JavaScript Sealed Classes

Manuel Serrano
Inria/UCA, Inria Sophia Méditerranée, 2004 route des Lucioles, Sophia Antipolis, France

Abstract

In this work, we study the JavaScript *Sealed Classes*, which differ from regular classes in a few ways that allow ahead-of-time (AoT) compilers to implement them more efficiently. Sealed classes are compatible with the rest of the language so that they can be combined with all other structures, including regular classes, and can be gradually integrated into existing code bases.

We present the design of the sealed classes and study their implementation in the `hopc` AoT compiler. We present an in-depth analysis of the speed of sealed classes compared to regular classes. To do so, we assembled a new suite of benchmarks that focuses on the efficiency of the `class` implementations. On this suite, we found that sealed classes provide an average speedup of 19%. The more classes and methods programs use, the greater the speedup. For the most favorable test that uses them intensively, we measured a speedup of 56%.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases JavaScript, Compiler, Dynamic Languages, Classes, Inline Caches

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.24

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.23`
Software (Source Code): `https://github.com/manuel-serrano/hop.git`

Acknowledgements My gratitude to L. Tratt for his suggestions to improve this paper, and to O. Melançon, E. Rohou, M. Feeley, and R. Findler for their comments, suggestions, and corrections.

1 Introduction

JavaScript's dynamicity makes the language very flexible and malleable. Programs are easy to adapt as their specification evolves. Data structures can be extended to meet new API requirements and extensions are easy to connect to existing code bases, etc. However, there is a flip side of this coin. First, many minor mistakes such as misspelled identifiers or omitted function arguments are usually not reported, which leads to programs that are incorrect without the programmer noticing. Second, the language is particularly difficult to implement efficiently. All fast JavaScript compilers, be they static (AoT) or dynamic (JIT), rely on heuristics and optimistic compilation to bridge the performance gap to more static languages. Consequently JavaScript programs suffer from the *performance cliff* syndrome [2, 16]: a mundane modification of the source code might dramatically affect its performance if the change, possibly very simple, defeats the compiler's heuristics.

To mitigate these problems, we explore the design and implementation of a dialect of JavaScript that extends the language with a few constructs and annotations that together allow compilers, specially AoT compilers, to generate better code. This project follows the same philosophy as JavaScript *strict mode* and *strong mode* [23]: programmers who are willing to sacrifice some dynamicity of their implementations will be rewarded with faster and more predictable performance. Sealed classes are a central component of this dialect and they are the subject of this paper.

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 24; pp. 24:1–24:27



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Sealed class instances and sealed class objects are class instances and class objects that are deprived of specific freedoms but that remain fully compatible with the rest of the language. All existing JavaScript programs can continue to run without any modification. Sealed classes trade some dynamicity for a more efficient implementation, in part because their design allows compilers to reuse the efficient techniques developed for class-based programming languages such as Smalltalk or Java.

In this paper we show that sealed classes are beneficial to the hopc [25, 26] AoT compiler. We show that they help it deliver faster and more predictable code. Although not studied in this paper, we believe that sealed classes could also benefit to JIT compilation because AoT and JIT compilers share many similar techniques for manipulating objects and properties. The main contributions of this work are:

- The characterization of lightweight constraints that allow AoT compilers to optimize the implementation of classes.
- A proof based on a complete implementation that sealed classes improve JavaScript performance of a full-fledged AoT compiler.
- An improvement to a well known technique for implementing single-inheritance class type checks.
- A benchmark suite for evaluating the performance of JavaScript classes.

The paper is organized as follows. We start in Section 2 with a brief introduction to JavaScript classes and we show why they are more difficult to implement that those of languages such as Java or C++. These difficulties motivate the introduction of sealed classes that are presented in Section 3. Their implementation is presented in Section 4. We measure their efficiency in Section 5. We present the related work in Section 6 and we conclude in Section 7.

2 Classes

Classes were added to JavaScript in version 6 [12]. They are syntactic extensions of functions [18]. They have brought a class-based programming flavor to JavaScript, without requiring any new runtime operations. Here are examples of class declarations collected from the MDN web site [19]:

```
class BaseClass {
  msg = 'hello world'
  basePublicMethod() {
    return this.msg
  }
}
class SubClass extends BaseClass {
  subPublicMethod() {
    return super.basePublicMethod()
  }
}
const instance = new SubClass()
console.log(instance.subPublicMethod()) // "hello world"
```

Recently classes have been extended to support private fields. These must be declared in the class, unlike normal properties whose declarations are optional. Private properties are prefixed with '`#`'.

```
class ClassWithPrivateField {
  #priv;
  constructor() {
    this.#priv = 42;
  }
  str() {
     return `${this.#priv}`;
  }
}
class SubClass extends ClassWithPrivateField {
  #subpriv;
  constructor() {
    super();
    this.#subpriv = 23;
  }
  str() {
     return `${this.#subpriv} ${super.str()}`;
  }
}
new SubClass();
```

Private properties are only accessible from within the methods of the class that defined them. They are not accessible from within methods of subclasses or from outside the class. In our example, the `#priv` private property is only accessible from the methods of `ClassWithPrivateField`, but not from the methods of `SubClass`. Attempting to access one raises the error "*Private field must be declared in an enclosing class*".

2.1 Class Implementation

As classes are merely *special functions* [18], they are naturally implemented as a translation into functions and prototype chains. For instance, here is how the TypeScript compiler transforms the two declarations above into plain JavaScript. JavaScript compilers do an equivalent transformation internally.

```
var BaseClass = (function () {
  function BaseClass() {
    this.msg = 'hello␣world';
  }
  BaseClass.prototype.basePublicMethod = function () {
   return this.msg;
  };
  return BaseClass;
}());
var SubClass = (function (_super) {
  function SubClass() {
    return _super !== null && _super.apply(this, arguments) || this;
  }
  SubClass.prototype.__proto__= _super.prototype;
  SubClass.prototype.subPublicMethod = function () {
    return _super.prototype.basePublicMethod.call(this);
  };
  return SubClass;
}(BaseClass));
```

Each class is transformed into a plain JavaScript function (lines 1 and 10 in the example). The inheritance relation between a class and its superclass is implemented by chaining the prototype object of the class with the prototype object of the superclass (see at line 14 in the example). Class properties are stored in the instances themselves; methods are stored in the prototype chains. Figure 1 shows the memory layout of `BaseClass`, `SubClass`, and one instance of `SubClass`.

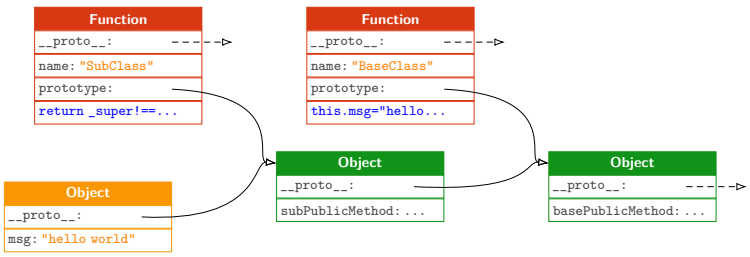


Figure 1 Two JavaScript classes and one instance.

Since a class instance is indistinguishable from a plain JavaScript object, the implementation techniques used for accessing plain objects are also those used for accessing class instances, namely inline caches [14]. We recall in the following paragraph the main principle of inline caches and hidden classes and we show their limitations when used for implementing JavaScript class accesses, re-using Serrano & Findler's presentation [28].

According to the JavaScript specification [12], accessing an object property involves the following steps:

1. convert the property name into a string S;
2. if the object owns a property S, return its value;
3. if the object has a prototype, restart at step (2) with the prototype object, return `undefined` otherwise.

The central point of the property access process is step (2). Since new properties can be added or removed at any time, checking whether an object owns a property involves looking up a key (the property name) in a dictionary (the object). Implemented literally, this protocol is several orders of magnitude slower than those of languages for which reading a structure field is a single memory read whose address is computed by adding an offset known at compile time to a base pointer.

The classic method for optimizing accesses to properties consists of associating to each object a *hidden class* and to each access a *lookup cache* [9, 3, 4]. When the property name is statically known, a very common case, a property access `obj.prop` can be implemented as follows in C (we use C to demonstrate such implementations throughout the paper):

```
1 if (obj->hclass == cache.hclass) {
2    val = obj->elements[cache.index];          // cache hit
3 } else {
4    val = cacheReadMiss(obj, "prop", &cache); // cache miss
5 }
```

When the execution environment supports dynamic code modification, the hidden class `cache.hclass` can be directly encoded as the operand of the assembly instruction implementing the test at line 1 as well as the offset `cache.index` of the property at line 2. This is where the name *inline cache* comes from.

Objects hidden classes evolve over time. One object's hidden class changes every time a property is added or removed. In the following example

```
1 const o = {};
2 o.x = 34;
3 o.y = 45;
4 delete o.x;
```

the object `o` will be successively associated with four different hidden classes: at line 1 with $h_0 = \{\}$, at line 2 with $h_1 = \{x \to 0\}$, at line 3 with $h_2 = \{x \to 0, y \to 1\}$, and at line 4 with $h_3 = \{y \to 1\}$.

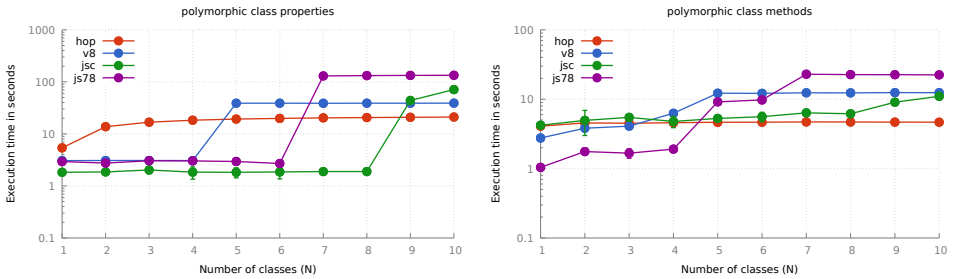


Figure 2 Impact of polymorphism on property accesses and method invocations for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). Horizontal axis is the number of classes involved, aka the degree of polymorphism. The vertical axis is the execution time. Lower is better. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times and the average is reported.

Method calls are handled by similar but subtly different techniques. Class methods are stored in instance prototypes rather than in the instances themselves. Thus, for a method call, it is more efficient to store the method found in the prototype directly in the cache and to check the object proper only on method cache misses:

```
if (obj->hclass == cache.pclass) {
   val = CALLN(cache.method, obj, a0, a1, ...);                // prototype cache hit
} else if (obj->hclass == cache.hclass) {
   val = CALLN(obj->elements[cache.index], obj, a0, a1, ...); // object cache hit
} else {
   val = cacheMethodMiss(obj, "prop", &cache, a0, a1, ...);   // cache miss
}
```

This allows for an efficient method call sequence but requires a complex invalidation mechanism because when a prototype object is modified, all inline caches currently armed with that prototype method must be invalidated. This increases the unpredictability of method invocation performance.

Inline caches efficiently handle monomorphic accesses, *i.e.,* when all the objects used with one cache share the same hidden class, but they do not efficiently handle polymorphic accesses that occur when the objects accessed from a specific location have different types. For that, an enhanced technique named *Polymorphic Inline Caches* has been proposed [15]. It relies on more elaborate test sequences [20], so do properties that are not directly held by the objects themselves but by objects in the prototype chain, and properties implemented by getters and setters.

To measure the effectiveness of polymorphic caches on class implementations, we conducted an experiment following Serrano & Feeley's methodology [27]. We consider a 10-class hierarchy consisting of one base class and 9 subclasses with a maximum inheritance depth of 5. The first test repeatedly accesses a property of the base class. Each run accesses the property the same number of times, but the number of classes, N, varies. When N is 1, only one instance of a single class is used. When N is 2, two instances of two different classes are used. When N is 3, three instances of three classes are used, etc. The second experiment uses a similar principle, but adapted to method invocations. The results of these experiments are reported in Figure 2. We observe that not all systems *fall off the cliff* at the same time but, sooner or later, they all fall. Hopc is the first to fall for accessing properties when N equals 2, but it falls less deeply than the other systems with a slowdown of *only* 5×. It is followed by V8 when N is 5, then by SpiderMonkey at 7, and eventually JavaScriptCore when N is 9.

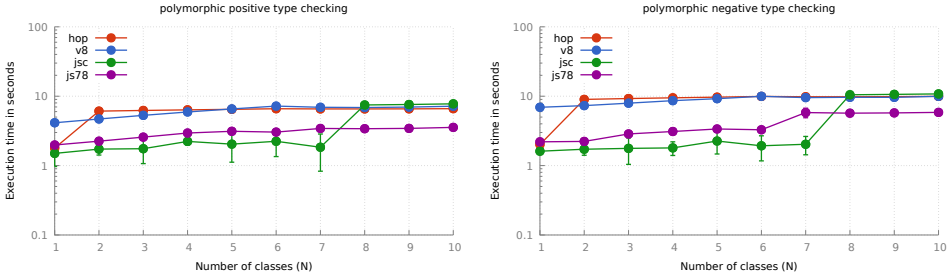


Figure 3 Impact of polymorphism on type predicates (`instanceof`) for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). The figure left-hand-side shows performance when type tests succeed. The right-hand-side, shows performance when type tests fail. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

The tuning of polymorphic inline caches have for sure been settled after careful examination of pre-class JavaScript programs, but this experiment shows that they are not well adapted for realistic applications that use class hierarchies to represent complex data structures. For instance, the hopc compiler represents its JavaScript abstract syntax tree with 52 different classes, so all the traversals that read properties from the root class would raise cache misses. Methods or properties accessed via the `super` keyword have the same limitations because they use inline cache-based implementations too [14].

The polymorphism also harms type predicates performance. This can be seen in the result of another experiment conducted with the same methodology, to evaluate the performance of the `instanceof` operator (Figure 3). Although less pronounced, we observe a similar phenomenon as with property accesses and method invocations: the performance degrades when the level of polymorphism increases.

☛ *Because instances of a class and instances of the super class are associated with different hidden classes, inline caches are not as well suited for implementing property accesses and type predicates of class-based programs as they are for regular objects.*

Implementing property accesses and type predicates efficiently is not the only difficulty of classes. Efficient instance creation is also a challenge, especially for AoT compilers, due to the dynamic nature of JavaScript class inheritance hierarchies. When declaring a class, its super class might not be known at compile time. The super-class position can be any expression, e.g., a function call or a variable reference. It is not even necessary for the super class to be another class. For instance, it may be an array or a function. This freedom has two important consequences. First, the compiler does not necessarily know the function to be invoked when compiling the `super` call (that JavaScript requires to appear before the first use of the `this` value in a constructor). This makes function inlining difficult. Second, the compiler does not know the memory size of the allocated instances. This difficulty is reinforced by instance properties that are not required to be statically declared but that can be added dynamically anywhere in the program. One proposed technique for dealing with variable allocation sizes [6] helps but the design of classes makes it complex to implement. For instance, class constructors are not even required to return the newly allocated objects, which implies extra tests if the allocation size is to be profiled.

The other difficulties come from the implementation of constructors themselves. A class is not required to declare a constructor. If it does not and if it inherits from another class, the constructor of the super class is called silently. In addition to retrieving the constructor function from the super class object and making the unknown call, a compiler faces a greater challenge: it does not necessarily know the arity of the super class constructor but,

nevertheless, all values that are passed to the constructor of the instance must be transmitted all the way up the inheritance tree, to the first declared constructor. This requires wrapping all arguments when invoking such a constructor. This has two negative impacts. First, there is cost associated with creating and initializing this wrapper. Second, it prevents constructors from receiving their arguments in registers as ordinary functions do. An alternative solution could be to use a variable arity calling convention for all constructors but it has the downside of slowing down all constructor calls.

If this meals were not already enough for the compiler, two other side dishes are also served. Inside a constructor `this` can only be used after the super constructor has been invoked, but a constructor is not required to use `this`. Resolving statically this so-called *dead zone* detection is undecidable, it is therefore not always possible to avoid generating a dynamic test that penalizes runtime efficiency. Secondly, JavaScript supports `new.target` expressions, which, inside constructors, return the classes used to create the instances. If the runtime system does not provide a way to traverse the stack for inspection purposes, `new.target` is required to be passed as an extra argument to the constructor.

☛ *The lack of static inheritance trees, the possibility of adding new properties inside or outside constructors, and the freedom to declare (or not) the constructor, make the efficient allocation and initialization of object instances challenging, especially for AoT compilers.*

Facing all these difficulties, a first attempt to trade some of the flexibility of classes for faster execution, called *strong mode*, has been proposed by the V8 development team. It is described in the following section.

2.2 JavaScript Strong Mode

The now defunct *strong mode* project from Google [23] had the dual goals to enforce static guarantees and to speed up execution. Strong mode defined a subset of JavaScript in order to forbid patterns that typically defeat compilers or that yield unpredictable performance. For instance, in strong mode, all variables had to be declared, functions have to be invoked with a number of arguments compatible with the actual number of parameters declared, and elements can not be removed from arrays.

Some restrictions were specifically designed to improve class implementation: properties could not be deleted, literal objects could not have duplicate properties, class instances were sealed after the constructor, class declarations were immutable bindings, and class constructors had to return the created object. Eventually, Google abandoned strong mode for various reasons [24]. Some of them were related to classes.

- "*Locking down classes: This was our biggest hope and the biggest failure*". For the sake of interoperability, strong mode classes were allowed to inherit from regular classes and vice versa. The team was unable to find a semantics that would work consistently and harmoniously with class inheritance and they eventually decided to give up on locking classes.
- "*ECMAScript6 classes lack property declarations*". Lacking property declarations made locking classes semantics complex because the set of properties could not be determined at creation time.
- "*ES6 performance s***!*". At that time, ECMAScript6 implementations were not yet delivering good enough performance when compared to previous JavaScript versions. Since strong mode required ECMAScript6, using strong mode resulted in a significant slowdown that was not compensated by the advantages of strong mode.
- "*Implementation complexity*". Since strong mode changes encompassed almost the entire language, its implementation was large and complex.

Since the strong mode proposal, JavaScript has evolved in several dimensions that allow us to reconsider class locking. Firstly, the class declaration itself has evolved. Optional instance properties in classes are now possible. They can even be made private, which implies that only the methods of that class itself can access them. Secondly, the language now supports modules so that it is possible to export immutable bindings, such as classes and functions.

Inspired by the strong mode endeavor and taking advantage of the latest JavaScript developments, we have designed *sealed classes*. Their design shares several constraints and restrictions with strong but relaxes others to support a gradual adoption and a smoother blending with regular classes. Thus, we believe that they can support the performance and reliability of strong mode classes while retaining most of the flexibility of regular classes. Sealed classes are presented in the next section.

3 Sealed Classes

Sealed classes are classes that are deprived of a minimal set of freedoms which, while retaining most of their flexibility, allows AoT JavaScript compilers to implement them using faster and more predictable techniques, much like class-based languages such as Smalltalk, Java, or C++. Sealed classes are syntactically similar to ordinary classes, except that their declaration is prefixed by the annotation "`// @sealed`". They are even so syntactically similar to classes that most class examples, including MDN examples, can be adapted by merely inserting the annotation on the lines preceding their declarations. All the examples of Section 2, for instance, are valid sealed classes.

Sealed classes are carefully designed so that they can be adopted gradually by programmers. For that, they are compatible with the rest of the language. They can be mixed with other data structures, including regular classes, they can be exported and imported from modules, and regular classes can inherit from sealed classes.

☛ *Sealed classes have an "erasure dynamic semantics". Ignoring the annotations of sealed classes does not change the semantics of correct programs. In consequence, any current JavaScript engine can execute correct programs using sealed classes.*

This is a central argument for adoption. This is also the reason why this paper does not include a formal semantics. The dynamic semantics of sealed classes are those of normal JavaScript classes.

Sealed classes are designed based on the observations and the results of experience presented in Section 2.1. They aim to correct the main slowness of regular classes by allowing AoT compilers to:

- use faster and more predictable techniques for implementing instance property access and method invocation in presence of polymorphism;
- use faster and more predictable type predicates;
- use faster class constructors.

Sealed classes are characterized by a set of constraints that, taken together, allow for improvements in those three dimensions. We will refer to each constraint individually when presenting their implementation, using the symbol ☆ to designate constraints that are enforced statically by the compiler, and ✪ those that require compile-time and run-time enforcements.

The key change is to allow the compiler to compute an accurate memory map of each instance, including the memory size and the offsets at which properties are stored. This is only possible if the compiler knows the whole class hierarchy. This requires that sealed

classes inherit only from other sealed classes and that the superclass hierarchy is known at compile-time. A static class hierarchy is sufficient to allow the compiler to set offsets for all properties declared in the class but it is insufficient to handle dynamically added properties on a per-instance basis, either in constructors or in the rest of the program. If this were allowed, properties declared in the class and those added dynamically would perform differently, resulting in unpredictable performance for non-expert programmers. Since sealed classes are intended to improve performance *and* predictability, we decided not to allow dynamic property (addition or removal) in sealed class instances. With this constraint, a compiler knows where all properties are stored, it can optimize their accesses, and it can raise an error if an undeclared property is accessed.

☆ **C1** *A sealed class inherits from another sealed class or from the value* `null`.
✪ **C2** *Sealed class instances are not extensible; their properties have to be declared in the class; their properties cannot be removed.*

C1 is compatible with separate compilation because the new JavaScript modules allow the compiler to know the list of imported bindings. **C1** does not prevent sealed classes from being declared and used in `eval` code because by the time a sealed class is to be evaluated, its super class already exists. In contrast, **C1** prevents an evaluated sealed class from being used as the super class of a compiled sealed class.

C2 is enforced statically by the compiler when it can track the type of an object and when it knows it is a sealed class instance. It is also enforced dynamically when either the types are unknown or when dynamic property names are used. For instance, in an expression such as "`o[`*expr*`]`", since *expr* is unknown at compile-time, a dynamic check is needed to verify that if `o` is a direct instance of a sealed class, the value of *expr* is a property of `o`.

Like sealed instances, prototypes of sealed classes are immutable and non-extensible. This is for two reasons. First, if they were extensible, the compiler could not raise errors when accessing properties that are not in the object, because the properties might be in the prototype chain. Second, if they were mutable, method invocations could not be statically resolved and would require techniques similar to inline caches, leading to unpredictability as presented in Section 2.1.

✪ **C3** *Sealed class prototypes are not extensible and they are immutable.*

C3 requires the same runtime support as **C2**.

In JavaScript, a class can be introduced in an expression or in a declaration. The constraint **C1** would prevent a sealed class expression from being the super class of another sealed class, which we believe makes sealed class expressions of little use. Therefore, for simplicity and consistency, sealed class expressions are not allowed.

☆ **C4** *Sealed classes can only be used in declarations and they are immutable.*

C1 and **C2** allow the compiler to efficiently compile property accesses when it knows that the type of the object is a sealed class, but the flexibility of plain classes makes it difficult for an AoT compiler to infer exact types. To improve the accuracy of static type analysis, two constraints are added, without which the compiler would have very little opportunity to optimize sealed class property accesses.

☆ **C5** *The constructors of sealed classes must return the freshly created instance or* `void`.
✪ **C6** *The methods of sealed classes can only be used on instances of that class or its derived classes.*

C5 guarantees that the type of a "`new SC()`" is an instance of *SC*, if *SC* is a sealed class. Note that this property does not hold if *SC* is a regular class. **C5** allows instances to be explicitly returned by the constructor for compatibility with existing code that uses JavaScript classes. **C6** guarantees that inside a method of a sealed class, the type of `this` is the sealed class. Note that this constraint does not apply to regular classes. In theory, **C6** requires type checks on the input of each method. In Section 4 we present an implementation technique that, in practice, almost completely removes them.

In the next section we discuss the implementation of sealed classes and we show how the 6 constraints that characterize them allow a compiler to exploit more efficient and more predictable techniques inspired by those of class-based programming languages. The overall performance benefit of these constraints is studied in Section 5.

4 Implementation

This section presents the main aspects of the implementation of sealed classes in the hopc AoT compiler. It shows how property access and type checking are implemented without inline caches by adapting techniques from class-based languages. It starts with the implementation of methods, and it ends with the creation of instances.

4.1 Object Representation & Properties

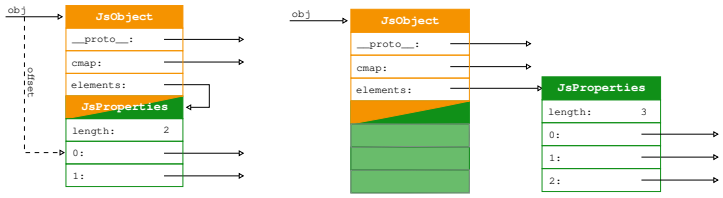
The constraints ☆ **C1**, ✪ **C2**, and ✪ **C3** allow the compiler to build complete maps of sealed class instances. Thus, for a property access "`obj.prop`", if the compiler knows that `obj` is a sealed class instance, it knows where `prop` is stored in `obj` and it no longer needs inline caches or guards, with a double benefit: dynamic checks are removed, and performance no longer depends on the degree of polymorphism (see Figure 2). When `obj` is known to be a sealed class instance, "`obj.prop`" is compiled as "`*(obj+`*k*`)`", where *k* is a constant known at compile time. ✪ **C5** and ✪ **C6** improve the precision of the type analysis. The first one gives the compiler the opportunity to track newly created sealed instances. The second one gives precise types to "`this`" in the methods of sealed classes.

Despite its apparent simplicity, this compilation scheme has subtle implications for the representation of objects. Indeed, since sealed class instances must be compatible with the rest of the runtime system, the representation of their properties must be compatible with those of regular objects, and thus with inline caches. In the remainder of this section we present the modifications applied to hopc for this purpose. First, we recall how hopc represented objects and implemented property accesses before this work [26].

Hopc uses a dual representation for objects. They are created with pre-allocated *inlined* properties. If, later, a new property is to be added, a new vector, large enough to contain the previously inlined properties and the new one, is allocated and the object representation switches from *inlined* to *extended*. The memory space previously used for the inlined properties is left unused. Figure 4 shows these two states of an object.

For each object constructor, the compiler estimates the size of the objects it creates. When an object is extended, this estimated value is incremented and the next time the constructor creates an object, it will create it larger [6]. This technique minimizes the waste of memory chunks because it reduces the number of times an object is represented as an extended object. It also maximizes the number of inlined property accesses.

Inline access is implemented as "`*(obj+cache`$_{index}$`)`", which is similar to accessing a sealed class instance property. As extended properties use one extra level of indirection their implementation requires an extra memory read: "`obj->elements[cache`$_{index}$`]`". In both



(a) inlined object. **(b)** extended object.

Figure 4 The two states of an object, inlined (4a) or extended (4b).

cases, the index where the property resides has to be discovered first. This is the purpose of hidden classes and inline caches presented in Section 2.1. In order to take advantage of inlined properties, hopc distinguishes them when checking inline caches:

```
1  if (obj->hclass == cache.iclass) {
2     val = *(obj+cache.index);                  // inline property
3  } else if (obj->hclass == cache.hclass) {
4     val = obj->elements[cache.index];          // extended property
5  } else {
6     val = cacheReadMiss(obj, "prop", &cache);  // cache miss
7  }
```

Hopc uses an encoding that makes inlined objects compatible with extended objects. When an object is inlined, its internal pointer `elements` points to the inlined chunk. When an object is extended, this pointer is updated to point to the newly allocated chunk of memory. Using a compatible representation for inlined and extended properties minimizes polymorphic inline caches, because when both encodings are used in the same location, the inline cache can be set to the extended object's hidden class, which also works for inline properties, and no cache misses will occur.

The previous inline cache sequence efficiently handles monomorphic accesses, *i.e.,* when all objects used at that source location share the same hidden class or when hidden classes distinguish between inline or extended properties, but it does not efficiently handle polymorphic accesses that occur when the objects accessed from a specific location have different types. As described in a previous study [27], to cope with polymorphism, hopc uses a long code sequence of successive tests inspired by earlier work [15]:

```
1  if (obj->hclass == cache.iclass) {
2     val = *(obj+cache.index);                                  // inline property
3  } else if (obj->hclass == cache.hclass) {
4     val = obj->elements[cache.index];                          // extended property
5  } else if (obj->hclass == cache.aclass) {
6     val = obj->elements[cache.index](obj);                     // accessor property
7  } else if (obj->hclass == cache.pclass) {
8     val = cache->owner->elements[cache.index];                 // prototype property
9  } else if (cache.vindex < obj->hclass->vtableLen) {
10    val = obj->elements[obj->hclass->vtable(cache.vindex)]; // polymorphic access
11 } else {
12    val = cacheReadMiss(obj, "prop", &cache);                  // cache miss
13 }
```

Unfortunately this implementation of inline caches is inefficient for class-based object-oriented programming, because instances of a class and instances of a super class are associated with different hidden classes (see Section 2.1). Therefore, when accessing a property from a method of the super class, the slow path on line 10 is used. This is a major problem suffered by JavaScript classes, which is solved when the compiler knows that `obj` is a sealed class instance, because in that case the access to the property is a mere memory access without

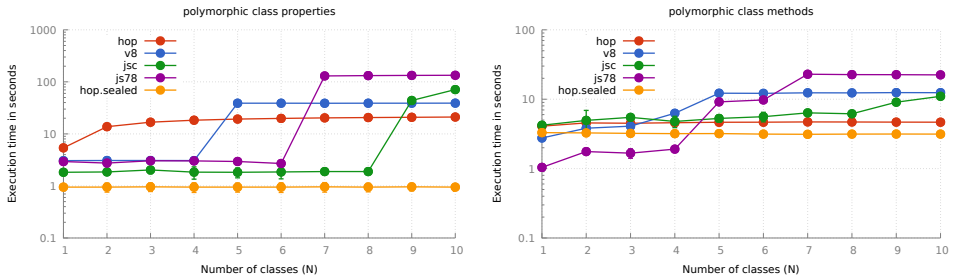
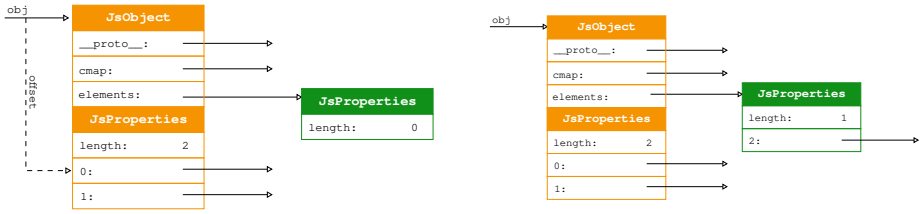


Figure 5 Impact of polymorphism on property accesses and method invocations for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), hopc (3.6.0-pre1), and hopc.sealed. Horizontal axis is the number of classes involved, aka the degree of polymorphism. The vertical axis is the execution time. Lower is better. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.



(a) inlined object.

(b) extended object.

Figure 6 The revised two states of an object, inlined (6a) or external (6b). Direct instances of sealed classes are always represented as inlined objects (6a). Other JavaScript objects, including instances of regular classes, whether their class inherits from a sealed or regular class, are represented as extended objects (6b).

any prior testing. This is the most substantial benefit of sealed classes; see the measurements in Figure 5. Sealed classes allow hopc to outperform all the other systems and allows hopc execution time to be independent of the degree of polymorphism. Note, however, that this experiment considers the best possible situation where the compiler can infer all types of all objects accessed and can then optimize all property accesses. In practice, the benefit for real programs is smaller because only a fragment a property accesses are optimized and inline caches must still be used in the majority of property accesses (see Section 5).

Despite the similarities with normal objects, sealed classes required an important change for supporting subclassing by regular classes. Class instances whose super class is a sealed class must have their properties that belong to the sealed class inlined because they need to be compatible with a sealed class representation and they also need to be extendable in order to behave as regular JavaScript objects. None of the representations of Figure 4 are compatible with this constraint. This forced us to change the hopc object implementation.

With the new encoding, objects retain their inline properties throughout their lifetime, and when they are extended, the newly allocated chunks of memory contain only additional properties (see Figure 6). This change has pros and cons. On the positive side, it reduces memory allocation and increases inlined accesses. On the negative side, it potentially increases polymorphic accesses. With the representation of Figure 4, an inlined object can be treated as an extended object because in both cases, `elements` correctly points to the property array. Then, at an access point, if both inlined and extended objects are used, the inline cache ends up being armed with the extended schema and all accesses are treated the same. With the

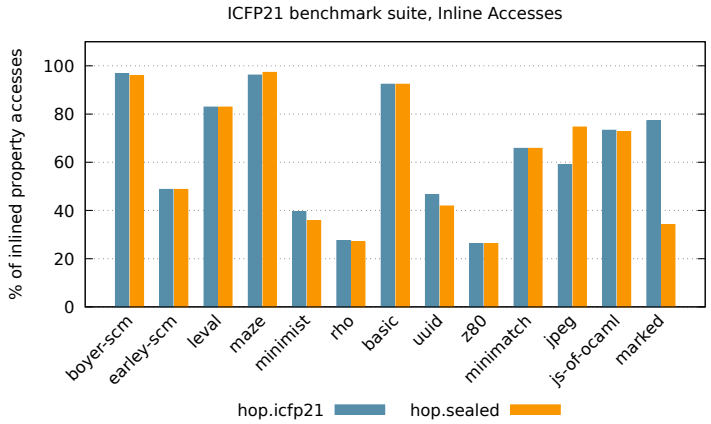


Figure 7 Percentage of inlined accesses for unmodified and modified hopc versions. Higher is better. Linear scale used.

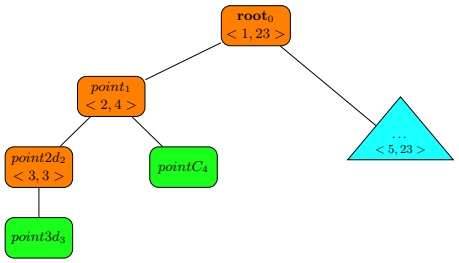


Figure 8 Cohen class numbering for single inheritance type checking.

new encoding, the inlined and extended representations are incompatible and if both are used at the same access point, they are considered as two different hidden classes, meaning it would be handled as a polymorphic access.

To measure the impact of this change, we compared the number of accesses using fast inlined property accesses for the unmodified and modified hopc versions. This experiment reuses the benchmark suite of our earlier artifact [26]. The results are presented in Figure 7. We observe that the new implementation has a marginal impact on program behaviors. The most significant impact is observed for the marked test since the new version seems to use about 40% fewer inlined properties. On a closer examination, it appears that the better behavior of the former hopc version is due to a different heuristic for allocating objects. The old version allocated them with provisional empty slots. The new version allocates them truly empty. Apart from this difference both versions behave similarly and do not show any significant performance difference.

4.2 Type Checking

Two main techniques for implementing type checking of class-based single inheritance prevail: *Cohen numbering* [7] and using an *inheritance vector*. Cohen's numbering is based on the observation that the entire class hierarchy forms a tree. The classes that are the nodes of this tree are numbered in a depth-first traversal and each class stores the class numbers of its left-most and right-most direct children (see Figure 8). To check that an instance i belongs to class $\mathcal{C}$ is done by checking that the class number of i is in the Cohen range $[\mathcal{C}_{min}..\mathcal{C}_{max}]$:

```
int isa(obj_t obj, class_t class) {
  int i = obj->class->number;
  return (i >= class->subclass_min) && (i <= class->subclass_max);
}
```

This is fast (but not as fast as the simple pointers comparison of the hidden class test) but this technique is not well suited to systems where classes are added dynamically because each addition requires renumbering the entire class hierarchy. As hopc compiles modules separately and supports dynamic loading of modules Cohen numbering is not well suited.

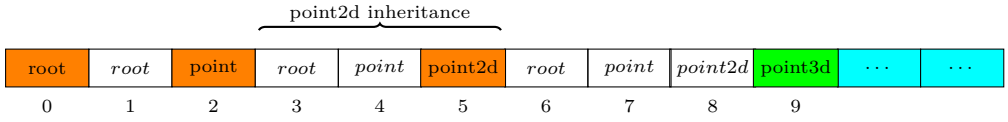
The second classical technique to implement type checking is to provide each class with a vector of its ancestors. Verifying that an instance i belongs to the class $\mathcal{C}$ is done by comparing the value of the ancestors vector at the index corresponding to the depth of $\mathcal{C}$ and $\mathcal{C}$ itself. Here is a possible implementation:

```
int isa(obj_t obj, class_t class) {
  class_t oclass = obj->class;
  return (oclass->depth <= class->depth) && (class->ancestor[class->depth] == class);
}
```

This involves five memory accesses and two integer comparisons but the dynamic addition of a class does not require any slow operation. Moreover, when the compiler statically knows the class being checked, it can compute its depth, *i.e.,* its number of ancestors, and use a faster variant:

```
int isa_depth(obj_t obj, class_t class, int depth) {
  class_t oclass = obj->class;
  return (oclass->depth <= class->depth) && (class->ancestor[depth] == class);
}
```

We use a slighty improvement of this technique by removing the range check and by removing one memory access. This required another change to the hopc object representation. Instances no longer contain a pointer to their class but rather its index. A global vector (`CLASSES`) contains all classes and their ancestors. In the vector `CLASSES`, the index of a class $\mathcal{C}$, which we notate as $\mathcal{C}_i$ contains its highest super class, *i.e.,* the root of the class hierarchy. The direct super class of $\mathcal{C}$ is stored at index $\mathcal{C}_i - 1$. Considering the class hierarchy of Figure 8 where the depth of `point2d` is 2 (it has two ancestors), if $\texttt{point2d}_i = 3$, then `CLASSES` looks like:



Checking if an instance o belongs to a class $\mathcal{C}$ is done with:

```
int isa(obj_t obj, class_t class) {
  int index = obj->class_index;
  return CLASSES[index + class->depth] == class;
}
```

This involves 4 memory accesses (as `CLASSES` is a global variable, accessing it requires a memory access).

Let us suppose that we check an instance of class $\mathcal{I}$ against a class $\mathcal{C}$. If $depth(\mathcal{C}) \leq depth(\mathcal{I})$ then comparing the value of $\texttt{CLASSES}[\mathcal{I}_i + depth(\mathcal{C})]$ and $\mathcal{C}$ implements the type check. If $depth(\mathcal{C}) > depth(\mathcal{I})$ accessing `CLASSES` will erroneously access the ancestor list of another class stored after $\mathcal{I}$. However, since $depth(\mathcal{I}) > 0$ the class that will be erroneously accessed is at an index smaller than $depth(\mathcal{C})$. Hence, it cannot be $\mathcal{C}$ because $\mathcal{C}$ is always

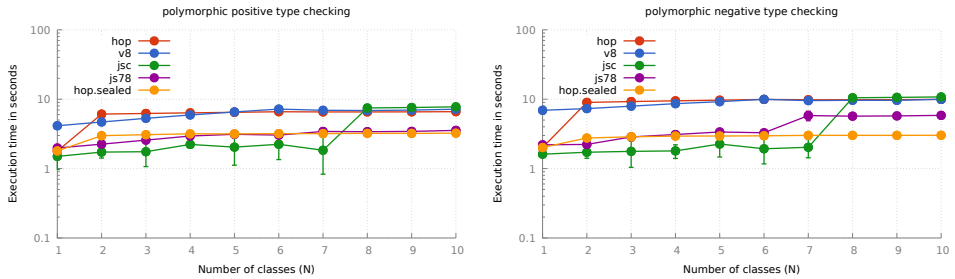


Figure 9 Impact of polymorphism on type predicates (`instanceof`) for V8 (9.4.146), Jsc (4.0), Js78 (C78.4.0), and hopc (3.6.0-pre1). The figure left-hand-side shows values when type tests succeed, the right-hand-side, when they fail. Logarithmic scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

stored at $\mathcal{J}_i + depth(\mathcal{C})$ in `CLASSES`. Thus, boundary checks can be avoided if enough spare space is reserved at the end of `CLASSES` (this space size is the biggest inheritance depth). Finally, assuming that `class->depth` is statically known, the predicate is simplified as follows:

```
int isa_depth(obj_t obj, class_t class, int depth) {
  int index = obj->class_index;
  return CLASSES[index + depth] == class;
}
```

This routine replaces the default implementation of `instanceof` on a type check cache miss. It greatly minimizes the negative impact of polymorphism on type checking as shown in Figure 9.

4.3 Methods

Constraint C6 requires that sealed methods be passed only instances of this class. To do this, hopc inserts dynamic checks at the beginning of each method that are compiled as:

```
class C {
   M(a0, a1, ...) {
      if (this instanceof C) {
         if (!isProxy(this)) {
            return Munsafe(a0, a1, ...);
         } else {
            ...this:Proxy, body optimized for size...
         }
      } else {
          ...raise an error...
      }
   }
   Munsafe(a0, a1, ...) { ...this:C, body optimized for speed... }
}
```

Inside $\mathcal{M}_{unsafe}$ the compiler knows that `this` is an instance of $\mathcal{C}$ so it optimizes property accesses, as we have already seen, but intuitively wrapping $\mathcal{M}$ also has the potential downside of requiring additional dynamic checks and extra function calls. This seldom happens.

Consider the expression "`obj.`$\mathcal{M}$`(e0, e1, ...)`". If the static type of `obj` is a sealed class, then the compiler rewrites the expression to "`obj.`$\mathcal{M}_{unsafe}$`(e0, e1, ...)`". Otherwise, it generates an inline cache sequence, for which the routine dealing with the cache miss will check whether `obj` is a sealed class. If it is, it will fill the cache with $\mathcal{M}_{unsafe}$. If it is not, it will fill it with $\mathcal{M}$, which will eventually check if it is a proxy, for which it will use a slow but compact implementation. Otherwise, it will raise an error and performance will not matter anymore. Holze et al. [15] first proposed this technique.

Constraint ✪ C3 allows the compiler to compile expressions "`obj.`$\mathcal{M}_{unsafe}$`(e0, e1, ...)`" without inline caches. Since prototype objects of sealed classes are immutable, the compiler statically knows all the methods of the sealed classes and groups them in a static vector. The vector of methods of a sealed class $\mathcal{C}_1$ that inherits from another class $\mathcal{C}_0$ is the concatenation of the $\mathcal{C}_0$'s inherited methods patched with overloaded methods and $\mathcal{C}_1'$s newly introduced methods. The hidden classes of sealed instances point to this vector. In pseudo C code, a sealed method invocation is then compiled as follows:

```
obj->hclass.methods[INDEX](obj, a0, a1, ...)
```

Finally a call "`super.`$\mathcal{M}$`(e0, e1, ...)`" inside a class $\mathcal{C}$ is implemented even more efficiently. Constraints ✪ C2, ✪ C3, and ✪ C6 allow the compiler to generate a direct call to the method $\mathcal{M}_{unsafe}$ of C's super class which, thanks to constraint ☆ C1, is statically known. Thus a call to a `super` method is either inlined or compiled as a direct jump.

4.4 Instance Creation

Sealed classes constraints drastically simplify instance creation. ☆ C4 allows the compiler to know which sealed class is involved in an allocation "`new` $\mathcal{E}$`(...)`". ☆ C1 and ✪ C2 allow the compiler to know the memory size of the instances to allocate at compile time. ☆ C1 allows the compiler to know the whole chain of super constructors to call when allocating an object. ✪ C2 allows the compiler to know the hidden class to associate with sealed instances, which is required for inline cache compatibility. Let us consider the following class hierarchy:

```
1  // @sealed
2  class baseclass {
3     a0;
4     constructor(a0) {
5        this.a0 = 0;
6     }
7  }
8  // @sealed
9  class subclass1 extends baseclass {
10    a1;
11 }
12 // @sealed
13 class subclass2 extends subclass1 {
14    a2;
15    constructor(a0, a1, a2) {
16       super(a0);
17       this.a1 = a1;
18       this.a2 = a2;
19    }
20 }
21 new subclass2(0, 1, 2);
```

At line 21, the compiler knows that `subclass2` instances have 3 properties and it knows the hidden class to which instances are associated, as well as their prototype. Thus, the object allocation is as follows (see Figure 6 for an explanation of the various fields):

```
1  JsObject *this = (subclass2)GC_malloc(sizeof(JsObject) + 3 * sizeof(JsObject *));
2
3  this->__proto__ = subclass2_PROTOTYPE; // initialize the instance prototype object
4  this->cmap = subclass2_HIDDENCLASS;    // initialize the instance hidden class
5  this->elements = 0L;                   // no extended property
6  this->length = 3;                      // 3 inlined properties
7  subclass2_CTOR(this, 0, 1, 2);         // invoke the constructor
```

The `subclass2` constructor is compiled into:

```
1 void subclass2_CTOR(subclass2 *this, JsObject *a0, JsObject *a1, JsObject *a2) {
2    {
3       subclass1 *super = (subclass1 *)this; // inlining of the baseclass constructor
4       *(&(super->inline0) + 0 * sizeof(JsObject *)) = a0; // set property this.a0
5    }
6    *(&(this->inline0) + 1 * sizeof(JsObject *)) = a1;      // set property this.a1
7    *(&(this->inline0) + 2 * sizeof(JsObject *)) = a2;      // set property this.a2
8 }
```

In this example, since there is no constructor declared in `subclass1` and the constructor of `baseclass` is short and simple, it is inlined in the constructor of the subclasses (line 2). The compiler knows that it is not necessary to provide a data structure for `new.target` because no constructor of this class hierarchy uses it. Except for the initialization of the fields specific to JavaScript for the prototype object and the hidden class, this implementation of object allocation is similar to those of static languages.

4.5 Final Consideration

In this section we have detailed the changes made to the hopc AoT compiler to support sealed classes. These changes were necessary to enforce the constraints given in Section 3 and they all contribute to improve performance. While some changes are subtle and require fine tuning, such as the new implementation of type checking (Section 4.2), the overall size of the changes is minimal. It took less than 1,000 lines in the compiler implementation and a few hundred lines in the runtime system.

5 Sealed Class Performance Evaluation

This section contains the performance evaluation and analysis. First, it presents the benchmarks used. Next, it presents a comparison of the performance of sealed versus regular classes. Finally, it analyzes the reason for the performance differences.

5.1 Benchmarks

JavaScript classes were introduced in ECMAScript 6. Although they are more than 5 years old, they are hardly used in standard JavaScript benchmarks. Only three JetStream2 tests use them: basices2015, flightplanner, and unipoker (that we modified to use only ascii strings). We used them and other tests from different sources. First, we have created class-based versions of Octane DeltaBlue and Richards. This task was straightforward because both of these programs were originally class-based Smalltalk programs, which were translated into JavaScript by introducing prototypes to simulate class declarations and inheritance. We merely performed the reverse translation by restoring classes. When possible we used private fields instead of public fields to implement class instance properties. Similarly, we adapted the maze test used to evaluate hopc performance in a previous publication [26]. We used the two benchmarks from the StrongType study [22] that use classes (crypto-class and raytrace-class). To complete the test suite, we have ported to JavaScript the Python benchmarks of the PyPerformance test suite that use classes [29]. The similarities between the two languages allowed us to use a straightforward line-by-line translation. Two programs required extra attention. First, the bm_hexiom.py test uses the `__getitem__` so-called *magic method* to expose the values of internal instance properties. We have produced two versions of this test. The first one, hexiom where the accesses via `__getitem__` have been replaced with accesses to an array of values and the second one, hexiom-proxy, where `__getitem__` is implemented using JavaScript proxy objects. This is the only test that uses JavaScript

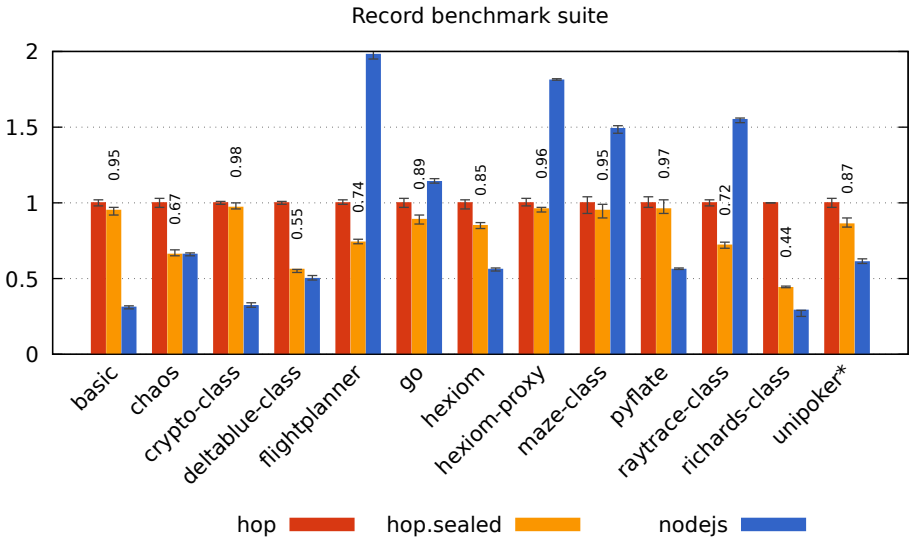


Figure 10 Sealed classes performance and V8 (9.4.146) class performance relative to Hop (3.6.0-pre1) classes performance. Lower is better. Linear scale used. Measures collected on Linux 5.14 x86_64, powered by an Intel Xeon W-2245. Each test and configuration has been executed 30 times.

proxies. The second test that required attention is bm_pyflate.py. This program uses 64bit bitwise operations which are not supported by JavaScript. We have implemented these operations using the recent JavaScript BigInt arithmetic.

5.2 Sealed Classes Performance

Figure 10 presents the overall performance evaluation. It reports the score of the unmodified hopc compiler, the score of the version supporting sealed classes (*hop.sealed*), and, for a neutral comparison, the performance of V8.[1]. We use this mainstream JIT compiler to show that on this set of benchmarks, hopc delivers a competitive performance and that the results we report in this experiment are realistic. To minimize the penalty imposed by JIT compilation we have calibrated the executions so that they last between 5 and 10 seconds.

Figure 10 shows that sealed classes are beneficial to almost all tests. The speedup can even go as high as 56% for the richards-class test. As presented in Section 4.3, the method bodies of sealed class are compiled twice: a speed-optimized version when `this` is an instance of the sealed class, and a code-size-optimized version when `this` is a proxy object. Despite this code duplication, Figure 11 shows that the speedup is not counterbalanced by a significant increase in code size. Indeed, the speed-optimized version is more compact than the implementation based on inline caches, and the difference between the two sizes is approximately the size of the compact version generated for proxy objects.

Simplifying the inline cache sequence reduces code size and accelerates execution because, by removing tests, it reduces the number of executed instructions and it minimizes the pressure on the processor's branch predictor, as we will see when studying the go test below. Figure 12 shows the percentage of simplified inline caches. Note that unipoker mostly uses arrays with so few object accesses that the measure is not relevant for that test.

[1] We observed that V8 executes faster if class properties are all replaced with instance properties added in the constructors. However, as these are an essential component of sealed classes and an official addition to JavaScript we have kept them for our evaluation.

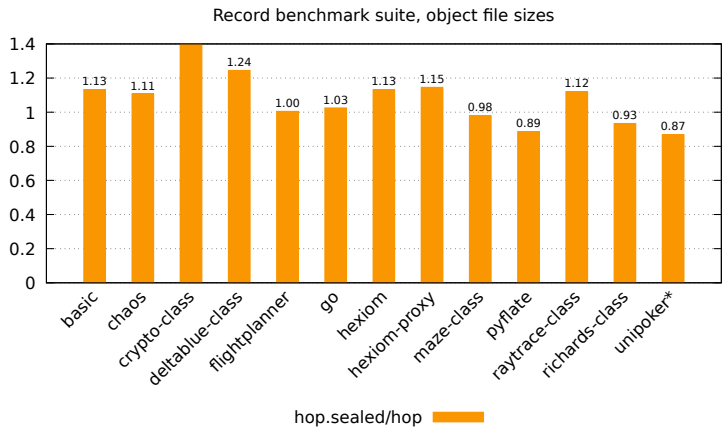


Figure 11 Sealed classes code sizes relative to Hop classes code sizes. Lower is better. Linear scale used.

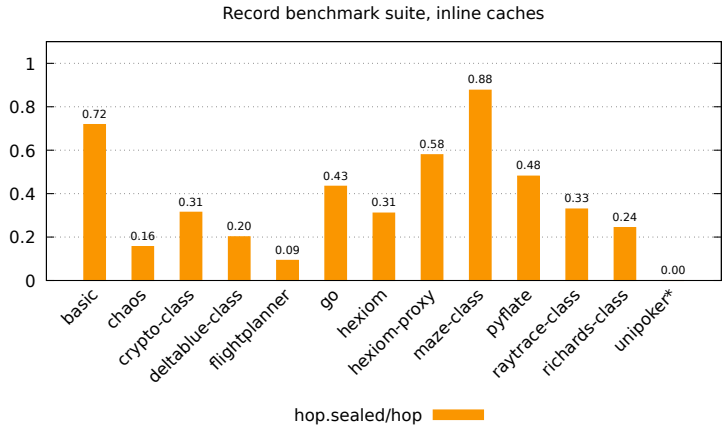


Figure 12 Sealed classes number of inline caches relative to Hop classes inline caches. Lower is better. Linear scale used.

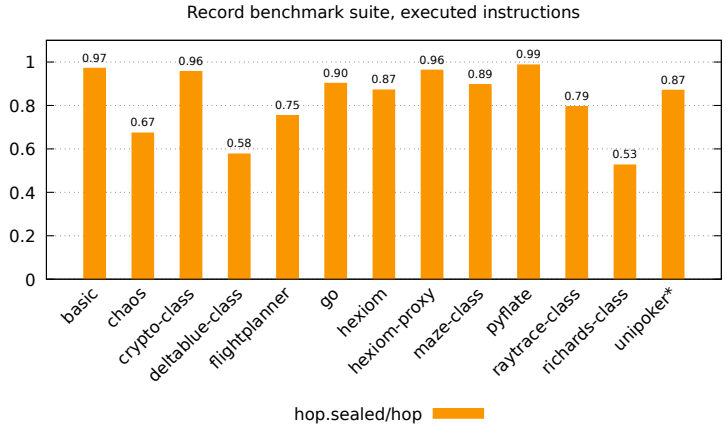


Figure 13 Comparison the number of executed instructions for classes and sealed classes. Lower is better. Linear scale used.

hop execution times (in ms)

```
564.5 (24%) GC_mark_from
230.1 (15%) GC_add_to_black_list_normal
 93.5 (10%) hashtable-get
 59.8 (08%) &@%kwhile1286
 50.4 (07%) GC_header_cache_miss
  6.0 (02%) GC_mark_local
  2.5 (02%) GC_malloc_kind
  2.2 (02%) &eqtest
  2.0 (01%) js-map-get
  1.6 (01%) hashtable-contains?
```

hop.sealed times (in ms)

```
447.3 (21%) GC_mark_from
175.6 (13%) GC_add_to_black_list_normal
120.8 (11%) hashtable-get
 92.2 (10%) &@%kwhile1511
 35.8 (06%) GC_header_cache_miss
  4.2 (02%) GC_malloc_kind
  4.2 (02%) GC_mark_local
  3.7 (02%) js-map-get
  3.6 (02%) &eqtest
  2.3 (02%) hashtable-contains?
```

Figure 14 Profiling information for the basic-es2015 benchmark. Functions are reported along with the cumulative time spent executing them and the percentage of the overall execution they are responsible for. The left hand-side is the class-based hop version. The right hand-side is the modified version supporting sealed classes. The profiling values are gathered with the Linux `perf` tool.

Unexpectedly, we observe that there is no direct correlation between the static number of inline cache simplifications and the performance improvement. For instance, the pyflate test does not benefit from any significant speedup when using sealed classes, even though half of inline caches were replaced by direct property accesses without testing. Using the Linux `perf` tool [13] we measured the impact of removing the type checks of inline caches type on the number of executed instructions. This is reported in Figure 13. Again, we do not observe a strong relationship between the simplification of the generated code and the number of instructions actually executed by the processor. More generally, while all measurements (Figures 10-13) confirm the speedup of sealed classes, we cannot establish a systematic correspondence between the measured runtime reduction and the other experiments. Thus, in the following sections we conduct an in-depth analysis of some tests.

5.3 Basic.js

Basic is part of the JetStream2 suite. It implements an interpreter for the Basic ECMAScript-2015 programming language. Sealed classes do not improve basic speed. The performance difference between the two versions is below the precision of the observation. The code size is only reduced by 3% and although the number of inline caches needed to compile the program is reduced by 28%, the sealed class version does not execute fewer machine instructions than the class version.

Figure 14 shows an excerpt of the Linux `perf` profiling of the two basic versions. The numbers are the milliseconds spent in each function and the percentage of the overall execution time this corresponds to. Functions prefixed with '`&`' are compiled JavaScript functions. The other functions are part of the hopc runtime system. The two compilation modes do not use the very same naming conventions for class and sealed class methods but the correspondence is fairly straightforward.

The execution time is mostly dominated by the garbage collector and by the implementation of JavaScript maps. The only client function that plays a significant role in the execution is the function named `&@kwhile1560`, as shown in the profiling report.

The test exercises JavaScript generators extensively because all the nodes of the abstract syntax tree representing the Basic source programs are encoded as generators. On an x86_64 platform, the completion of the test allocates about 6.8GB. About half of the allocated objects are arrays and the other half are generators. These allocations are unrelated to classes or sealed classes or field accesses. For instance, the hopc memory profiler reports that 23% of the overall allocations of arrays are used to prepare a function using the `apply` form.

hop execution times (in ms)

```
1029.8 (32%) GC_mark_from
 442.3 (21%) GC_add_to_black_list_normal
  90.4 (10%) GC_header_cache_miss
  30.5 (06%) GC_malloc_kind
  13.9 (04%) &Spline.__call__
   8.8 (03%) GC_mark_local
   2.7 (02%) &GVector.linear_combination
   1.5 (01%) GC_find_header
   1.3 (01%) GC_allochblk_nth
   0.8 (01%) &@GVector%CTOR
```

hop.sealed times (in ms)

```
1103.6 (33%) GC_mark_from
 492.4 (22%) GC_add_to_black_list_normal
 101.6 (10%) GC_header_cache_miss
  25.1 (05%) &__call__%%R@Spline
  24.4 (05%) GC_malloc_kind
  11.2 (03%) GC_mark_local
   1.7 (01%) GC_find_header
   1.5 (01%) GC_allochblk_nth
   1.0 (01%) &#transform_point@Chaosgame
   0.7 (01%) __pthread_mutex_trylock
```

Figure 15 Profiling information for the chaos benchmark.

The hopc compiler compiles generators and `yield` expressions using a CPS transformation. The `&@kwhile1560` function, which is by itself, responsible for 10% of the execution, is the compilation of a generator `while` loop. This is a regular function, not a class method, so the introduction of sealed classes does not impact its compilation. This explains why the main impact of sealed classes on basic is the code size reduction more than the execution speed.

A similar situation occurs for the go benchmark. Although the original Python version uses classes, there is no polymorphism involved and in the class JavaScript version, 99% of the accesses are optimally handled by inline caches. The sealed class version removes a significant number of these inline caches but if this reduces the size of the generated code, it does not accelerate the execution that significantly because the processor branch predictor compensates for these extra tests, as reported by the Linux `perf` report:

hop

```
    57,510.34 msec task-clock
      255,988      context-switches
        9,790      cpu-migrations
        4,636      page-faults
105,678,673,722    cycles
112,477,659,094    instructions
 23,867,169,583    branches
    396,229,049    branch-misses
```

hop.sealed

```
    56,174.36 msec task-clock
      257,539      context-switches
       10,514      cpu-migrations
        4,669      page-faults
101,498,166,407    cycles
101,682,162,341    instructions
 20,985,760,131    branches
    392,313,687    branch-misses
```

We observe that the class based version executes 10% more branches than the sealed class version but the number of miss-predicted branches is almost identical for the two programs.

The same phenomenon is observed for the maze-class test. It uses classes for its data structure but it rarely uses object polymorphism. In the class-based version most accesses are efficiently handled with inline caches. As for go, the main benefit is the code size reduction.

5.4 Chaos.js

Chaos is the line-by-line transcription of the Python bm_chaos.py program. Its execution is dominated by the allocation and reclaiming of objects and boxed real numbers. The GC itself consumes about 60% of the overall execution so there is not much left for sealed classes to optimize. However, one single client function, `Spline.__call__`, is responsible for more than 10% of the overall execution, and this function allocates most of the benchmark class instances (see Figure 15). The sealed class version benefits from the faster allocation schema (see Section 4.4) to out-perform the class version.

5.5 Deltablue-class.js

Deltablue-class is a modified version of the Octane test where prototype chains are replaced with classes. The sealed class based version is significantly faster than the class based version. The acceleration comes to a large extent from the replacement of the inline caches with direct field accesses as visible in Figure 12. The more efficient polymorphic implementation of sealed classes is also an important factor of acceleration for this test. Deltablue-class defines a hierarchy of classes for representing constraints whose base class is defined as:

```
150 class Constraint {
151    strength;
152
153    constructor(strength) {
154       this.strength = strength;
155    }
```

Several classes inherit from `Constraint`, e.g., `UnaryConstraint`, which is defined as:

```
230 class UnaryConstraint extends Constraint {
231    #myOutput;
232    #satisfied;
233
234    constructor(v, strength) {
235       super(strength);
236       this.#myOutput = v;
237       this.#satisfied = false;
238       this.addConstraint();
239    }
```

When an `UnaryConstraint` instance is created, the class constructor invokes the constructor of the superclass line 235. This triggers the execution of the `Constraint` constructor and the execution of line 154. Each `Constraint`'s subclass has a dedicated hidden class, so the assignment of the `strength` property is polymorphic. hopc's inline cache profiler reports that the assignment is executed 3×10^6 times. For 1×10^6 of them, the inline cache has matched a inlined property instance but for 2×10^6, a polymorphic assignment has been executed, which involves using slower hopc vtables [27]. The replacement of classes with sealed classes enables the compiler to generate a code that always uses an inlined property assignment. Overall, the cache profiler reports that the number of polymorphic accesses and assignments drops from 56×10^6 to 8×10^6 (a $7\times$ reduction) when switching from classes to sealed classes.

At line 235 the constructor invokes the constructor of the superclass and at line 238, it invokes the class method `addConstraint`. As `this` is the receiver of the two method invocations, the more efficient compilation of Section 4.3 contributes to accelerate this benchmark.

5.6 Flightplanner.js

Flightplanner is a benchmark of the JetStream2 suite. Sealed classes accelerate it by about 26%. The Linux `perf` report Figure 16 shows that a significant part of the acceleration is due to the simplification of the allocation of the `Leg` object that is inlined in the sealed class execution. The class execution uses slow polymorphic accesses that we observe by the significant part of the execution spent in the function `js-object-vtable-push!`. The sealed class execution only uses direct instance accesses and then never calls this function.

5.7 Raytrace-class.js

Raytrace-class is the class-based TypeScript version of the Octane as described in Richards et al. [22] from which type annotations have been erased. This is one of the few tests that uses inheritance and object polymorphism extensively, and as such, it is one of the benchmarks

hop execution times (in ms)

```
1398.8 (37%) GC_mark_from
  41.5 (06%) &@Leg%CTOR
   9.0 (03%) js-object-vtable-push!
   7.6 (03%) GC_header_cache_miss
   6.9 (03%) GC_malloc_kind
   4.6 (02%) &FlightPlan.resolveWaypoint
   4.4 (02%) GC_add_to_black_list_normal
   3.5 (02%) open-string-hashtable-get
```

hop.sealed execution times (in ms)

```
2026.8 (45%) GC_mark_from
   9.5 (03%) GC_header_cache_miss
   8.4 (03%) GC_malloc_kind
   6.2 (02%) GC_add_to_black_list_normal
   3.6 (02%) GC_allochblk_nth
   3.5 (02%) &<@resolveWaypoint%%R11757>
   2.7 (02%) string-hashtable-get
   2.1 (01%) open-string-hashtable-get
```

Figure 16 Profiling information for the Flightplanner benchmark.

hop execution times (in ms)

```
 485.3 (22%) &Scheduler.schedule
 310.5 (18%) &TaskControlBlock.run
 127.7 (11%) &HandlerTask.run
  92.2 (10%) &TaskControlBlock.isHeldOrSusp
  52.7 (07%) &Scheduler.queue
  21.5 (05%) &TaskControlBlock.checkPriAdd
  16.1 (04%) &IdleTask.run
  13.7 (04%) &WorkerTask.run
  13.2 (04%) &DeviceTask.run
   7.4 (03%) &Scheduler.suspendCurrent
```

hop.sealed execution times (in ms)

```
 621.0 (25%) &schedule%%R@Scheduler
 390.9 (20%) &run%%R@TaskControlBlock
 141.8 (12%) &run%%R@HandlerTask
  62.1 (08%) &queue%%R@Scheduler
  19.1 (04%) &checkPriAdd%%R@TaskControlBlock
  16.0 (04%) &release%%R@Scheduler
  15.1 (04%) &run%%R@IdleTask
  14.3 (04%) &suspendCurrent%%R@Scheduler
  12.9 (04%) &run%%R@DeviceTask
  12.5 (04%) &run%%R@WorkerTask
```

Figure 17 Profiling information for the richards-class benchmark.

that takes full benefit of sealed classes, with a speedup of 28%. Interestingly, we note that StrongScript improves the same benchmark by 22%. For this test, the benefit of the sealed class encoding seems to be similar to that of static type information. We observe a more contrasted situation for crypo-class and richards-class. StrongScript accelerates crypto-class by 6.2% but slows down richards-class by 14% while sealed classes speed them up by 3% and 56%.

5.8 Richards-class.js

Richards-class is the class-based version of the Octane test. It is the benchmark that benefits the most from sealed classes with a reduction of the execution time of about 56%. Figure 17 shows the excerpt of the Linux `perf` profiling of the two versions of the program.

The function `Scheduler.schedule` is where most of the benefit of sealed classes comes from. Its implementation is as follows:

```
1  schedule() {
2     this.#currentTcb = this.#list;
3     while (this.#currentTcb != null) {
4        if (this.#currentTcb.isHeldOrSuspended()) {
5           this.#currentTcb = this.#currentTcb.link;
6        } else {
7           this.#currentId = this.#currentTcb.id;
8           this.#currentTcb = this.#currentTcb.run();
9        }
10    }
11 }
```

This function is favorable for sealed classes because all accesses of the form `this.#`*`private-name`* are compiled as direct property accesses (see Section 4.1). In spite of this significant acceleration, the compiler's type inference is not powerful enough to infer a precise type for the expression `this.#currentTcb.link`, so hopc is not able to generate a fast sealed class method invocation and it cannot perform an efficient inlining of the methods `isHeldOrSuspended`

and `run`. The overall performance is still lagging behind the JIT V8 compiler. A better type inference, maybe one as good as those provided by TypeScript [1] or Flow [5], would significantly improve the hopc's score on this benchmark.

6 Related Work

The literature on efficient techniques for implementing classes, methods, and type checking in object-oriented programming languages is abundant and as old as these languages themselves. Implementing **Smalltalk** efficiently was a major concern [9]. Over the years, techniques have been proposed to combine static and dynamic properties [8], and techniques for implementing multiple inheritance efficiently have become necessary with languages such as **Eiffel** or **C++** [11]. In this work, we reuse these well-known techniques which we have adapted (see Section 4.2) to the JavaScript context. Our contribution is twofold. First, we propose ways to adapt classical implementation techniques designed for class-based languages so that they can cooperate with those designed to implement the dynamic features of prototype-based languages. Second, we identify some minimal restrictions that must be applied to JavaScript to allow compilers to take substantial benefit from these techniques.

Several attempts have been made to enforce stronger static guarantees and to improve the performance of JavaScript. **TypeScript** [17, 1], of course, has paved the way. It has extended JavaScript with classes before they have been integrated into the language. It has supported type annotations and static type inference in order to detect errors at compile time. A gradual type system has also been proposed to enforce runtime type correctness [21]. However, runtime acceleration is not a goal of TypeScript, in part because it compiles to plain JavaScript code and does not propose a native implementation that could take advantage of type information. This is the motivation of the **StrongType** proposal [22] which, on some benchmarks, manages to improve performance.

Retargeting the TypeScript compiler to map classes to sealed classes might be an interesting direction because the programmer interested in static error detection might also be interested in the dynamic error detection that sealed classes allow and the better performance they deliver. In the same vein, other class-based languages that compile to JavaScript, for instance **Scala.js** [10], might benefit from better performance.

Flow [5] is the Facebook JavaScript type checker. Its purpose is similar to that of TypeScript. Since Flow does not need type annotations and claims to infer more types than TypeScript, it might be interesting to try to use its type inference to statically detect classes that would obey the rules of sealed classes in order to automatically seal them for better efficiency. This is a project for future work.

Sealed classes share two motivations with Google's V8 **strong mode** endeavor [23]: enforcing static guarantees and speeding up class implementation. Strong mode defines a subset of JavaScript in order to prohibit patterns that usually defeat compilers or give unpredictable performance. Some restrictions imposed by *strong mode* are also imposed by sealed classes: properties cannot be deleted, class instances are sealed after the constructor, class declarations are immutable bindings, and class constructors must return the created object, but there are also important differences.

- Strong mode classes and regular classes were allowed to inherit from each other but for interoperability reasons, strong mode class instances and regular classes instances could not be mixed without restriction. This proved to be a show stopper for strong mode locking classes. Sealed classes take a different approach. Regular classes can inherit from sealed classes, but sealed classes cannot inherit from regular classes. This obviously restricts the use of sealed classes, but this is mandatory to improve performance while still preserving some degree of dynamicity.

- Strong mode was declared at the module level so that with a module all classes were either strong or *sloppy*. This made it difficult to adopt strong mode for existing code bases because module-level granularity proved to be too granular, making strong mode too much of an all or nothing decision. Instead, classes can be *sealed* on a declaration-by-declaration basis, one at a time, which simplifies the transition.
- The sealing of strong mode instances turned out to be impractical and incompatible with the rest of the language. Here again, sealed classes differ. A sealed class instance is sealed but an instance of a class that inherits from a sealed class is only partially sealed. Only its "*sealed class instance self*" is sealed. Its *regular class instance self* may be extended or reduced. Instances of these classes benefit from a faster implementation for their sealed part and, for their regular class part, they work like any other ordinary object.
- Sealed classes benefit from the recent JavaScript extensions that allow class field declarations, a feature not available when strong mode was proposed.
- Sealed classes are easy to implement and have minimal impact on the components of an existing system. They are very well delimited. In the implementation of the hopc compiler, apart from the modification of the object representation, they had no impact on the rest of the implementation. In total, the implementation of sealed classes represents less than 1,000 isolated lines of code in the compiler and a few hundred lines in the runtime system. In contrast, strong mode came as a whole, and strong classes could not be isolated from the rest of the proposal. The implementation was complex with ramifications in many V8 components.

7 Conclusion

Sealed classes trade a little bit of the dynamicity of JavaScript classes for faster and more predictable execution. All the benchmarks we tested benefit from sealed classes. Some benefit from a code size reduction and others benefit from speedup. Some benefit from both.

Sealed classes are compatible with the rest of the JavaScript runtime system. They can be passed to functions, returned by them, stored in data structures, and they can be used as the super classes of sealed and ordinary classes. Thus, in existing programs, sealed classes can gradually replace those classes that naturally respect the restrictions they impose. Infringements to the rules of sealed classes are detected, so that sealing classes does not present the risk of silently corrupting operational programs.

The dynamic semantics of sealed classes that do not raise errors is identical to that of regular classes. They can therefore already be used by unmodified JavaScript engines, although in this case there is no runtime acceleration. To benefit from this acceleration, we have modified the AoT hopc compiler. We have shown that the average speedup due to sealed classes is of 19% on a variety of programs using classes. This paper has detailed this implementation. It is simple and required less than 1,000 lines of new lines of code for the compiler and a few hundred lines of code for the runtime system. Sealed classes deliver better performance than regular classes *and* they are easy to implement.

References

1 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 257–281, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-662-44202-9_11.

2 Carl Friedrich Bolz. Better JIT Support for Auto-Generated Python Code. https://www.pypy.org/blog/, September 2021.

3 C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Conference Proceedings on Programming Language Design and Implementation*, PLDI '89, New York, NY, USA, 1989. ACM. doi:10.1145/73141.74831.

4 C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, USA, 1989. ACM. doi:10.1145/74878.74884.

5 Avik Chaudhuri, Basil Hosmer, and Gabriel Levi. Flow, a new static type checker for JavaScript, November 2014. URL: https://engineering.fb.com/2014/11/18/web/flow-a-new-static-type-checker-for-javascript.

6 D. Clifford, H. Payer, M. Stanton, and B. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, New York, NY, USA, 2015. doi:10.1145/2887746.2754181.

7 N. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):626–629, 1991. doi:10.1145/115372.115297.

8 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, 1995. doi:10.1007/3-540-49538-X_5.

9 Peter L. Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800017.800542.

10 Sébastien Doeraene. Cross-Platform Language Design in Scala.Js (Keynote). In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, page 1, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3241653.3266230.

11 Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.*, 43(3):18:1–18:48, 2011. doi:10.1145/1922649.1922655.

12 ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. ECMA, 6.0 edition, June 2015.

13 Brendan Gregg. Linux systems performance. https://www.usenix.org/conference/lisa19/presentation/gregg-linux, October 2019.

14 M. Hölttä. Super fast super property access. https://v8.dev/blog/fast-super, February 2021.

15 U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, UK, 1991. doi:10.1.1.126.7745.

16 B. Meurer and M. Bynens. The story of a V8 performance cliff in React. https://v8.dev/blog/react-cliff, August 2019.

17 Microsoft. TypeScript, Language Specification, version 0.9.5, November 2013.

18 Mozilla Developper Network. Classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, November 2021.

19 Mozilla Developper Network. Public class fields. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields, July 2021.

20 F. Pizlo. Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore, July 2020.

21 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676971.

22 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.76.

23 Andreas Rossberg. A Strong Mode for JavaScript (Strawman proposal). URL: https://docs.google.com/document/d/1Qk0qC4s_XNCLemj42FqfsRLp49nDQMZ1y7fwf5YjaI4/view#heading=h.w5az3vf8le5k.

24 Andreas Rossberg. An update on Strong Mode, February 2016. URL: https://groups.google.com/g/strengthen-js/c/ojj3TDxbHpQ/m/5ENNAiUzEgAJ.

25 M. Serrano. Javascript aot compilation. In *14th Dynamic Language Symposium (DLS)*, Boston, USA, November 2018. doi:10.1145/3276945.3276950.

26 M. Serrano. Of AOT Compilation Performance. *Proceedings of the ACM on Programming Languages*, August 2021. doi:10.1145/3473575.

27 M. Serrano and M. Feeley. Property Caches Revisited. In *Proceedings of the 28th Compiler Construction Conference (CC'19)*, Washington, USA, February 2019. doi:10.1145/3302516.3307344.

28 M. Serrano and R. Findler. Dynamic Property Caches, a Step towards Faster JavaScripts Proxy Objects. In *Proceedings of the 29th Compiler Construction Conference (CC'20)*, San Dieo, USA, February 2020. doi:10.1145/3377555.3377888.

29 V. Stinner et al. The Python Benchmark Suite. https://github.com/python/pyperformance, 2021.