Typed Objects in JavaScript

Nicholas D. Matsakis David Herman
Mozilla Research
{nmatsakis, dherman}@mozilla.com

Dmitry Lomov
Google
dslomov@chromium.org

Abstract

JavaScript's typed arrays have proven to be a crucial API for many JS applications, particularly those working with large amounts of data or emulating other languages. Unfortunately, the current typed array API offers no means of abstraction. Programmers are supplied with a simple byte buffer that can be viewed as an array of integers or floats, but nothing more.

This paper presents a generalization of the typed arrays API entitled *typed objects*. The typed objects API is slated for inclusion in the upcoming ES7 standard. The API gives users the ability to define named types, making typed arrays much easier to work with. In particular, it is often trivial to replace uses of existing JavaScript objects with typed objects, resulting in better memory consumption and more predictable performance.

The advantages of the typed object specification go beyond convenience, however. By supporting *opacity*—that is, the ability to deny access to the raw bytes of a typed object—the new typed object specification makes it possible to store objects as well as scalar data and also enables more optimization by JIT compilers.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Web applications are becoming increasingly sophisticated. Advances in JavaScript engines mean that full-fledged applications can often run with competitive performance to their desktop counterparts. Many of these applications make heavy use of *typed arrays*.

Typed arrays are an API—initially standardized by the Khronos Group [13] but now due for inclusion in the next version of the JavaScript standard, ES6 [3]—that permits JavaScript programs to create large arrays of scalar types with very little overhead. For example, users could create an array of `uint8` or `uint16` values and be assured that the memory usage per element is only 1 or 2 bytes respectively. (We cover the existing typed arrays API in more detail in Section 2.)

Although the typed arrays API has seen widespread usage, it also suffers from some important shortcomings. For example, it is not possible to store references to JavaScript objects in these arrays, nor is it possible to construct higher-level abstractions beyond scalar types (e.g., an array of structures). Finally, some of the core decisions in the API design can hinder advanced optimizations by JIT compilers.

This paper presents the *typed objects* API. Typed objects is a generalization of the typed arrays API that lifts these limitations. In the typed objects API, users can define and employ their own structure and array types (§3). These structure and array types are integrated with JavaScript's prototype system, making it possible for users to attach methods to them (§4). Moreover, these types are not limited to scalar data, but can also be used to store references to JavaScript objects and strings (§5). Finally, we have made a number of small changes throughout the design that improve the ability of JavaScript engines to optimize code that uses the typed objects API (§6).

The typed objects API is currently slated for inclusion in the ES7 standard, which is expected to be released in 2015. The API should be available in browsers much earlier than that, however—and in fact Nightly builds of Firefox already contain an prototype implementation. Although this implementation is incomplete, we briefly describe how it integrates with the JIT compilation framework, and give some preliminary measurements of its performance (§7). Finally, we compare the typed objects API to existing projects (§8).

2. Typed arrays today

Most implementations of JavaScript today support an API called *typed arrays*. Typed arrays provide the means to efficiently manage large amounts of binary data. They also support some unique features such as *array buffer transfer*, which permits data to be moved from thread to thread.

The typed array standard defines a number of array types, each corresponding to some scalar type. For example, the types `Uint32Array` and `Int32Array` correspond to arrays of unsigned and signed 32-bit integers, respectively. There are also two floating point array types `Float32Array` and `Float64Array`.

Instantiating an array type creates a new array of a specified length, initially filled with zeroes:

```
var uints = new Uint32Array(100);
uints[0] += 1; // uints[0] was 0, now 1
```

Each array type also defines a `buffer` property. This gives access to the array's backing buffer, called an *array buffer*. The array buffer itself is simply a raw array of bytes with no interpretation. By accessing the array buffer, users can create multiple arrays which are all views onto the same data:

```
var uint8s = new Uint8Array(100);
uint8s[0] = 1;
```

```
var uint32s = new Uint32Array(uint8s.buffer);
print(uint32s[0]); // prints 1, if little endian
```

It is also possible to instantiate an array buffer directly using `new ArrayBuffer(size)`, where `size` is the number of bytes.

The final type in the typed array specification is called `DataView`; it permits "random access" into an array buffer, offering methods to read a value of a given type at a given offset with a given endianness. Data view is primarily used for working with array buffers containing serialized data, which may have very particular requirements. We do not discuss it further in this paper.

2.0.1 Using array buffer views to mix data types

The typed array specification itself does not provide any higher means of abstraction beyonds arrays of scalars. Therefore, if users wish to store mixed data types in the same array buffer, they must employ multiple views onto the same buffer and read using the appropriate array depending on the type of value they wish to access.

For example, imagine a C struct which contains both a `uint8` field and a `uint32` field:

```
struct Example {
    uint8_t f1;
    uint32_t f2;
};
```

To model an array containing `N` instances of this struct requires creating a backing buffer and two views, one for accessing the `uint8` fields, and one for accessing the `uint32` fields. The indices used with the various views must be adjusted appropriately to account for padding and data of other types:

```
var buffer = new ArrayBuffer(8 * N);
var uint8s = new Uint8Array(buffer);
var uint32s = new Uint32Array(buffer);
uint8s[0] = 1;  // data[0].f1
uint32s[1] = 2; // data[0].f2
uint8s[8] = 1;  // data[1].f1
uint32s[3] = 2; // data[1].f2
```

This snippet begins by creating a buffer containing `N` instances of the struct, allotting 8 bytes per instance (1 byte for the `uint8` field, 3 bytes of padding, and then 4 bytes for the `uint32` field). Next, two views (`uint8s` and `uint32s`) are created onto this buffer. Accessing the `uint8` field `f1` at index `i` can then be done via the expression `uints8[i*8]`, and accessing the `uint32` field `f2` can be done via the expression `uint32s[i*2+1]`. (Note that the array index is implicitly multiplied by the size of the base type.)

One place where multiple views are effectively employed is by compilers that translate C into JavaScript, such as emscripten [9] and mandreel [7]. Such compilers employ a single array buffer representing "the heap", along with one view per data type. Pointers are representing as indices into the appropriate view; hence a pointer of type `uint32*` would be an index into the `uint32` array.

2.0.2 Array buffer transfer

JavaScript is a single-threaded language with no support for shared memory. Most JavaScript engines, however, support the *web workers API*, which permits users to launch distinct workers that run in parallel. These workers do not share memory with the original and instead communicate solely via messaging.

Generally, messages between workers are serialized and recreated in the destination. However, because array buffers contain only raw, scalar data, it is also possible to *transfer* an array buffer to another worker without doing any copies.

Transfering an array buffer does not copy the data. Instead, the data is moved to the destination, and the sender loses all access. Any existing aliases of that buffer or views onto that buffer are *neutered*, which means that their connection to the transferred buffer is severed. Any access to a neutered buffer or view is treated as if it were out of bounds.

Array buffer transfer is an extremely useful capability. It permits workers to offload large amounts of data for processing in a parallel thread without incurring the costs of copying.

2.0.3 Limitations

The typed arrays have proven to be very useful and are now a crucial part of the web as we know it. Unfortunately, they have a number of shortcomings as well. Alleviating these shortcomings is the major goal of the typed objects work we describe in this paper.

The single biggest problem is that typed arrays do not offer any means of abstraction. While we showed that it is possible to store mixed data types within a single array buffer using multiple views, this style of coding is inconvenient and error-prone. It works well for automated compilers like emscripten but is difficult to use when writing code by hand.

Another limitation is that typed arrays can only be used to store scalar data (like integers and floating point values) and not references to objects or strings. This limitation is fundamental to the design of the API, which always permits the raw bytes of a typed array to be exposed via the array buffer. Even with scalar data, exposting the raw bytes creates a small portability hazard with respect to endianness; if however the arrays were to contain references to heap-allocated data, such as objects or strings, it would also present a massive security threat.

The technique of aliasing multiple views onto the same buffer also creates an optimization hazard. JavaScript engines must be very careful about reordering accesses to typed array views, because any two views may in fact reference the same underlying buffer. In practice, most JITs simply forego reordering, since they do not have the time to conduct the required alias analysis to show that it is safe.

3. The Typed Objects API in a nutshell

The Typed Objects API is a generalization of the Typed Arrays API that supports the definition of custom data types. Along the way, we also tweak the API to better support optimization and encapsulation.

3.1 Defining types

The typed objects API is based on the notion of *type objects*. A type object is a JavaScript object representing a type. Type objects define the layout and size of a continuous region of memory. There are three basic categories of type objects: primitive type objects, struct type objects, and array type objects.

Primitive type objects. Primitive type objects are type objects without any internal structure. All primitive type objects are predefined in the system. There are 11 of them in all:

```
any         uint8       int8        float32
object      uint16      int16       float64
string      uint32      int32
```

The majority of the primitive types are simple scalar types, but they also include three reference types (`any`, `object`, and `string`). The reference types are considered *opaque*, which means that users cannot gain access to a raw array buffer containing instances of these types. The details of opacity are discussed in Section 5.

Struct type objects. Type objects can be composed into structures using the `StructType` constructor:

```
var Point = new StructType({x:int8, y:int8});
```

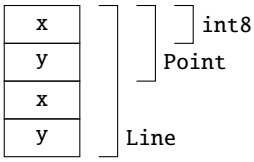

Figure 1: Layout of the `Line` type defined in Section 3.1.

This example constructs a new type object called `Point`. This type is a structure with two 8-bit integer fields, `x` and `y`. The size of each `Point` will therefore be 2 bytes in total.

In general, the `StructType` constructor takes a single object as argument. For each property `f` in this object, there will be a corresponding field `f` in the resulting struct type. The type of this corresponding field is taken from the value of the property `f`, which must be a type object.

Structures can also embed other structures:

```
var Line = new StructType({from:Point, to:Point};
```

Here the new type `Line` will consist of two points. The layout of `Line` is depicted graphically in Figure 1. It is important to emphasize that the two points are laid out continuously in memory and are not pointers. Therefore, the `Line` struct has a total size of 4 bytes.

Array type objects. Array type objects are constructed by invoking the `arrayType()` method on the type object representing the array elements:

```
var Points = Point.arrayType(2);
var Line2 = new StructType({points:Points});
```

In this example, the type `Points` is defined as a 2-element array of `Point` structures. Array types are themselves normal type objects, and hence they can be embedded in structures. In the example, the array type `Points` is then used to create the struct type `Line2`. `Line2` is equivalent in layout to the `Line` type we saw before but it is defined using a two-element array instead of two distinct fields.

The `arrayType()` constructor can be invoked multiple times to create multidimensional arrays, as in this example which creates an `Image` type consisting of a 1024x768 matrix of pixels:

```
var Pixel = new StructType({r:uint8, g:uint8,
                            b:uint8, a:uint8});
var Image = Pixel.arrayType(768).arrayType(1024);
```

3.2 Instantiating types

Once a type object `T` has been created, new instances of `T` can be created by calling `T(init)`, where `init` is an optional *initializer*. The initial data for this instance will be taken from the initializer, if provided, and otherwise default values will be supplied.

If the type object `T` is a primitive type object, such as `uint8` or `string`, then the instances of that type are simply normal JavaScript values. Applying the primitive type operators simply acts as a kind of cast. Hence `uint8(22)` returns `22` but `uint8(257)` returns `1`.

Instances of struct and array types, in contrast, are called *typed objects*. A typed object is the generalized equivalent of a typed array; it is a special kind of JavaScript object whose data is backed by an array buffer. The properties of a typed object are defined by its type: so an instance of a struct has a field for each field in the type, and an array has indexed elements.

As an example, consider this code, which defines and instantiates a `Point` type:

```
var Point = new StructType({x:int8, y:int8});
...
var point = Point(); // x, y initially 0
point.x = 22;
point.y = 257; // wraps to 1
```

Since `Point()` is invoked with no arguments, all the fields are initialized to their default values (in this case, `0`). Assigning to the fields causes the value assigned to be coerced to the field's type and then modifies the backing buffer. In this example, the field `y` is assigned `257`; because `y` has type `int8`, `257` is wrapped to `1`.

Struct types can also be created using any object as the initializer. The initial value of each field will be based on the value of corresponding field within the initializer object. This scheme permits standard JavaScript objects to be used as the initializer, as shown here:

```
var Point = new StructType({x:int8, y:int8});
...
var point = Point({x: 22, y: 257});
// point.x == 22, point.y == 1, as before
```

The value of the field is recursively coerced using the same rules, which means that if you have a struct type that embeds other struct types, it can be initialized using standard JavaScript objects that embed other objects:

```
var Point = new StructType({x:int8, y:int8});
var Line = new StructType({from:Point, to:Point});
var line = new Line({from:{x:22, y:256},
                     to:{x:44, y:66}});
// line.from.x == 22, line.from.y == 1
// line.to.x == 44, line.to.y == 66
```

Assignments to properties in general follow the same rules as coercing an initializer. This means that one can assign any object to a field of struct type and that object will be adapted as needed. The following snippet, for example, assigns to the field `line.from`, which is of `Point` type:

```
var Point = new StructType({x:int8, y:int8});
var Line = new StructType({from:Point, to:Point});
...
var line = Line();
line.from = {x:22, y:257};
// line.from.x == 22, line.from.y == 1
```

As before, `line.from.x` and `line.from.y` are updated based on the `x` and `y` properties of the object.

Creating an array works in a similar fashion. The following snippet, for example, creates an array of three points, initialized with values taken from a standard JavaScript array:

```
var Point = new StructType({x:int8, y:int8});
var PointVec = Point.arrayType(3);
...
var points = PointVec([{x: 1, y: 2},
                       {x: 3, y: 4},
                       {x: 5, y: 6}]);
```

Note that here the elements are `Point` instances, and hence can be initialized with any object containing `x` and `y` properties.

For convenience and efficiency, every type object `T` offers a method `array()` which will create an array of `T` elements without requiring an intermediate type object. `array()` can either be supplied the length or an example array from which the length is derived. The previous example, which created an array of three points based on the intermediate type object `PointVec`, could therefore be rewritten as follows:

```
var Point = new StructType({x:int8, y:int8});
...
```

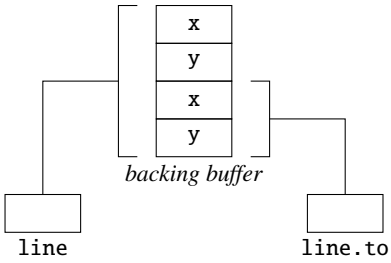

Figure 2: Accessing a property of aggregate type returns a new typed object that aliases a portion of the original buffer.

```
var points = Point.array([{x: 1, y: 2},
                          {x: 3, y: 4},
                          {x: 5, y: 6}]);
```

In this version, there is no need to define the `PointVec` type at all.

3.3 Accessing properties and aliasing

Accessing a struct field or array element of primitive type returns the value of that field directly. For example, accessing the fields of a `Point`, which have `int8` type, simply yields JavaScript numbers:

```
var Point = new StructType({x:int8, y:int8});
...
var point = Point({x: 22, y: 44});
var x = point.x; // yields 22
```

Accessing a field or element of aggregate type returns a new typed object which points into the same buffer as the original object. Consider the following example:

```
var Point = new StructType({x:int8, y:int8});
var Line = new StructType({from:Point, to:Point});
...
var line = Line({from:{x:0, y:1},
                 to:{x:2, y:3}});
var point = line.to;
point.x = 4; // now line.to.x == 4 as well
```

Here, the variable `line` is a struct containing two `Point` embedded within. The expression `line.to` yields a new typed object `point` that aliases `line`, such that modifying `point` also modifies `line.to`. That is, both objects are views onto the same buffer (albeit at different offsets). The aliasing relationships are depicted graphically in Figure 2: the same backing buffer is referenced by `line` and the result of `line.to`.

3.3.1 Equality of typed objects

The fact that a property access like `line.to` yields a new typed object raises some interesting questions. For one thing, it is generally true in JavaScript that `a.b === a.b`, unless `b` is a getter. But because the `===` operator, when applied to objects, generally tests *pointer equality* for objects, `line.to === line.to` would not hold, since each evaluation of `line.to` would yield a distinct object.

We chose to resolve this problem by having typed objects use a *structural definition of equality*, rather than testing for pointer equality. In effect, in our system, a typed object can be considered a four tuple:

1. Backing buffer;
2. Offset into the backing buffer;
3. Type and (if an array type) precise dimensions;
4. Opacity (see Section 5).

```
1  function Cartesian(x, y) {
2    this.x = x;
3    this.y = y;
4  }
5  Cartesian.prototype.toPolar = function() {
6    var r = Math.sqrt(x*x + y*y);
7    var c = Math.atan(y / x);
8    return new Polar(r, c);
9  };
10 function Polar(r, c) {
11   this.r = r;
12   this.c = c;
13 }
14 var cp = new Cartesian(22, 44);
15 var pp = cp.toPolar();
```

Figure 3: Defining classes for cartesian and polar points in standard JavaScript.

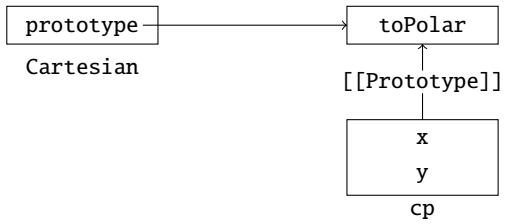

Figure 4: Prototype relationshipships for the `Cartesian` function and one of its instances, `cp`. The label `[[Prototype]]` indicates the prototype of an object.

```
1  var Cartesian = new StructType({x:float32,
2                                  y:float32});
3  Cartesian.prototype.toPolar = function() {
4    var r = Math.sqrt(x*x + y*y);
5    var c = Math.atan(y / x);
6    return Polar({r:r, c:c});
7  };
8  var Polar = new StructType({r:float32,
9                              c:float32});
10 var cp = Cartesian({x:22, y:44});
11 var pp = cp.toPolar();
```

Figure 5: Attaching methods to type objects works just like attaching methods to regular JavaScript functions.

Two typed objects are considered equal if all of those tuple elements are equal. In other words, even if `line.to` allocates a fresh object each time it is executed, those objects would point at the same buffer, with the same offset and type, and the same opacity, and hence they would be considered equal.

In effect, the choice to use structural equality makes it *invisible* to end-users whether `line.to` allocates a new typed object or simply uses a cached result. This is not only more user friendly (since `line.to === line.to` holds) but can also be important for optimization, as discussed in Section 6.

4. Integrating with JavaScript prototypes

Typed objects are designed to integrate well with JavaScript's prototype-based object system. This means that it is possible to define methods for instances of struct and array types.

The ability to define methods makes it possible to migrate from normal JavaScript "classes"[1] to types based around typed objects. This is particularly useful for common types, as the representation of typed objects can be heavily optimized.

In this section, we describe how typed objects are integrated with JavaScript's prototype system. Before doing so, however, we briefly cover how ordinary prototypes in JavaScript work, since the system is somewhat unusual.

4.1 Standard JavaScript prototypes

In prototype-based object systems, each object `O` may have an associated prototype, which is another object. To lookup a property `P` on `O`, the engine first searches the properties defined on `O` itself. If no property named `P` is found on `O`, then the search continues with `O`'s prototype (and then the prototype's prototype, and so on).

One very common pattern with prototypes is to emulate classes by having a designated object `P` that represents the class. This object contains properties for the class methods and so forth. Each instance of the class then uses that object `P` for its prototype. Thus looking up a property on an instance will fallback to the class.

JavaScript directly supports this class-emulation pattern via its **`new`** keyword. Figure 3 demonstrates how it works. A "class" is defined by creating a function that is intended for use as a constructor. In Figure 3, there are two such functions: `Cartesian`, for cartesian points, and `Polar`, for polar points. Each function has an associated `prototype` field which points to the object that will be used as the prototype for instances of that function (called `P` in the previous paragraph). For a given constructor function `C`, therefore, one can add methods to the class `C` by assigning them into `C.prototype`, as seen on line 5 of Figure 3.

New objects are created by writing a new expression, such as the expression **`new`** `Cartesian(...)` that appears on line 14. The effect of this is to create a new object whose prototype is `Cartesian.prototype`, and then invoke `Cartesian` with `this` bound to the new object. The function `Cartesian` can then initialize properties on `this` as shown. Note that the property `prototype` on the function `Cartesian` is not the prototype of the function, but rather the prototype that will be used for its instances.

The prototype relationship for the function `Cartesian` and the instance `cp` is depicted graphically in Figure 4. The diagram shows the function `Cartesian` and its instance `cp`. The function `Cartesian` has a single property, `prototype`, which points at an (unlabeled) object `O`. `O` has a single property, which is the method `toPolar` that is installed in the code on line 5. `O` serves as the prototype for the instance `cp`. In addition, the instance `cp` has two properties itself, `x` and `y`. Therefore, an access like `cp.x` will stop immediately, but a reference to `cp.toPolar` will search the prototype `O` before being resolved.

4.2 Prototypes and typed objects

Typed objects make use of prototypes in the same way. Struct and array type objects define a `prototype` field, just like the regular JavaScript functions. When a struct or array `T` is instantiated, the prototype of the resulting typed object is `T.prototype`. Installing methods on `T.prototype` therefore adds those methods to all instances of `T`.

Figure 5 translates the example from Figure 3 to use typed objects. It works in a very analagous fashion. Two type objects, `Cartesian` and `Polar`, are defined to represent coordinates. A `toPolar` method is installed onto `Cartesian.prototype` just as before (line 3). As a result, instances of `Cartesian` (such as `cp`) have the method `toPolar`, as demonstrated on line 11. In fact, because both normal JavaScript functions and typed objects handle

[1] As JavaScript is prototype-based, a class is really more of a convention.

```
1  var Color = new StructType({r:uint8, g:uint8,
2                              b:uint8, a:uint8});
3  var Row384 = Color.arrayType(384);
4  var Row768 = Color.arrayType(768);
5
6  Color.arrayType.prototype.average = function() {
7    var r = 0, g = 0, b = 0, a = 0;
8    for (var i = 0; i < this.length; i++) {
9      r += this[i].r; g += this[i].g;
10     b += this[i].b; a += this[i].a;
11   }
12   return Color({r:r, g:g, b:b, a:a});
13 };
14
15 var row384 = Row384();
16 var avg1 = row384.average();
17
18 var row768 = Row768();
19 var avg2 = row768.average();
```

Figure 6: Defining methods on an array type. Methods installed on `Color.arrayType.prototype` are available to all arrays of colors, regardless of their length.

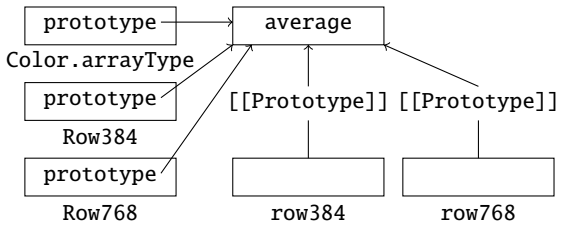

Figure 7: The prototype relationships for the types and instances from Figure 6.

prototypes in such a similar fashion, we can use the same diagram (Figure 4) to depict both of them.

In the example as written, the typed objects code is not a drop-in replacement for the standard JavaScript version, due to differences in how instances of `Cartesian` are created. For example, creating a coordinate with typed objects is written:

```
Cartesian({x:22, y:44})
```

but in the normal JavaScript version it was written:

```
new Cartesian(22, 44)}
```

This difference is rather superficial and easily bridged by creating a constructor function that returns an instance of the struct type:

```
var CartesianType = new StructType({...});
CartesianType.prototype.toPolar = ...;
function Cartesian(x, y) {
  return CartesianType({x:x, y:y});
}
```

Due to the specifics of how JavaScript **`new`** expressions work, existing code like **`new`** `Cartesian(22, 44)` will now yield a `CartesianType` object.

4.3 Prototypes and arrays

One important aspect of the design is that all array type objects with the same element type share a prototype, even if their lengths differ. The utility of this design is demonstrated in Figure 6. This code creates a `Color` type and then two different types for arrays of

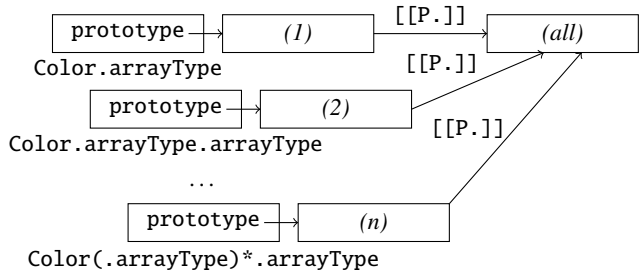

Figure 8: Illustrates the prototypes for multidimensional arrays. There is a distinct prototype for each number of dimensions. Each of those prototypes in turn inherits from a common prototype representing an array of any number of dimensions. The label `[[P.]]` here indicates the prototype of an object.

color, `Row384` and `Row768`. Even though these types have different lengths, the values of their `prototype` fields are the same, and hence `Row384` and `Row768` will have the same prototype. The prototypes at play are illustrated graphically in Figure 7.

The prototype for array types with a given element can be accessed without actually instantiating an array type object. On line 6 of Figure 6, the method `average()` is defined and attached to `Color.arrayType.prototype`. As shown in Figure 7, this is the same object which will later be used as the `prototype` for `Row384` and `Row768`.

The reason that all arrays share the same prototypes regardless of length is that many programs never instantiate explicit array type objects at all, since the lengths of arrays often vary with the input. For example, rather than defining explicit `Row384` and `Row768` types, it is more common to create arrays of colors ad-hoc:

```
var row384 = Color.array(384);
var avg1 = row384.average();
var row768 = Color.array(768);
var avg2 = row768.average();
```

Our design accommodates this pattern without difficulty as creating methods for array instances does not require creating actual array type objects.

The decision to have all array type objects share a prototype came late in the design cycle. In the initial design, all type objects – whether array or not – had their own `prototype` field.[2] This led to numerous problems. One problem was that because the lengths of most arrays are not uniform, users frequently instantiated "throw-away" array type objects that were used for exactly one array, which is both annoying and inefficient. Another problem was that many methods return arrays whose lengths are not known in advance; `filter()` is an example. If the type of an array includes its length, then clearly the type of array returned by `filter()` cannot be known in advance, and thus users cannot identify which prototype it will have. The current design avoids these obstacles.

4.3.1 Multidimensional arrays

One of the main uses cases for typed objects is to support numerical applications, which often work with matrices and other multi-dimensional structures. The prototypes for multi-dimensional arrays follow the same principles as for single-dimensional arrays: all arrays with the same rank – that is, the same number of dimensions – share a prototype, regardless of the precise lengths of each dimension. For example, all two-dimensional arrays of `Color` will have the same prototype, but a three-dimensioanl array of `Color` would have a distinct prototype. These prototypes can be accessed by stringing together references to `arrayType`. The prototype that will be used for three-dimensional arrays of `Color` elements, for example, is available using the path:

```
Color.arrayType.arrayType.arrayType.prototype
```

Both for efficiency and because there can be an arbitrary number of dimensions, the prototypes for arrays are created lazilly upon request.

Finally, to permit the creation of methods that apply to any multidimensional array, regardless of the number of dimensions, all prototypes for arrays inherit from a common ancestor prototype, as shown in Figure 8. The diagram depicts how there are separate prototypes for arrays of each possible dimension (labeled *(1)*, *(2)*, and *(n)*). Each of those prototypes then inherits from a common prototype (labeled *(all)*) that applies to all arrays of `Color`, no matter how many dimensions they have.

5. Array buffers and opacity

The original typed arrays API does not include any means of protecting or encapsulating an array buffer. Given a typed array view, it is always possible to recover the original array buffer and create a new view on top of it. This unrestrictied aliasing has many important ramifications:

1. Objects and other reference types (like strings) cannot be allowed into an array buffer, because that array buffer could always be reinterpreted casting into an array of bytes. This would permit users to observe the raw bytes of a pointer value. This is a security nightmare.
2. Passing a view on a portion of a buffer into a function actually gives that function access to the entire buffer. There is no means to securely give away access only to a portion of the buffer. This limits the sorts of APIs we can support.
3. In the absence of alias analysis, the JIT must assume that a write to any typed array potentially affects every other typed array. This inhibits optimization.

Like typed arrays, the typed objects API includes two standalone functions, `buffer()` and `offset()`, that can be used to obtain the backing buffer in which a typed object resides as well as the offset of its data. Unlike typed arrays, however, we also include the means to deny this access.

Any individual typed object can be made *opaque*, in which case the `buffer()` function returns `null` (and `offset()` returns `undefined`). This is done by invoking the module-level function `opaque(obj)`, which returns an opaque copy of the typed object `obj`. The new typed object is otherwise identical to the original: it points into the same array buffer, with the same offset, and the same type.

Opaque typed objects are useful because they serve as a capability that only exposes a limited portion of the backing buffer; the remainder of the buffer that is outside the scope of that typed object's type remains inaccessible. For example, given an array of structs `points`, the expression `opaque(points[3])` yields a typed object that can only be used to access element 3 of the array. If the typed object were not opaque, someone who possesses `points[3]` could simply obtain access to the entire buffer and thus access other elements of `points`.[3]

[2] In type theory terms, one might say that in our initial design, all types were nominal, whereas in the current design, struct types are nominal but array types are structural. It is encouraging to note that many statically typed languages take a similar approach.

[3] The need for encapsulation is immediate and real: for example, we are building parallel APIs that employ opaque pointers to guarantee data-race freedom.

```
1  var Color = new StructType({r: uint8, g: uint8,
2                              b: uint8, a: uint8});
3  var Palette = Color.arrayType(256);
4  var Header = new StructType({palette: Palette,
5                               height: uint32,
6                               width: uint32});
7
8  function process(header, data) {
9    for (var x = 0; x < header.width; x++)
10     for (var y = 0; y < header.height; y++)
11       processDataPoint(data[x][y], header.palette);
12 }
```

Figure 9: Example program that is easier to optimize due to the way equality is defined for typed objects. In particular, the access to `header.palette` on line 11 can easily be identified as loop invariant.

It is also possible for a type object to be opaque. When a type object is opaque, then all of it instances are automatically opaque. The reference types `string`, `any`, and `object` are always opaque, as are any struct or array types that transitively contain reference types. This avoids the security hazard of reinterpret casting a pointer into its raw bytes, and also avoids exposing the precise way that each engine represents pointers.

In addition, an opaque type can be explicitly derived from any other type using the `opaqueType()` method. `opaqueType()` returns a new type object with the same structural definition as the receiver, but which is opaque. For example, the following code snippet creates an opaque version of the `Color` type, so that instances of `Color` never expose their backing buffer:

```
var Color = new StructType({r: uint8, g: uint8,
                            b: uint8, a: uint8})
            .opaqueType();
```

Using opaque types has some subtle advantages for optimization, as is discussed in the next section.

6. Enabling optimization

The typed objects API has been designed with an eye towards enabling aggressive optimization in JavaScript JITs. In this section, we describe some of the finer points of the design and why they are important.

6.1 Defining equality for typed objects

As discussed in Section 3.3.1, equality between typed objects is defined structurally, unlike other objects which use pointer equality. Section 3.3.1 showed that this definition better meets user expectations but another important motivation for this change is performance. A pointer-based notion of equality obligates the engine to allocate a fresh typed object unless it can prove that this object is never tested for equality with any other object. This can lead to non-obvious performance pitfalls.

Consider the code in Figure 9, specifically line 11. This line invokes a function and passes `header.palette` as the second argument. Because the type `Palette` is an aggregate type, the expression `header.palette` (potentially) allocates a new typed object on every iteration. This in turn stresses the garbage collector and leads to more frequent GCs. This situation is particularly unfortunate because the expression `header.palette` *looks* innocuous.

In an ideal world, the JavaScript engine would extract the expression `header.palette` as loop invariant. After all, the only effect of the expression is to create a new typed object that aliases `header`.[4] Using a pointer-based notion of equality, though, this optimization would generally only be possible if `processDataPoint()` were inlined and the engine could be sure that the result of `header.palette` never escapes. Using a structural definition of equality frees the engine from this restriction, as the user cannot observe whether `header.palette` is evaluated once or many times.

Benchmark	1. Standard	2. Typed Objects	Ratio 1:2
Array of scalars	1040ms	837ms	1.243
Struct fields	936ms	1227ms	0.763
Array of structs	1970ms	1064ms	1.852

Figure 10: Performance of various microbenchmarks on the prototype Firefox implementation. We wrote equivalent versions of each microbenchmark using standard JavaScript objects and typed objects, and measured the execution time.

6.2 Memory usage and opacity

Using the typed objects API rather than standard JavaScript objects naturally gives the JavaScript engine a lot of room to optimize memory usage. The struct definition declares all possible fields and their types, so in principle engines can allocate precisely the correct amount of memory required (along with some headers, naturally). Opaque types also imply that the array buffer for instances of that type will never be accessed externally, which may help the engine to optimize the representation further.

6.3 Opacity and aliasing

Opaque types also give strong aliasing guarantees than transparent types, which can be useful for optimization. Specifically, If the engine sees accesses to two distinct fields that are both defined in opaque types, it can be guaranteed that those two fields do not alias. This can permit much more aggressive reordering of code. With transparent types, engines must always consider the possibility that the same buffer has been reinterpreted as two different types.

7. Implementation Status

The typed objects API described in this paper is being standardized as part of the ES7 specification. The major design work is completed and we are in the process of writing the formal text.

7.1 Firefox Implementation

The API is being actively implemented into Firefox's SpiderMonkey JavaScript engine and is available in Nightly builds. It will become more broadly available as standardization proceeds. As of the time of this writing, the implementation targets an older version of the API than what is described in this paper. We expect it to be updated to match the API described here within the coming months.

Preliminary integration with the JIT has also been done and initial performance results are encouraging, though much work remains. The API fits particularly well with SpiderMonkey's existing type inference infrastructure [11]. Like all optimizing JavaScript engines (and indeed all optimizing JITs), SpiderMonkey employs a two-phase system. When code is first executed, the engine monitors the types of all objects it encounters. Once a function has been executed many times, it is identified as a hot method, and the engine recompiles it. At this point, the engine has gathered a reasonably complete profile of what types of objects the function

[4] Note that it is not important whether `header` itself is aliased or not, as those aliases cannot change the buffer with which `header` is associated, nor the type of `header`.

commonly operates on, and thus it can specialize for those types. If new types of objects are encountered, the optimized code will have to be thrown out and recompiled with weaker assumptions.

In SpiderMonkey at least, optimizing typed objects does not require any modification to the existing type monitoring infrastruture. The reason for this is that the type observation already gathers information about the prototype of each object it observes. Since each type object is connected to a distinct prototype (§4), the optimizing compiler can determine everything it needs about the layout of an object by looking at its prototype.

7.1.1 Performance measurements

Due to the immaturity and incomplete nature of the current implementation, we have not performed extensive benchmarking (we are focusing our efforts on bringing the API up to date first). However, we have created some simple micro-benchmarks to evaluate the effectiveness of the type monitoring infrastructure. Results can be found in Figure 10. The sources to the benchmarks are included in the appendix, as they provide interesting examples of how code that uses normal JavaScript objects can be converted to use typed objects.

All of these benchmarks are small microbenchmarks with a central loop that reads and writes from some particular kind of data structure. We tuned the number of iterations so that the typed object variation would complete in approximately one second. This was compared against an equivalent benchmark that translated the typed object data structure into naive JavaScript equivalents (for example, an array becomes a JavaScript array, a struct becomes a JavaScript object, and scalar values become JavaScript numbers). Since we were controlling the number of iterations, the precise timings are not significant, but the ratio of the two values gives an idea of the performance of typed objects relative to normal JavaScript code.

These measurements were taken on a Intel i7-3520M CPU running at 2.90 GHz using an optimized build of the SpiderMonkey trunk as of the time of this writing. Up to the minute measurements for these benchmarks are also available on the SpiderMonkey performance monitoring web site *Are we fast yet?* [1], under the "assorted" section.

The first microbenchmark, "array of scalars", consists of a loop reading and writing bytes from and to an array. In this case, the typed objects code outperforms the standard JavaScript arrays, likely because arrays occupy less memory (1 byte per element vs 8 bytes for a standard JS array) and hence there are fewer cache misses.

The second microbenchmark, "struct fields", creates a struct with two fixed-length arrays as fields and repeatedly reads and writes those fields. In this case, the amount of data is fixed and hence cache effects are not an issue. The standard JavaScript code paths outperform typed objects, indicating an area where more work is needed in our implementation.

The final microbenchmark, "array of structs" creates a 1024x768 image of `Color` structs. In this version, the typed objects outperforms the standard JavaScript one. We believe this is for several reasons: the color fields are smaller and laid out inline in the typed objects version, which is more cache friendly; furthermore, reading and writing those fields doesn't require any pointer chasing, unlike the standard JS version.

7.2 Pure JavaScript implementation

The API has also been implemented in pure JavaScript using typed arrays. The source code for this implementation can be downloaded from GitHub at the following URL:

`https://github.com/dslomov-chromium/structs.js`

Naturally this implementation does not provide good performance, but it is useful to get a feeling for how the API works.

7.3 Experience using the API

Helping to validate the design, the Firefox implementation has been used to develop algorithms for image manipulation and other computationally intensive tasks. These algorithms also make use of further APIs for parallel computation which build on typed objects and which are outside the focus of this paper. Some examples can be found at the following URL:

`http://intelglobal13.github.io/ParallelJS-CV/`

8. Related work

The most closely related work in JavaScript is the existing typed array APIs, which are described in detail in Section 2.

There are a large number of foreign function interface APIs [2, 4–6] that work in a similar way to the typed objects API. Python's ctypes API [2] is a representative example (and one which was particularly influential on the early design of typed objects). In the ctypes API, as with typed objects, users create objects that describe types in the C language. These structures can be instantiated and passed to C functions. The focus on FFI integration however leads to a rather different end product from our own work. For example, ctypes is limited to scalar data, and its type system includes notions (like union) that are rather specific to C. It has also not been designed with high performance or JIT integration in mind. (Mozilla in fact has a port of Python's ctypes API [5] used for internal JavaScript, which will likely be reimplementing using typed objects in the future.)

The Python data model includes a notion of *slots* [8] that can be used to control memory usage: when a class is defined, the user may specify a fixed set of slots. Instances of that class are then pre-instantiated with precisely that set of fields, and they cannot be extended with additional fields later. This is useful when there will be a large number of instances of the class. In comparison to the typed objects API, the slots mechanism is more limited, because it does not permit the types of each slot to be declared. Types can allow for significantly higher memory savings in some cases. The slot model also does not allow for "inline" allocation of structs and objects, which means that it would not help in situations like the "array of structs" benchmark from Figure 7.

Advanced VMs can automatically perform *object inlining* [10, 15] or *object combining* [14], in which object hierarchies are collapsed in order to avoid extra allocations and pointer indirections. Doing this sort of optimization automatically is convenient in some cases, but also less reliable than (and significantly more complicated for the implementor than) having user-supplied type annotations.

The structural notion of equality that we use for typed objects makes them a kind of *fat pointer*, as seen in the language Cyclone [12]. Fat pointers in Cyclone are a safe kind of pointer which carries the bounds of the array it points into along with it, so that dereferences can be bounds checked. In our case, the fat pointers carry not only the bounds of the memory it points at but the type as well, which is appropriate for a dynamically typed language like JavaScript.

9. Conclusion

We have presented a JavaScript API for defining typed objects. Typed objects support precise type layouts similar to those found in C or other statically typed languages. Typed objects have been designed for performance, with a focus on enabling strong alias analysis and other optimization properties.

A. Benchmark Sources

This appendix contains the sources to the three microbenchmarks. In each case, there are two distinct "preambles" that initialize either a typed object data structure or else an equivalent standard JavaScript data structure. The inner loop then follows and is the same for both versions. Timings include both the initialization and the inner loop. We tuned the `NUM_ITERS` variable manually to achieve running times of approximately one second for the typed objects version.

A.1 Array of scalars

```
// In the typed objects version:
var array_in = uint8.array(NUM_ITERS);
var array_out = uint8.array(NUM_ITERS);

// In the standard version:
var array_in = [];
var array_out = [];

// Common to both versions:
for(var i = 0; i < NUM_ITERS; i++)
  array_in[i] = 1;
var sum = 0;
for(var k = 0; k < 15; k++)
  for(i = 0; i < NUM_ITERS; i++)
    array_out[i] = bdArray_in[i];
```

A.2 Struct fields

```
// In the typed objects version:
var ThreeVector = float64.arrayType(3);
var Result = new StructType({pos: ThreeVector,
                             nor: ThreeVector});
var p = new Result();

// In the standard version:
var p = { pos: [0, 0, 0], nor: [0, 0, 0] };

// Common to both versions:
var v = [1, 2, 3];
for (var i = 0; i < NUM_ITERS; i++) {
  v[0] = i+0.5;
  v[1] = i+1.5;
  v[2] = i+2.5;
  out.pos[0] = v0[0];
  out.pos[1] = v0[1];
  out.pos[2] = v0[2];
  out.nor[0] += v0[0];
  out.nor[1] += v0[1];
  out.nor[2] += v0[2];
}
```

A.3 Array of structs

```
// In the typed objects version:
var {StructType,uint8} = TypedObject;
var Color = new StructType({r:uint8, g:uint8,
                            b:uint8, a:uint8});
var Image = Color.array(1024, 768);
var image = new Image();

// In the standard version:
var image = [];
for (var x = 0; x < WIDTH; x++) {
  image[x] = [];
  for (var y = 0; y < HEIGHT; y++) {
    image[x][y] = {r:0, g:0, b:0, a:0};
  }
}

// Common to both versions:
for (var i = 0; i < NUM_ITERS; i++) {
  for (var b = 0; b < 256; b += 2) {
    for (var x = 0; x < width; x++) {
      for (var y = 0; y < height; y++) {
        image[x][y].r = b;
        image[x][y].g = b;
        image[x][y].a = b;
      }
    }
  }
}
```

References

[1] Are we fast yet? `http://arewefastyet.com`.

[2] ctypes—A foreign function interface for Python. `http://docs.python.org/2/library/ctypes.html`.

[3] Draft Specification for ES.next (Ecma-262 Edition 6). `http://wiki.ecmascript.org/`.

[4] Java Native Access. `https://github.com/twall/jna`.

[5] JSctypes. `https://wiki.mozilla.org/JSctypes`.

[6] LibFFI. `https://sourceware.org/libffi/`.

[7] Mandreel. `http://www.mandreel.org`.

[8] The Python Data Model, Section 3.4.2.4. `http://docs.python.org/2/reference/datamodel.html`.

[9] Alon Zakai et al. Emscripten. `http://www.emscripten.org`.

[10] J. Dolby. Automatic Inline Allocation of Objects. In *Programming Language Design and Implementation*, PLDI '97, pages 7–17, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. .

[11] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. .

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

[13] The Khronos Group. Typed Array Specification v1.0. `https://www.khronos.org/registry/typedarray/specs/1.0/`, February 2011.

[14] Veldema, Ronald and Jacobs, Ceriel J. H. and Hofman, Rutger F. H. and Bal, Henri E. Object Combining: A New Aggressive Optimization for Object Intensive Programs: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):439–464, Apr. 2005. ISSN 1532-0626. .

[15] C. Wimmer and H. Mössenböck. Automatic Feedback-directed Object Inlining in the Java Hotspot™ Virtual Machine. In *Virtual Execution Environments*, VEE '07, pages 12–21, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. .