SJS:

A Type System for JavaScript with Fixed Object Layout

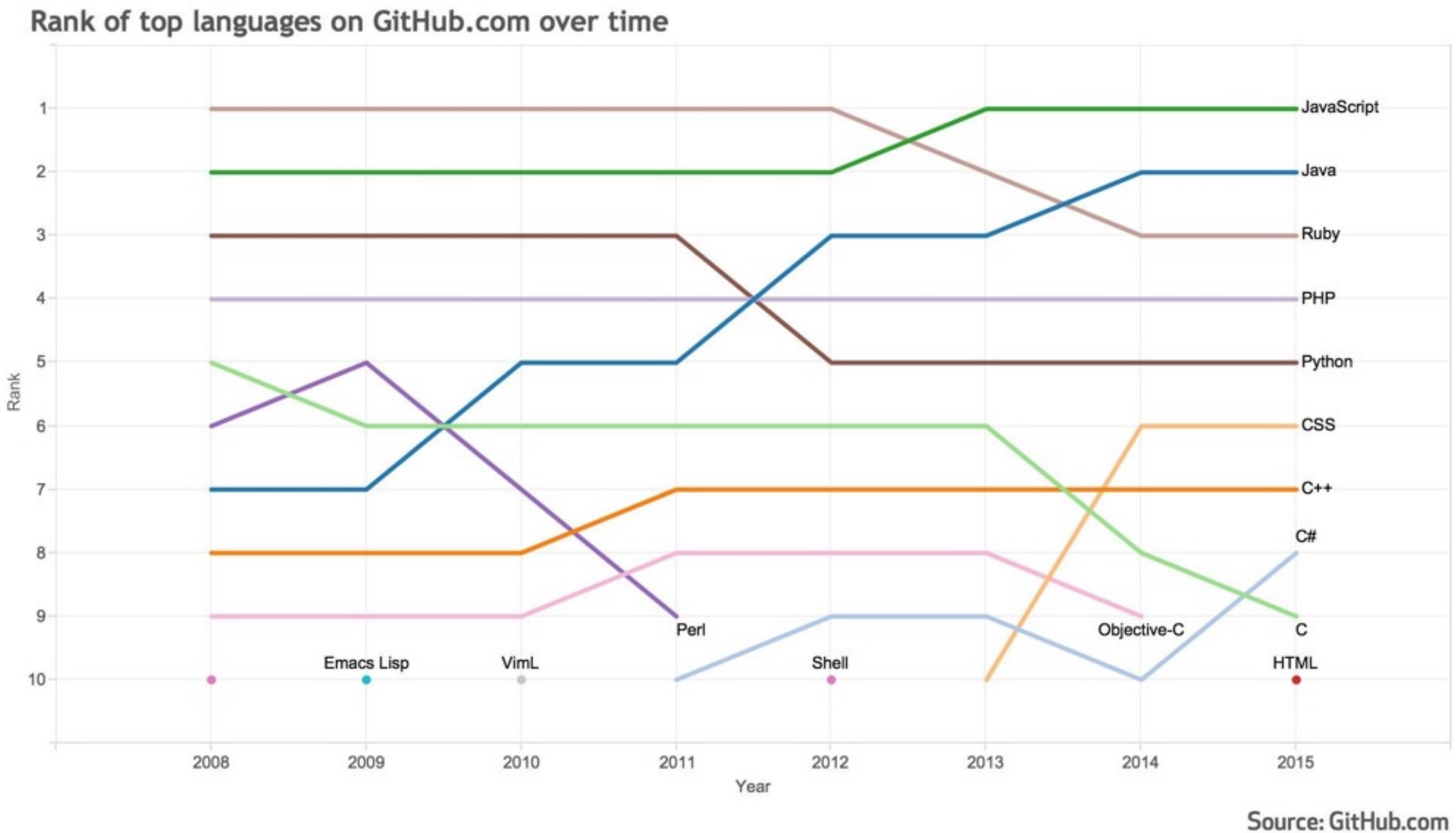
Wontae Choi*, Satish Chandra+, George Necula*, and Koushik Sen*

SAS 2015 @ Saint-Malo, France

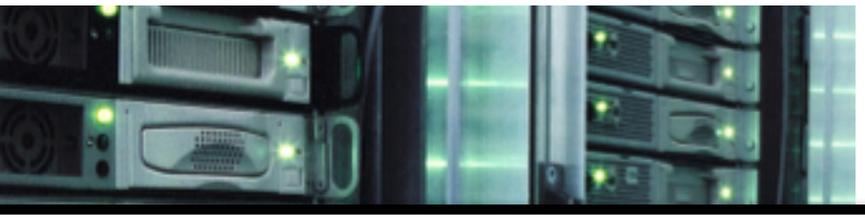
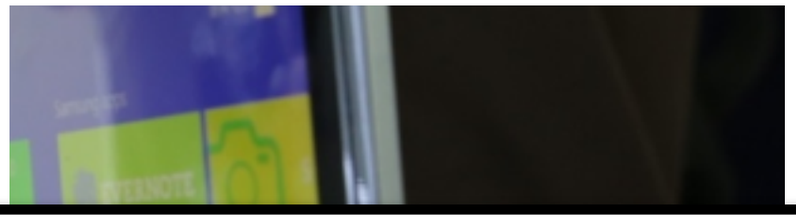
* University of California, Berkeley + Samsung Research America

This project started during a summer internship of the first author
at Samsung Research America in 2013 and 2014.

Why JavaScript? Popular



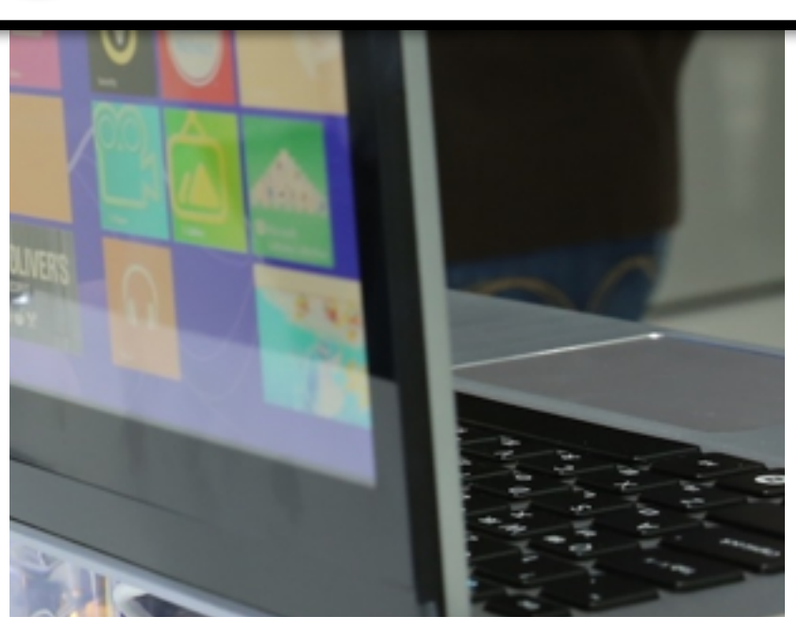
Portable Web/Server-side/Desktop Apps










Dynamic Yet Fast: JIT

- Resolve expensive dynamic features at runtime:
  - E.g. Property access
    - x.foo, x["foo"], x["fo"+"o"]
      - Objects are hash tables (+ prototype)
      - Optimized via inline-caching (with runtime-type analysis)

JIT Not Suitable for Small Devices

JIT Not Suitable for Small Devices

- JIT compilers are memory hungry / draining energy.
- May quickly exhaust resources from small devices …

AOT instead of JIT

- Ahead-of-time (AOT) compilation
  - provides a similar performance
  - without the cost of resource hungry JIT.

- Requirements:
  - static type system
  - with fixed-object layout property.

SJS: Lightweight JavaScript

Requirements

- Statically compiled
- Statically typed (sound)
- Portable (subset of JavaScript)
- High-level features
- Light-weighted Annotation

SJS: Lightweight JavaScript

Requirements	Existing systems
	asm.js
• Statically compiled	✓
• Statically typed (sound)	✓
• Portable (subset of JavaScript)	✓
• High-level features	✗
• Light-weighted Annotation	✗

SJS: Lightweight JavaScript

Requirements	Existing systems	
	asm.js	TypeScript
• Statically compiled	✓	✗
• Statically typed (sound)	✓	▲
• Portable (subset of JavaScript)	✓	▲
• High-level features	✗	✓
• Light-weighted Annotation	✗	✓

SJS: Lightweight JavaScript

Requirements	Existing systems		Our System
	asm.js	TypeScript	SJS
• Statically compiled	✓	✗	✓
• Statically typed (sound)	✓	▲	✓
• Portable (subset of JavaScript)	✓	▲	✓
• High-level features	✗	✓	✓
• Light-weighted Annotation	✗	✓	✓

SJS: Lightweight JavaScript

Requirements	Existing systems		Our System
	asm.js	TypeScript	SJS
• Statically compiled	✓	✗	✓
• Statically typed (s			✓
• Portable (subset of JavaScript)	✓	▲	✓
• High-level features	✗	✓	✓
• Light-weighted Annotation	✗	✓	✓

Why no static compilation?
It does not guarantee fixed-object layout!

Fixed Object Layout ?

- Standard record type system is not useful for JavaScript.

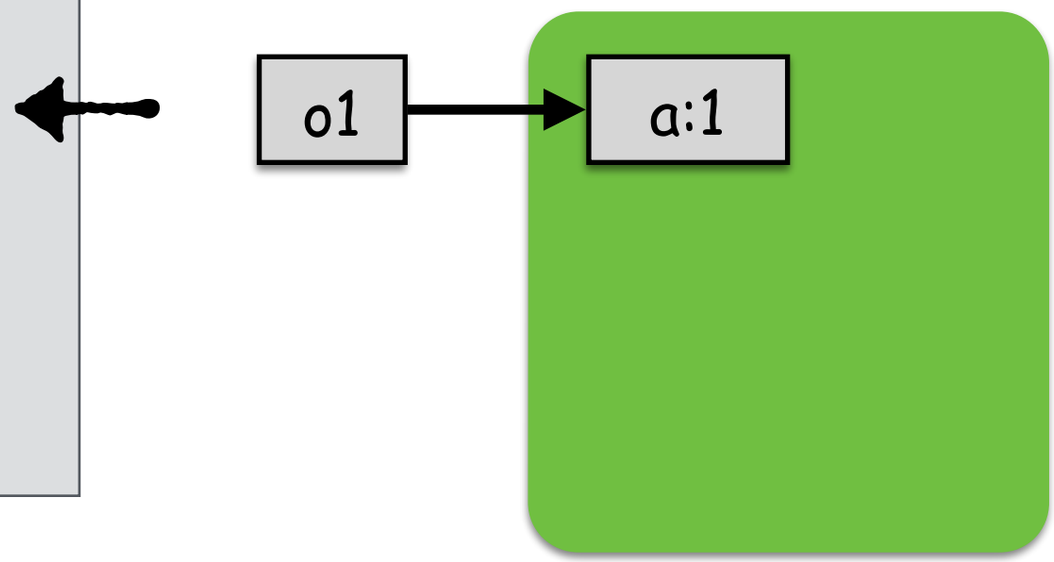
- Three main challenges:

1. Prototyping
2. Methods
3. Subtyping

Challenge #1: Prototype

Example

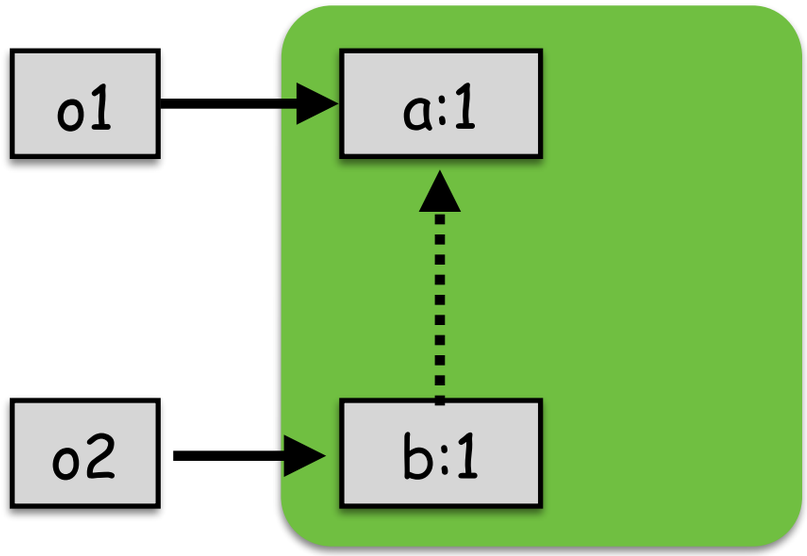
```
// o1: { a: Int }
var o1 = { a: 1 };
```



Challenge #1: Prototype

Example

```
// o1: { a: Int }
var o1 = { a: 1 };
// o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };
```



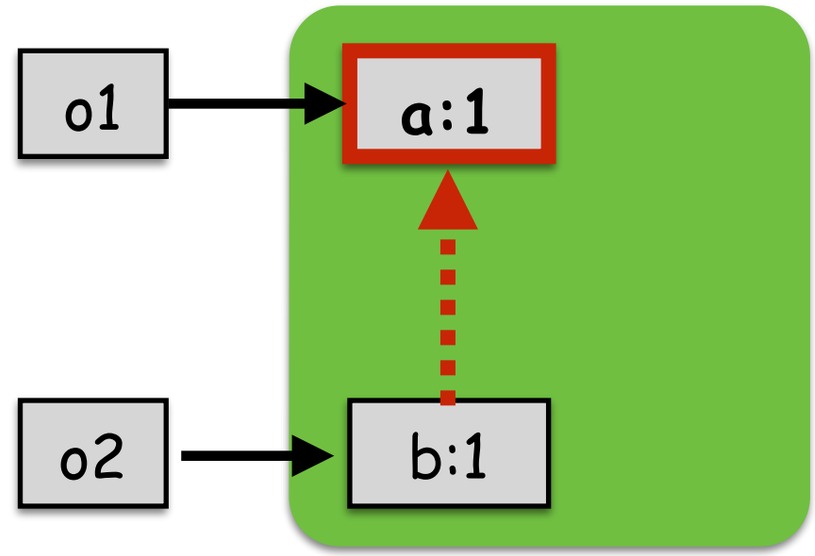
An object can have a prototype object.

Challenge #1: Prototype

Example

```
// o1: { a: Int }
var o1 = { a: 1 };
// o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };

print(o2.a);
```



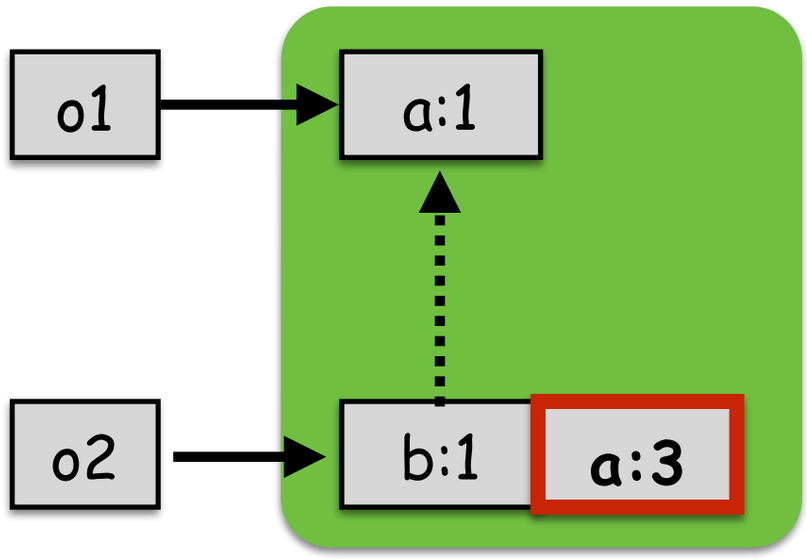
A failed read is delegated to the prototype.

Challenge #1: Prototype

Example

```
// o1: { a: Int }
var o1 = { a: 1 };
// o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```



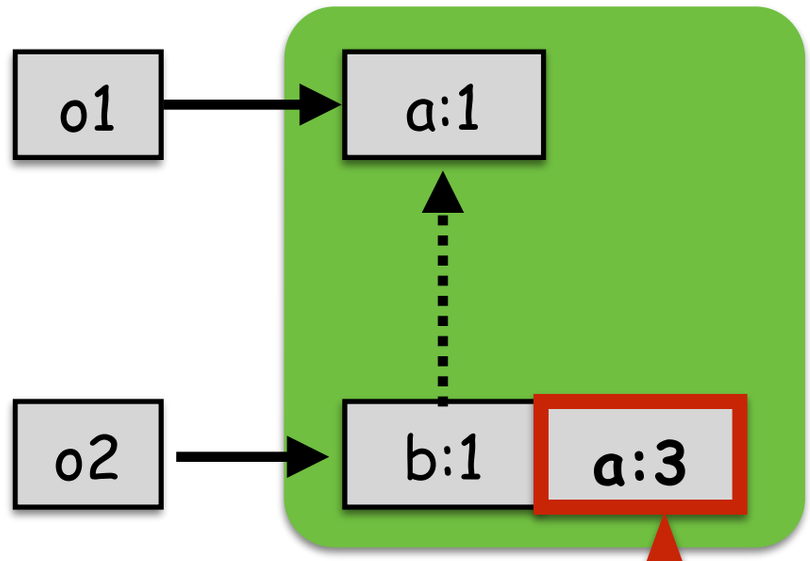
A write operation can adds an attribute
even the attribute exist!

Challenge #1: Prototype

Example

```
// o1: { a: Int }
var o1 = { a: 1 };
// o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```



A write operation can c
even the attribu

Layout has changed !!
Bad for AOT compilation.

Challenge #1: Prototype

Solution: attribute ownership

```
//o1: { a: Int }
var o1 = { a: 1 };
//o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```

Attribute Ownership

- Type system tracks the ownership of attributes

Challenge #1: Prototype

Solution: attribute ownership

```
//o1: { a: Int }, own = {a}
var o1 = { a: 1 };
//o2: { a: Int, b:Int }
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```

Attribute Ownership

- Type system tracks the ownership of attributes

Challenge #1: Prototype

Solution: attribute ownership

```
//o1: { a: Int }, own = {a}
var o1 = { a: 1 };
//o2: { a: Int, b:Int }, own = {b}
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```

Attribute Ownership

- Type system tracks the ownership of attributes

Challenge #1: Prototype

Solution: attribute ownership

```
//o1: { a: Int }, own = {a}
var o1 = { a: 1 };
//o2: { a: Int, b:Int }, own = {b}
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```

Attribute Ownership

Type Error !!
a is not owned by o2

- Type system tracks the ownership of attributes
- For update operations, attribute ownership is checked.

Challenge #1: Prototype

Solution: attribute ownership

```
//o1: { a: Int }, own = {a}
var o1 = { a: 1 };
//o2: { a: Int, b:Int }, own = {b}
var o2 = { b: 1, __proto__: o1 };


o2.a = 3;
```

Attribute Ownership

Type Error !!
a is not owned by o2

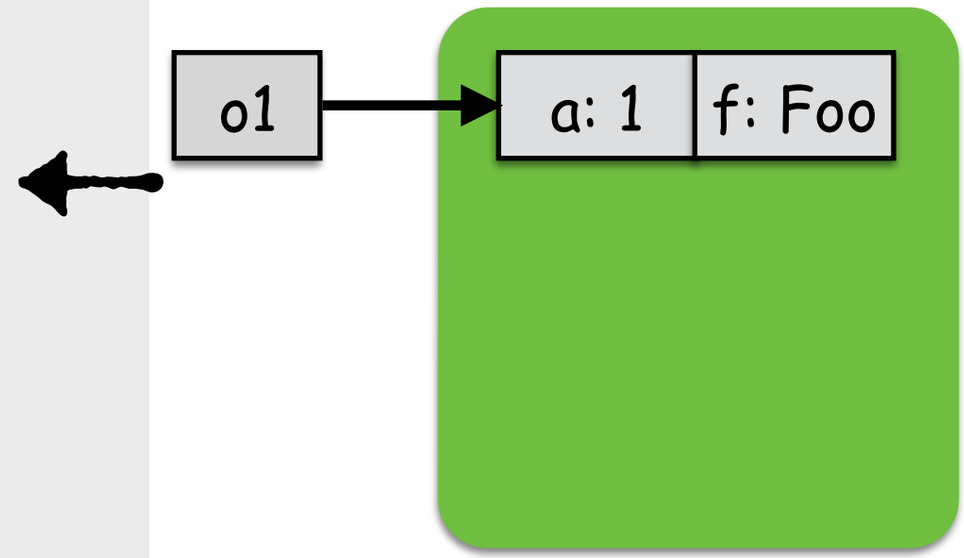
- Type system tracks the ownership of attributes
- For update operations, attribute ownership is checked.

Challenge #2: Method

Example

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };
```



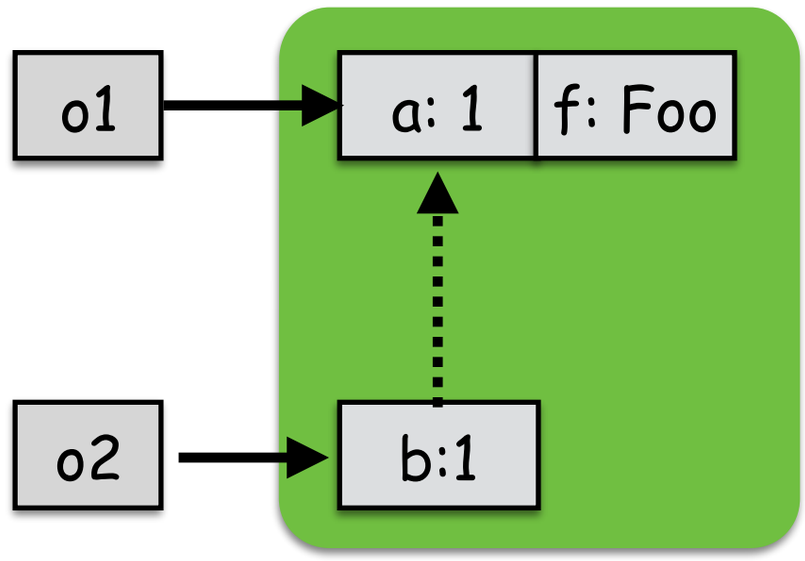
Challenge #2: Method

Example

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };
```



Challenge #2: Method

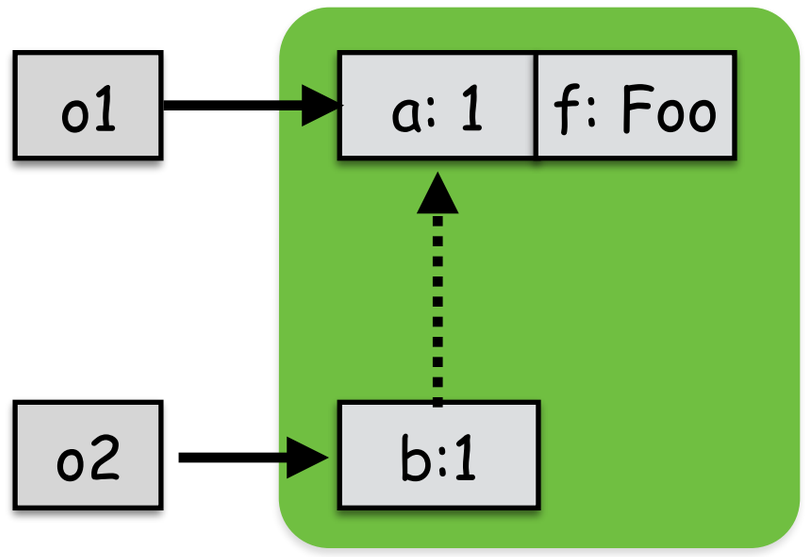
Example

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```



Challenge #2: Method

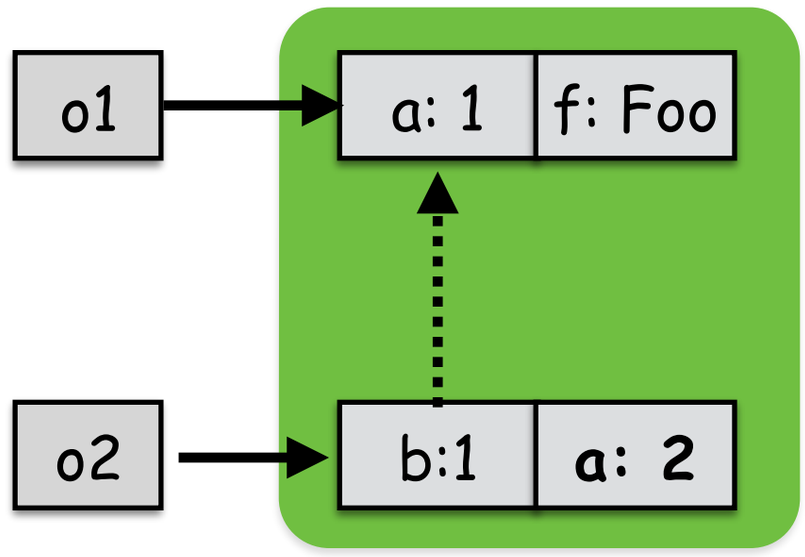
Example

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```



A method call can update an attribute (and layout)

Challenge #2: Method

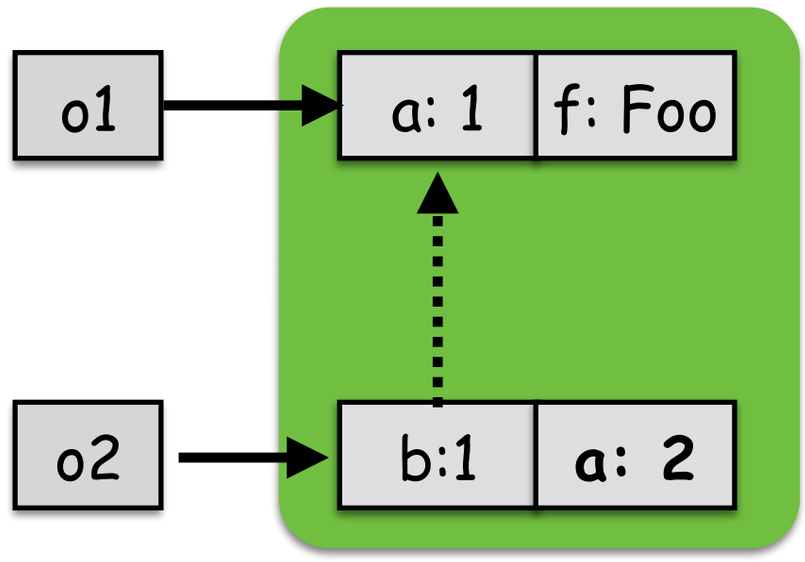
Example

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```



No information about f().
Ownership is not enough!

A method call can update an attribute (and layout)

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

- Tracking attributes which should be owned by inheritors (iown).

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

Inheritors should own it.

- Tracking attributes which should be owned by inheritors (iown).

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}, iown = {a}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

- Tracking attributes which should be owned by inheritors (iown).

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}, iown = {a}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

Inheritors should own it.

- Tracking attributes which should be owned by inheritors (iown).
- Inheriting objects should own them.

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}, iown = {a}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

Inheritors should own it.

Type Error !!
o2 does not own a

- Tracking attributes which should be owned by inheritors (iown).
- Inheriting objects should own them.

Challenge #2: Method

Solution: inheritor-own attributes

```
function Foo (x) { this.a = 2 }

// o1: { a: Int, f: Int=>Undef }
// own = {a, f}, iown = {a}
var o1 = { a: 1,  f: Foo };

// o2: { a: Int, b: Int, f: Int=>Undef }
// own = {b}
var o2 = { b: 1, __proto__: o1 };

o2.f();
```

Inheritors should own it.

Type Error !!
o2 does not own a

- Tracking attributes which should be owned by inheritors (iown).
- Inheriting objects should own them.

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4
```

o4 looks like a subtype of **o3**

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
```

o5 owns **a**, which is required to inherit **o3**.

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
o5.f();
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
o5.f();
```

invoke **o4.f()**

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Example

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
o5.f();
```

Layout will change!!

invoke o4.f()

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Source of the problem

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4
```

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Source of the problem

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},   iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},   iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4
```

iown is overshadowed.

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Source of the problem

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
```

Checking with imprecise **iown**

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Source of the problem

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
```

Checking with imprecise **iown**

iown should be precise for prototyping

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Source of the problem

```
// o3: { a: Int, f: Int=>Undef }
// own = {a, f},  iown = {a}
var o3 = { a: 1, f: fun(x){this.a = 2} }

// o4: { a: Int, c:Int, f: Int=>Undef }
// own = {a, c, f},  iown = {a, c}
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o5: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o5 = { a: 4, __proto__: o3 }
```

To subtype or not to subtype …

Checking with imprecise **iown**

iown should be precise for prototyping

Subtyping and Prototyping do not play well together .

Challenge #3: Subtyping

Solution: Precise and Approximate objects

- **Operations**
  - **Prototyping** for precise types
  - **Subtyping** for approximate types
- **Creation**
  - A new object has a precise type.
- **Casting**
  - A precise type can be downcast to an approximate type.

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }
```

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }


o3 = o4
```

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4
```

Error !!
No subtyping on precise types

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o6: approx. { a: Int, f: Int=>Undef }
// own = {a}
var o6 = (*) ? o3 : o4
o6.f()
```

Error !!
No subtyping on precise types

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o6: approx. { a: Int, f: Int=>Undef }
// own = {a}
var o6 = (*) ? o3 : o4
o6.a = 1
o6.f()
```

Error !!
No subtyping on precise types

Safe:
Downcasting (implicit)

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o6: approx. { a: Int, f: Int=>Undef }
// own = {a}
var o6 = (*) ? o3 : o4
o6.a = 1
o6.f()
```

Error !!
No subtyping on precise types

Safe:
Downcasting (implicit)

Safe:
Updates and method calls are allowed.

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o6: approx. { a: Int, f: Int=>Undef }
// own = {a}
var o6 = (*) ? o3 : o4
o6.a = 1
o6.f()

// o7: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o7 = { a: 4, __proto__: o6 }
```

Error !!
No subtyping on precise types

Safe:
Downcasting (implicit)

Safe:
Updates and method calls are allowed.

Challenge #3: Subtyping

Solution: Precise and Approximate objects

```
// o3 and o4 are precise
var o3 = { a: 1, f: fun(x){this.a = 2} }
var o4 = { a: 2, c: 3,
           f: fun(x){this.c = 4} }

o3 = o4

// o6: approx. { a: Int, f: Int=>Undef }
// own = {a}
var o6 = (*) ? o3 : o4
o6.a = 1
o6.f()

// o7: { a: Int, f: Int=>Undef }
// own = {a}, iown = {}
var o7 = { a: 4, __proto__: o6 }
```

Error !!
No subtyping on precise types

Safe:
Downcasting (implicit)

Safe:
Updates and method calls are allowed.

Error:
No prototyping on approximate types.

Theoretic Result

- **Formally defined core calculus**
  - Static and dynamic semantics.
  - Objects, high-order functions, method updates, first-class method value, prototyping, and subtyping.
- **Fixed object layout theorem**
  - A well-typed program never modifies object layouts after object construction.
- **Corollary**
  - A well-typed programs can be compiled ahead-of-time.

Evaluation: Implementation

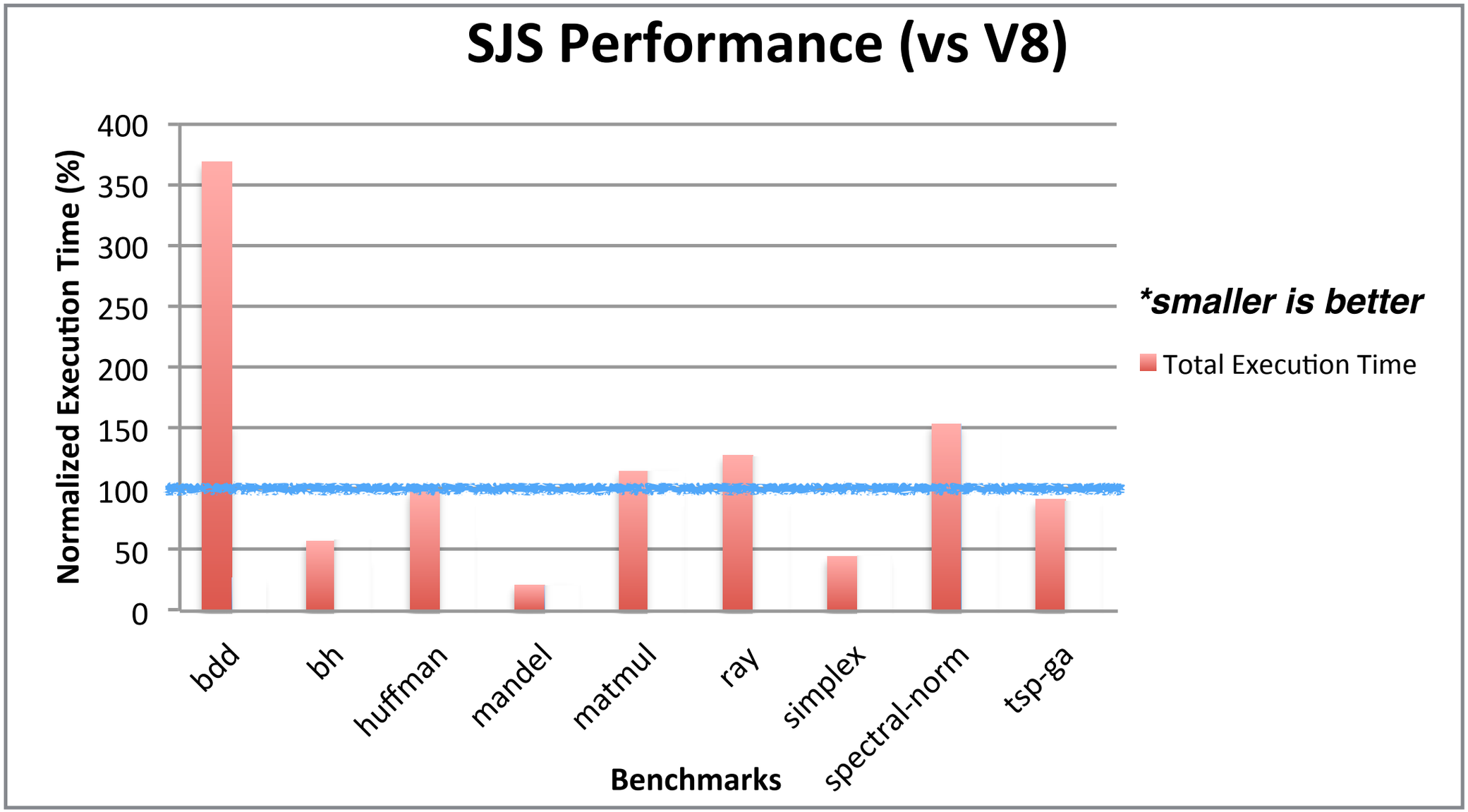
- **Type inference engine + Compiler (to C).**
  - Type inference requires annotations for base types.
  - Qualifiers (iown, own, etc.) are automatically inferred.
  - The resulting C program is compiled with the **Boehm garbage-collector** using **Clang**.

Evaluation: Usability

- **Benchmarks**
  - 2 web apps
  - 2 octane benchmarks
  - 500-2000 lines of code

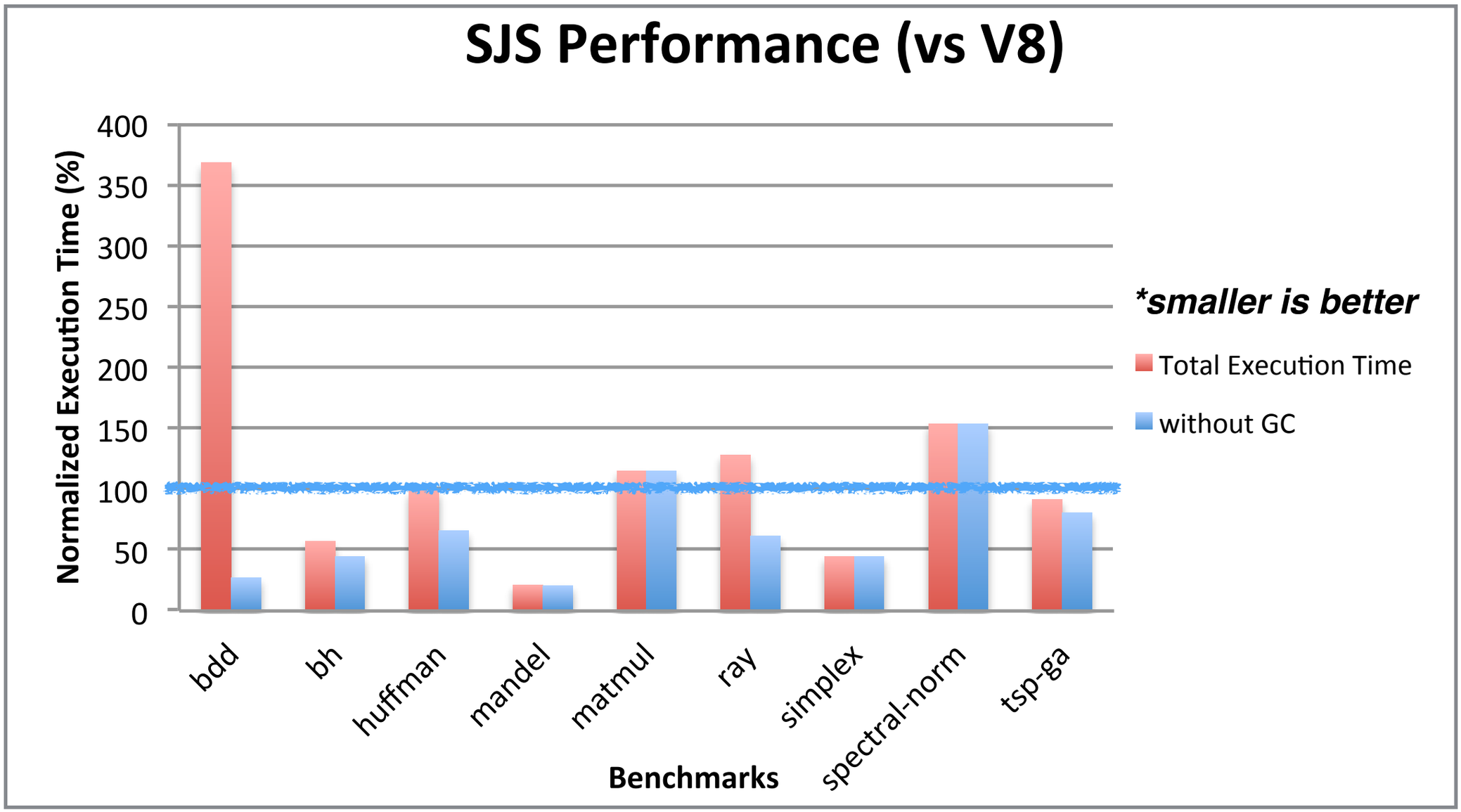
- **Results**
  - 1 type annotation / 8.34 lines of code.
  - 86% of annotations are for function parameters.

Evaluation: Performance



- A performance compatible with **node.js**
- Without using a resource hungry JIT!

Evaluation: Performance



- A performance compatible with **node.js**
- Without using a resource hungry JIT!

Summary

Conclusion

AOTC of JavaScript is doable **with**
a type system guaranteeing fixed object layout.

Summary

Conclusion

AOTC of JavaScript is doable **with**
a type system guaranteeing fixed object layout.

AOTC provides JIT like performance
without the cost of JIT.