slots → ok b10(v13), fail b9 ; predicted class
b10(v13: obj{L5 types slots}):
    v51 = load_slot v13 state                ; val{int32}; load from constant offset
    v19 = unbox.i32 v51                      ; no guard; serves both reads
    check.fuse F0 → ok b14, fail b15         ; global STATE_HELD still = 4
b14:
    v23 = i32.and v19, 4
    v25 = i32.cmp.ne v23, 0
    [ ... check ≠ 2, too; box bool, return ... ]
}
```

Codegen: A Tale of Two Tiers

- MIR (mid-IR): a *typed IR*; lower JS bytecode + likely-types into MIR
- Baseline: a direct JS bytecode to Wasm translation
 - Most ops are translated to *runtime helper calls*; priority is small code

Codegen: A Tale of Two Tiers

- MIR (mid-IR): a *typed IR*; lower JS bytecode + likely-types into MIR
- We can optimize MIR extensively:
 - Guard hoisting, redundant-guard elimination, infallible guard removal
 - Loop-invariant code motion (LICM)
 - Escape analysis, SRoA (replacing objects with scalars), lazy remat on de-opt
 - Box/unbox-pair removal
 - Branch folding, constant propagation, ...

Insight #4: Read vs. Write Guard Cost

- How do we actually consume *likely types on obj fields*? Where do we guard?
 - Naively, every field-load must guard: we don't know anything *for sure*
 - Likely field type still helped us decide what specialized code to generate
 - But this is expensive!

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - Let's add a header word

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - Let's add* a header word

* actually, steal: on 32-bit systems, SpiderMonkey has a padding word we can use

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - Let's add* a header word
 - Let's allocate an ID to every entry in the type-lattice
 - *(remember it's a tree! we can do subtype tests with range checks!)*

* actually, steal: on 32-bit systems, SpiderMonkey has a padding word we can use

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - Let's add* a header word
 - Let's allocate an ID to every entry in the type-lattice
 - *(remember it's a tree! we can do subtype tests with range checks!)*
 - Let's then “stamp” an object with its alloc-site ID, and clear the stamp if a *write* invalidates the expected type for that field

* actually, steal: on 32-bit systems, SpiderMonkey has a padding word we can use

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - “This object belongs to likely-class C and its fields satisfy those types”
 - Barriers on every store check the type, and clear a header bit on failure

Insight #4: Read vs. Write Guard Cost

- Insight: we can *dynamically track whether an object has its expected types*
 - “This object belongs to likely-class C and its fields satisfy those types”
 - Barriers on every store check the type, and clear a header bit on failure
 - Now we can guard once per object-ref that we see; all reads are free

Insight #5: Fix/Align the Object Layouts

- Background: in SpiderMonkey, “slots” are allocated in order they’re added

Insight #5: Fix/Align the Object Layouts

- Background: in SpiderMonkey, “slots” are allocated in order they’re added
 - Ideally, we compile PropGet/PropSet to fixed-offset loads/stores for perf
 - We can try to “predict” (at least a prefix) by analyzing constructor bodies

Insight #5: Fix/Align the Object Layouts

- Background: in SpiderMonkey, “slots” are allocated in order they’re added
 - Ideally, we compile PropGet/PropSet to fixed-offset loads/stores for perf
 - We can try to “predict” (at least a prefix) by analyzing constructor bodies
- This is sometimes foiled by:
 - Varying/conditional execution order (real JIT can adapt or go megamorphic)

Insight #5: Fix/Align the Object Layouts

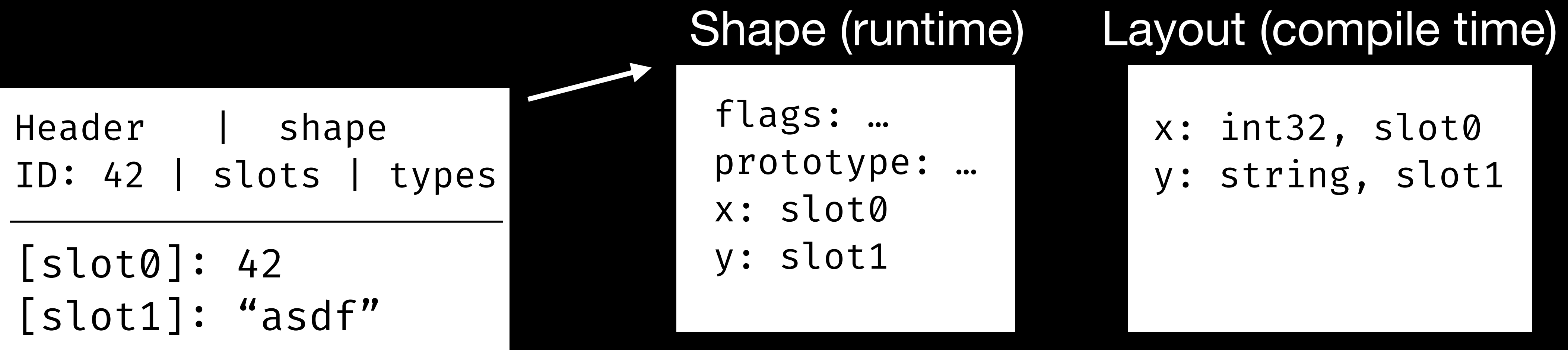
- Background: in SpiderMonkey, “slots” are allocated in order they’re added
 - Ideally, we compile PropGet/PropSet to fixed-offset loads/stores for perf
 - We can try to “predict” (at least a prefix) by analyzing constructor bodies
- This is sometimes foiled by:
 - Varying/conditional execution order (real JIT can adapt or go megamorphic)
 - Imperfect static analysis (we can only get a partial answer to “what is the full trace of execution”! Rice’s Theorem again!)

Insight #5: Fix/Align the Object Layouts

- *Insight*: we can allocate slots in any order we choose (if we tweak the runtime)
 - For a given class ID, *fix the property -> slot mapping*
 - (Shape can still vary, if some properties are absent!)
 - For a *subtree* of class IDs, *align the common prefixes*
 - Analogous to, e.g., C++ or Java super/subclass layout

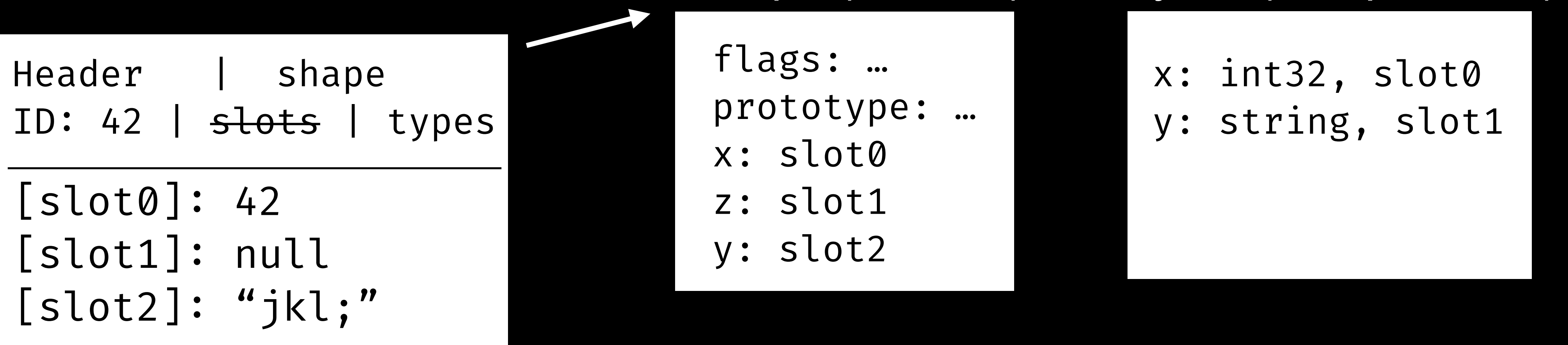
Insight #5: Fix/Align the Object Layouts

- Objects have *two* “likely” prediction dimensions: field types, and field slots
 - Each can be invalidated separately



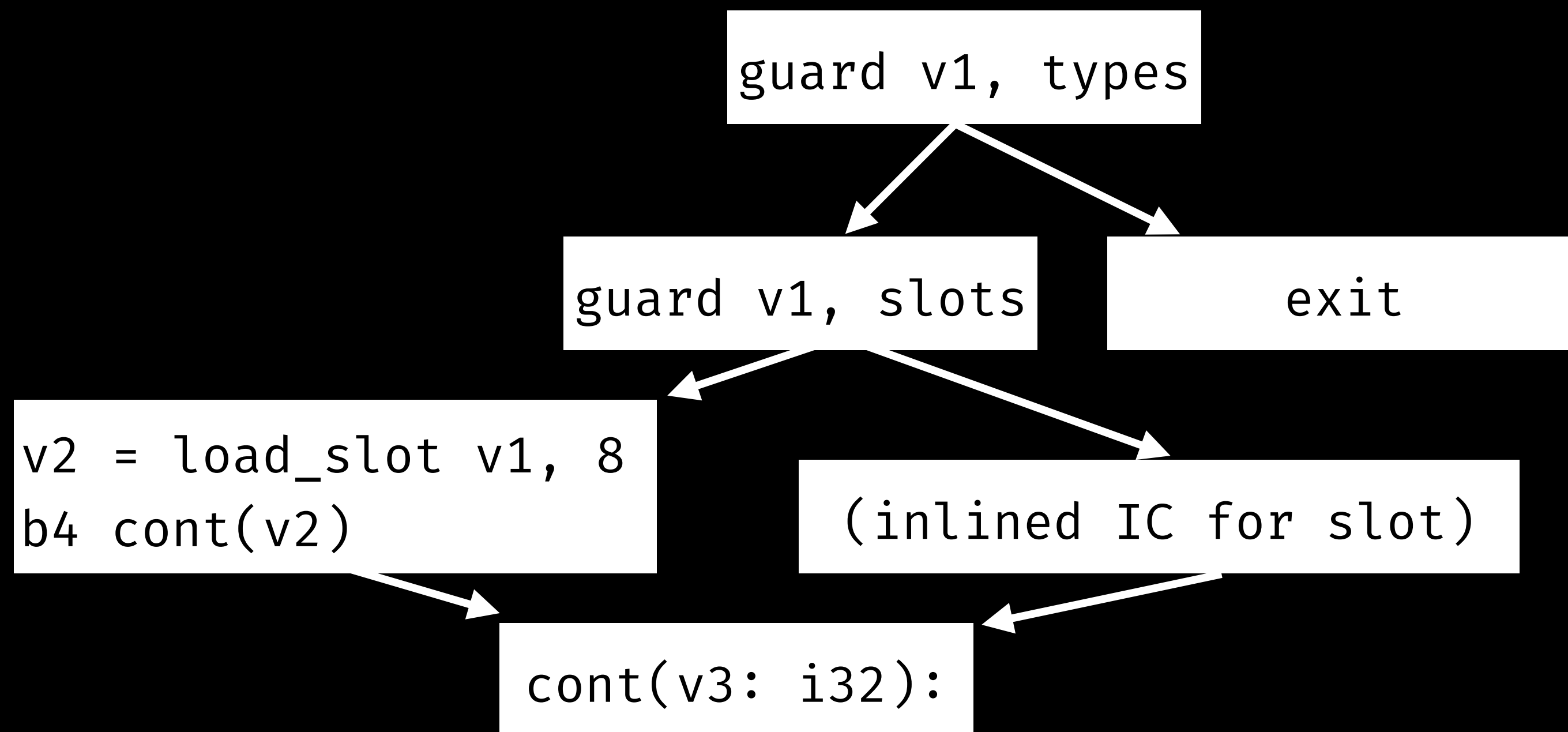
Insight #5: Fix/Align the Object Layouts

- Objects have *two* “likely” prediction dimensions: field types, and field slots
 - Each can be invalidated separately



Insight #5: Fix/Align the Object Layouts

- Objects have *two* “likely” prediction dimensions: field types, and field slots
 - Invalidated slots can stay in fastpath but with mini IC sites for slot lookup



Codegen: Type Contexts and Two-Track Approach

- Insidious issue: *irreducible control flow* in the presence of side-exits/side-entrances
 - Consider e.g. de-opting to generic path halfway through loop
 - Generic copy of function has a side-entrance at *almost every opcode across all loop nests*
 - Performance matters (cannot do label-based relooper): we want “onramps” back to optimized path too

Codegen: Type Contexts and Two-Track Approach

- Insidious issue: *irreducible control flow* in the presence of side-exits/side-entrances
- Initially: waffle's *reducify*, a dual to *relooper*: code-duplicate tails instead

Codegen: Type Contexts and Two-Track Approach

- Insidious issue: *irreducible control flow* in the presence of side-exits/side-entrances
- Initially: waffle's *reducify*, a dual to *relooper*: code-duplicate tails instead
- This blows up; instead, make “loop header tokens” part of type context and carefully arrange a “side-entrance” path alongside each “steady-state” path for each loop

Codegen: Type Contexts and Two-Track Approach

- Insidious issue: *irreducible control flow* in the presence of side-exits/side-entrances
- Initially: waffle's *reducify*, a dual to *relooper*: code-duplicate tails instead
- This blows up; instead, make “loop header tokens” part of type context and carefully arrange a “side-entrance” path alongside each “steady-state” path for each loop
- We need `multiloop` (Wasm proposal: fully general CFG in a region)
 - Prototype: up to 27% smaller code, up to 18% perf improvement

NightMonkey

- In the SpiderMonkey tradition: JaegerMonkey, IonMonkey, WarpMonkey, ...

NightMonkey

- In the SpiderMonkey tradition: JaegerMonkey, IonMonkey, WarpMonkey, ...
- NIGHT: Nonlocal Inference with Guiding Heuristics for Types

NightMonkey

- In the SpiderMonkey tradition: JaegerMonkey, IonMonkey, WalrusMonkey
- NIGHT: Nonlocal Inference with Guiding Heuristics for Types



NightMonkey

- In the SpiderMonkey tradition: JaegerMonkey, IonMonkey, WalrusMonkey
- NIGHT: Nonlocal Inference with Guiding Heuristics for Types



Night monkeys, also known as **owl monkeys** or **douroucoulis**^[2] ([/duːruːˈkuːlɪz/](#)), are nocturnal **New World monkeys** of the **genus** ***Aotus***, the only living member of the **family** **Aotidae** ([/eɪˈoʊɪdiː/](#)). The genus comprises eleven species which are found across

NightMonkey

- In the SpiderMonkey tradition: JaegerMonkey, IonMonkey, WalrusMonkey
- NIGHT: Nonlocal Inference with Guiding Heuristics for Types

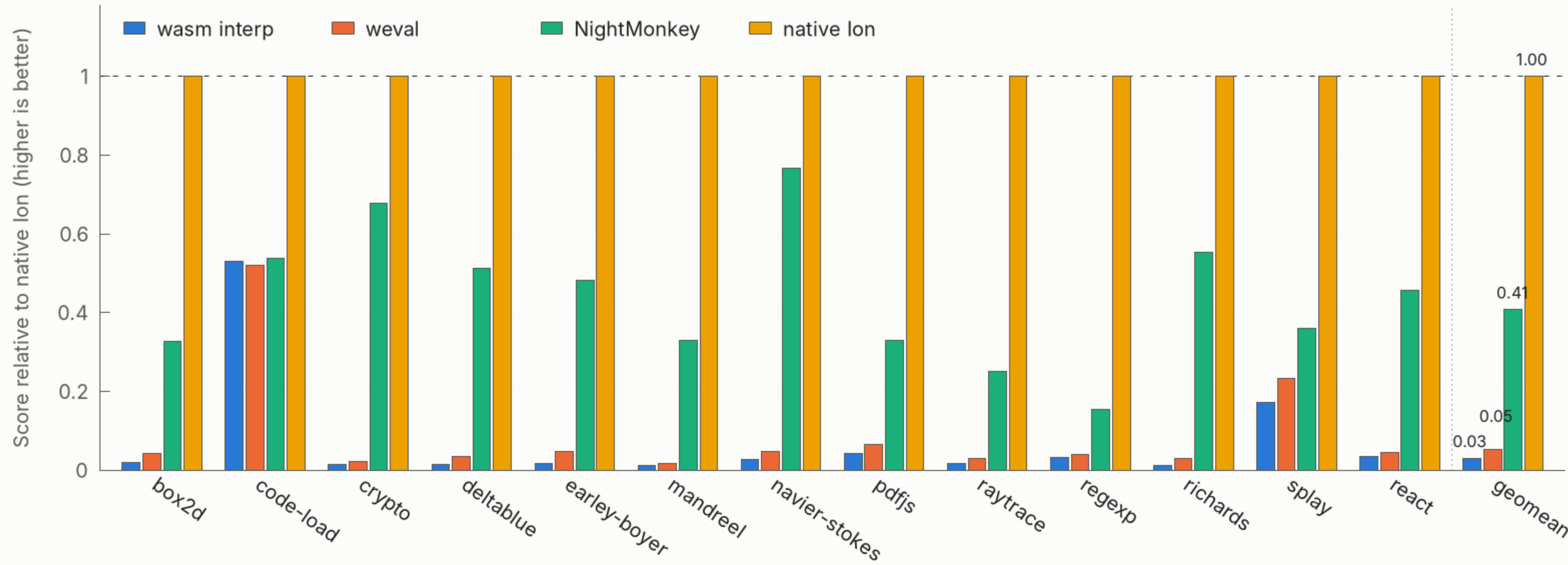


Night monkeys, also known as owl monkeys or douroucoulis^[2] ([/duːruːˈkuːliz/](#)), are nocturnal New World monkeys of the genus **Aotus**, the only living member of the family **Aotidae** ([/eɪˈoʊtɪdiː/](#)). The genus comprises eleven species which are found across

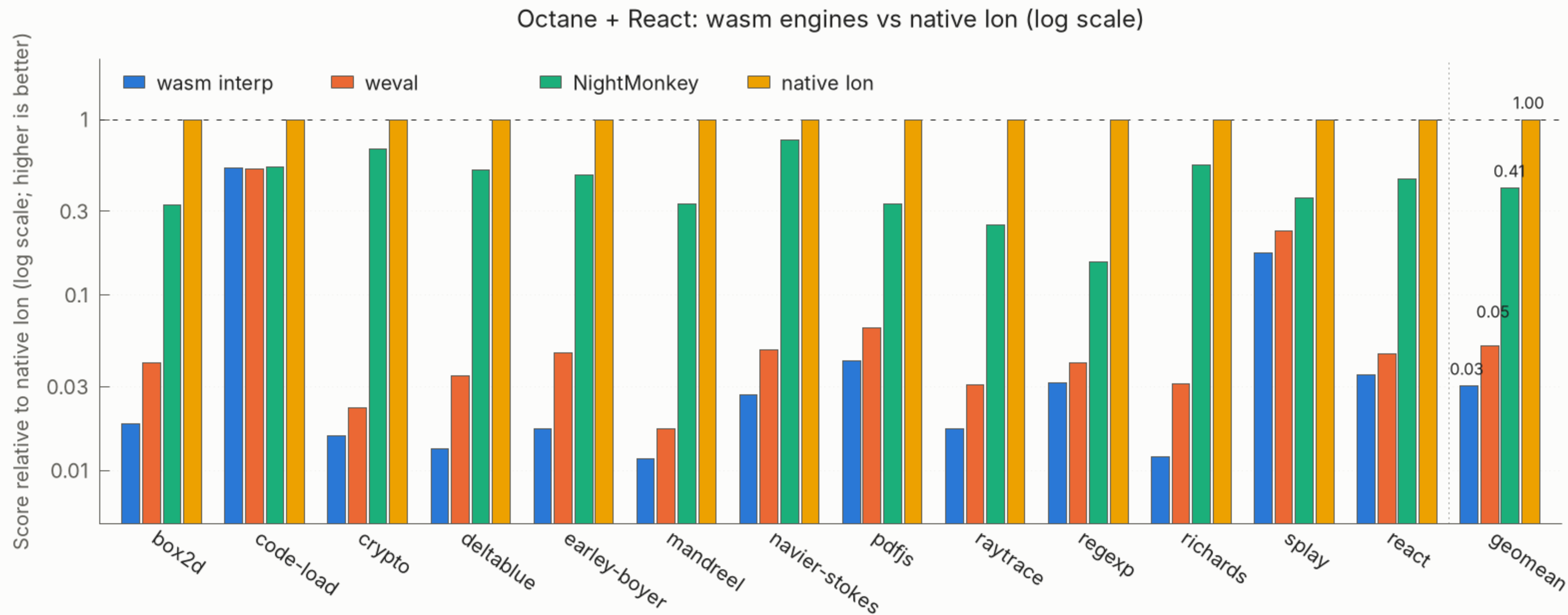
!!!

Results

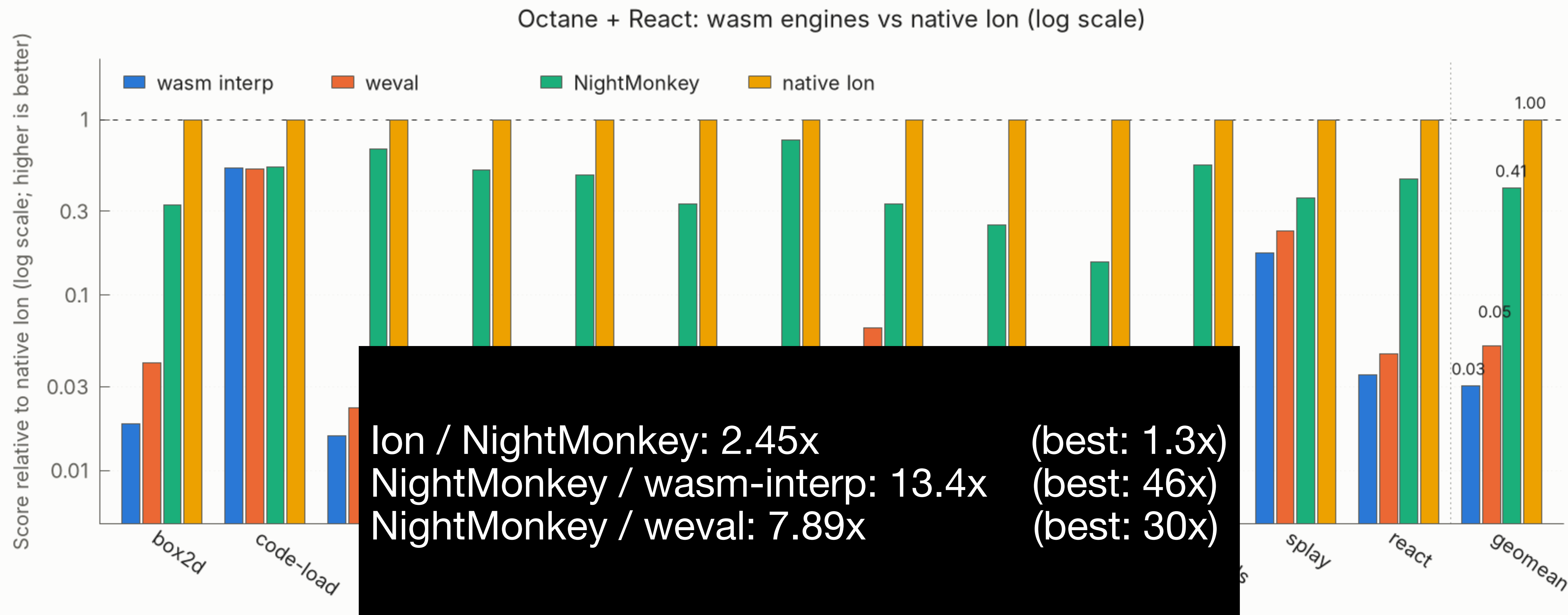
Octane + React: wasm engines vs native Ion



Results (Log Scale)



Results (Log Scale)



Effect Breakdown

<i>Config</i>	Performance vs. NightMonkey
No likely-types facts	0.44x
No heap abstractions/types	0.76x
No direct call targets	0.82x
No MIR-level inlining	0.87x
No field-slot predictions	0.88x
No interprocedural arg/return types	0.88x
No int32 specialization for numerics	0.89x
No MIR opts	0.95x

vs. Porffor

Benchmark	Score	Status
richards	1470	passes
crypto	—	hang
raytrace	—	TypeError: undefined is not a function
earley-boyer	—	ReferenceError: k is not defined
regexp	—	ReferenceError: re0 is not defined
splay	—	hang
navier-stokes	—	261MiB bytecode in one function
pdfjs	—	Porffor compiler stack overflow
mandreel	—	Porffor compiler stack overflow
code-load	—	eval/new Function not supported
box2d	—	TypeError: undefined is not a function
react-bench	—	ReferenceError: getOwnPropNames

vs. Porffor

AOT: ~16.4k; native JIT: ~28k

Benchmark	Score	Status
richards	1470	passes
crypto	—	hang
raytrace	—	TypeError: undefined is not a function
earley-boyer	—	ReferenceError: k is not defined
regexp	—	ReferenceError: re0 is not defined
splay	—	hang
navier-stokes	—	261MiB bytecode in one function
pdfjs	—	Porffor compiler stack overflow
mandreel	—	Porffor compiler stack overflow
code-load	—	eval/new Function not supported
box2d	—	TypeError: undefined is not a function
react-bench	—	ReferenceError: getOwnPropNames

vs. Porffor

- Porting effort required
 - wasm-opt to convert legacy exceptions to exnrefs
 - Custom ABI polyfill
 - Refactor Octane harness not to use closures
- Very impressive + important project! But targets a different use-case:
 - New code written from scratch, can avoid missing features
 - No interprocedural type inference (AFAIK); TS annotations instead

Discussion / Lessons Learned

- Philosophy seems powerful: unsound / optimistic analysis + guards
 - Most reasonable programs that care about perf will try to be “regluar”:
 - avoid shape heterogeneity
 - avoid type heterogeneity except with localized, controlled polymorphism
 - avoid weird corners of the language (e.g. 0 vs -0, ...)

Discussion / Lessons Learned

- Philosophy seems powerful: unsound / optimistic analysis + guards
 - Most reasonable programs that care about perf will try to be “regluar”:
 - avoid shape heterogeneity
 - avoid type heterogeneity except with localized, controlled polymorphism
 - avoid weird corners of the language (e.g. 0 vs -0, ...)
 - “Weird language semantics aren’t real and can’t hurt us” (in analysis)

Discussion / Lessons Learned

- Don't try to beat Rice's Theorem
 - It's *really* tempting to do “*just one more analysis, I swear this is the last one*”
 - You *can* overfit to benchmarks; do you want to?

Discussion / Lessons Learned

- Simple principles beat “soup of heuristics”
 - Basic block versioning fell over because of too many “just so” context-management heuristics
 - Instead, “MIR is always most-optimal; any loss of types/guards/assumptions goes to baseline”

Discussion / Lessons Learned

- I don't see a reason why this *couldn't* apply to other languages
 - But the above truths may vary in degree: e.g. weird language semantics matter if most programs use them
 - Critically, *you need an init phase and snapshot* and the *weird-metaprogramming-happens-in-snapshot* hypothesis to hold

Discussion / Lessons Learned

- I don't see a reason why this *couldn't* apply to other languages
 - But the above truths may vary in degree: e.g. weird language semantics matter if most programs use them
 - Critically, *you need an init phase and snapshot* and the *weird-metaprogramming-happens-in-snapshot* hypothesis to hold
 - This is kind-of “the JIT hypothesis”, but with explicit phases

Next Steps

- Merge into StarlingMonkey (PRs open); ship it!
- What's left in codegen quality?
 - A few cases of “overly clever runtime dispatch” that foil likely-types — unclear how to solve
 - E.g. pdfjs: string-keyed callback+context pairs in a dictionary
 - Undoubtedly more fastpaths to implement (versus Warp+CacheIR's years of opts)
- Can we ingest TypeScript ascriptions as (better) likely-types seeds?
 - This gives a nice progressive performance-engineering path to the user, too
- Can this approach be applied to other dynamic languages?

Thanks! Questions?

Chris Fallin, F5
VMIL 2026 (at SPLASH 2026)
Oct 8, 2026, Oakland, CA

Ahead-of-time Compilation of JavaScript to WebAssembly via Static Type Inference