What's up with monomorphism?

11 jan 2015 by Vyacheslav Egorov

Talks and blog posts about JavaScript performance often emphasize importance of *monomorphic* code. However they usually don't provide any digestible explanation of what monomorphism/polymorhism is and why it matters. Even my own talks often boil down to Hulk-style «**ONE TYPE GOOD. TWO TYPE BAD!!!**» dichotomy. Unsurprisingly one of the most common requests I get when people reach out to me for a performance advice is a request to explain *what monomorphism actually means*, how polymorphism arises and why it is bad.

It does not help that the word "polymorphism" itself is extremely overloaded. Within the classical object-oriented programming *polymorphism* usually means subtyping and ability to override behavior of the base class. Haskell programmers would think about parametric polymorphism instead. However polymorphism which JS performance talks warn against is somewhat different — it's a *call site polymorphism*.

I have explained this notion in so many different ways before that I finally decided to write a blog post about it - so that next time I can just link to it and not improvise.

[I also decided to try a new approach to explaining things - trying to capture interactions between various parts of the virtual machine in short comics. This is an new area for me, so please don't hesitate and send any feedback my way. Does it make it easier to understand? Does it make it harder to understand? This post also omits various details that I considered not important, redundant or only tangentially related - feel free to send questions my way if you feel I omitted something you really wanted to know]

Dynamic lookup 101

For simplicity this post will mostly concentrate on the simplest property access in JavaScript, like `o.x` in the code below. At the same time it's important to understand that everything we are going to talk about applies to any *dynamically bound* operation be it a property lookup or an arithmetic op and even goes beyond JavaScript.

```
function f(o) {
  return o.x
}

f({ x: 1 })
f({ x: 2 })
```

Imagine for a moment that you are interviewing for a great position at Interpreters Ltd. and your interviewer asks you to design and implement property lookup for a JavaScript VM. What would be the simplest and most straightforward answer to this question?

Obviously you can't go any simpler than taking JS semantics as it is described in the ECMAScript Language Specification (aka ECMA 262) and transcribing [[Get]] algorithm word by word from English into C++, Java, Rust or Malbolge depending on your language of choice for the interview.

In fact if you open a random JS interpreter you most likely will discover something like this:

```
jsvalue Get(jsvalue self, jsvalue property_name) {
  // 8.12.3 [[Get]] implementation goes here
}

void Interpret(jsbytecodes bc) {
  // ...
  while (/* has more bytecodes */) {
    switch (op) {
```

```
      // ...
      case OP_GETPROP: {
        jsvalue property_name = pop();
        jsvalue receiver = pop();
        push(Get(receiver, property_name));
        // TODO(mraleph): throw exception in strict mode per 8.7.1 step 3.
        break;
      }
      // ...
    }
  }
}
```

This is an absolutely valid way to implement property lookup, however it has one significant problem: if we pit our property lookup implementation against those used in modern JS VMs we will discover that it is far too slow.

Our interpreter is *amnesiac*: every time it does a property lookup it has to execute a generic property lookup algorithm, it does not learn anything from the previous attempts and has to pay full price again and again. That's why performance oriented VMs implement property lookup in a different way.

What if each property access in our program was capable of learning from objects that it saw before and apply this knowledge to similar objects? Potentially that would allow us to save a lot of time by avoiding costly generic lookup algorithm and instead use a quicker one that only applies to objects of certain *shape*.

We know that it is costly to figure out where the given property is inside an arbitrary object, so we would like to do this lookup once and then put the *path* to this property into a cache using *object's shape* as a key. Next time we see an object with the same shape we can just get the path from the cache instead of computing it from scratch.

This optimization technique is known as **Inline Caching** and I have written about it before. For this post I am going to leave concrete implementation details aside and will instead focus on an aspect I previously ignored: each inline cache is first and foremost **a cache** and just like any other cache it has *size* (number of currently cached entries) and *capacity* (maximum number of cached entries).

Lets take a look at the example again:

```
function f(o) {
  return o.x
}

f({ x: 1 })
f({ x: 2 })
```

What's the expected number of cached entries for IC at `o.x`?

Given that `{x: 1}` and `{x: 2}` have the same shape (aka _hidden class_ or _map_) the answer is 1. This is precisely the state of cache that we call *monomorphic* because it saw only objects of a single shape.

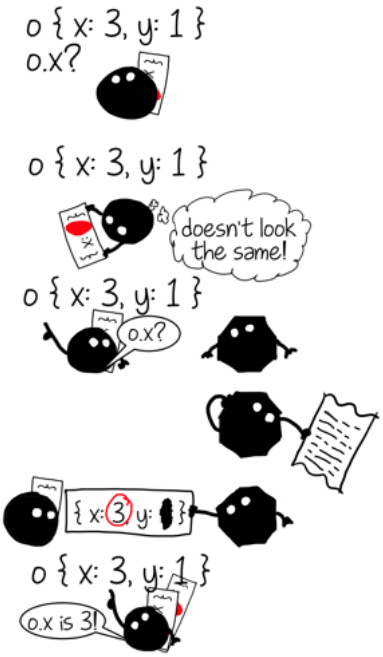
[m
a f

What happens if we now call `f` with an object of a different shape?

```
f({ x: 3 })
// o.x cache is still monomorphic here
f({ x: 3, y: 1 })
// what about now?
```

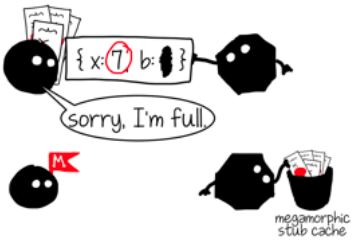
`{x: 3}` and `{x: 3, y: 1}` are objects of different shapes so the cache is no longer monomorphic, it now contains two cache entries one for a shape `{x: *}` and one for a shape `{x: *, y: *}` - our operation

now is in *polymorphic* state with a degree of polymorphism 2.



If we continue calling `f` with objects of different shapes it's degree of polymorphism will continue to grow until it reaches a predefined threshold - maximum possible capacity for the inline cache (e.g. `4` for property loads in V8) - at that point cache will transition to a *megamorphic* state.

```
f({ x: 4, y: 1 }) // polymorphic, degree 2
f({ x: 5, z: 1 }) // polymorphic, degree 3
f({ x: 6, a: 1 }) // polymorphic, degree 4
f({ x: 7, b: 1 }) // megamorphic
```



Megamorphic state exists to prevent uncontrolled growth of polymorphic caches, it means *"I have seen too many shapes **here**, I give up tracking them"*. In V8 megamorphic ICs can still continue to cache things but instead of doing it locally they will put what they want to cache into a global hashtable. This hashtable has a fixed size and entries are simply overwritten on collisions.

Now a small exercise to check understanding:

```
function ff(b, o) {
  if (b) {
    return o.x
  } else {
    return o.x
  }
}

ff(true, { x: 1 })
ff(false, { x: 2, y: 0 })
ff(true, { x: 1 })
ff(false, { x: 2, y: 0 })
```

1. How many property access inline caches are in the function `ff`?
2. What's state they are in?

Answers: there are 2 caches, both are monomorphic because each sees only objects of one shape.

Performance implications

At this point performance characteristics of different IC states should become clear:

- monomorphic is the fastest possible IC state if you hit the cache all the time (**ONE TYPE GOOD**);
- ICs in polymorphic state perform linear search among cached entries;
- ICs in megamorphic state probe global hash table and thus are slowest among ICs, but hitting global cache is still better than complete IC miss;
- IC miss is expensive - you have to pay for transitioning to runtime plus cost of the generic operation.

However this is only half of the truth: in addition to speeding up your code inline caches also serve as *spies* for almighty optimizing compiler — which will eventually come and try to speed your code up even further.

Speculations and optimizations

Inline caches can't deliver peak performance alone due to two issues:

- each IC acts on its own, knowing nothing about its neighbors;
- each IC can ultimately fallback to runtime if it can't handle its input: which means it's essentially a generic operation with generic side-effects and often unknown result type.

```
function g(o) {
  return o.x * o.x + o.y * o.y
}

g({x: 0, y: 0})
```

For example in the function above each IC (there are 7: `.x`, `.x`, `*`, `.y`, `.y`, `*`, `+`) will act on its own. Each property load IC will check `o` against the same cached shape. Arithmetic IC at `+` will check whether its inputs are numbers (and what kind of number - as V8 internally has different number representations) - even though this could be derived from that state of `*` ICs.

Arithmetic operations in JavaScript are inherently typed e.g. `a | 0` always returns 32-bit integer and `+a` always returns a number, but most other operations have no such guarantees. This turns writing an ahead-of-time optimizing compiler for JavaScript into an extremely difficult problem. Instead of compiling JavaScript once in an AOT fashion, most JavaScript VMs sport several execution tiers. For example in V8 code starts to execute without any optimizations, compiled with a baseline non-optimizing compiler. Hot functions are later recompiled by an optimizing compiler.

[as
typ
res
tha
typ
ah
ne
co

Waiting for code to warm up serves two distinct purposes:

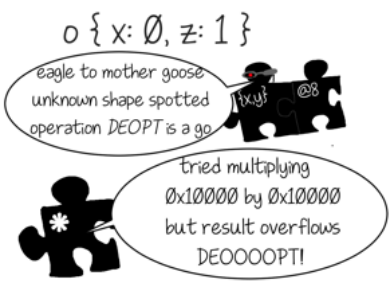
- it decreases startup latency: optimizing compiler is slower than non-optimizing, which means optimized code should be used enough for optimizations to pay off;
- it gives inline caches a chance to collect *type feedback*.

As it was already stressed above human written JavaScript usually does not contain enough inherent type information to allow full static typing and AOT compilation. JIT has to speculate: it has to make *educated* guesses about the usage and behavior of the code it optimizes and generate specialized code that is valid under certain assumptions. In other words compiler needs to guess what kind of objects are seen in a particular place in the function it optimizes. By a lucky coincidence that's precisely the information inline caches collect!

- Monomorphic cache says "I've **only** seen type A";
- Polymorphic cache of degree N says "I've **only** seen A_1, …, A_N";
- Megamorphic cache says "I've seen a lot of things.";

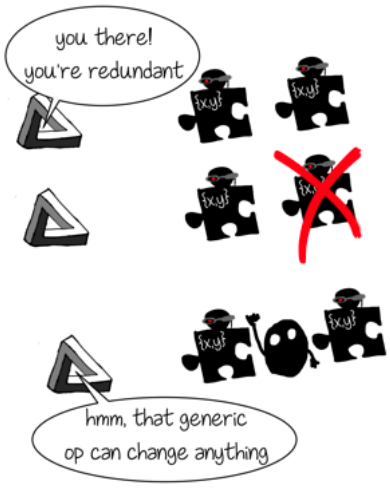
Optimizing compiler looks at the information collected by inline caches and builds *intermediate representation* (IR) accordingly. IR instructions are usually more specific and low level than generic JS operations. For example if IC for `.x` saw only objects of shape `{x, y}` then optimizer can take an IR instruction that loads property from a fixed offset inside an object and use it to load `.x`. Of course it is unsafe to apply such instruction to arbitrary objects so optimizer prepends a *type guard* before it. Type guard checks the shape of the object before it reaches specialized operation and if it does not match an expected shape the execution of the optimized code **can not** continue - instead we have to jump into the unoptimized code and continue execution there. This process is called *deoptimization*. Deoptimization reasons however are not limited to type guard violations: arithmetic operation can be specialized for 32bit integers and will deoptimize if result overflows this representation, an indexed property load `arr[idx]` can be specialized for a inbounds access and will deoptimize if `idx` is out of bounds or `arr` has no property `idx` (it's a *hole*), etc.

It should now become clear that the process of optimization tries to address two previously outlined weaknesses:



unoptimized	**optimized**
each operation has arbitrary unknown side effects because it is generic and implements full semantics	code specializations limit or eliminate unpredictability, side-effects are well defined (e.g. load of property by offset has no side-effects)

each operation is on its own, learns individually and does not exchange information with neighbors	operations are decomposed into lower level IR instructions which are then optimized together, this allows to discover and eliminate redundancies between them



Indeed building specialized IR based on the type feedback is just the first step in the optimization pipeline. Once IR is ready compiler will run multiple passes over it trying to discover invariants and eliminate redundancies. Analyses that are run at this stage usually are *intraprocedural* and compiler is forced to assume the worst arbitrary side-effects every time it encounters a call. Here it is important to realize that generic unspecialized operations are essentially *calls* themselves: e.g. + evaluation can call `valueOf` and a property access `o.x` can easily result in a getter invocation. This means any operation which optimizer for some reason failed to specialize completely might become a stumbling block for subsequent optimization passes.

One very common case of redundancy are repeated type guards that check the same value against the same shape. Here is how initial IR for function `g` (see above) could have looked like:

```
    CheckMap v0, {x,y}  ;; shape check
v1  Load v0, @12        ;; load o.x
    CheckMap v0, {x,y}
v2  Load v0, @12        ;; load o.x
i3  Mul v1, v2          ;; o.x * o.x
    CheckMap v0, {x,y}
v4  Load v0, @16        ;; load o.y
    CheckMap v0, {x,y}
v5  Load v0, @16        ;; load o.y
i6  Mul v4, v5          ;; o.y * o.y
i7  Add i3, i6          ;; o.x * o.x + o.y * o.y
```

This IR checks `v0` against the same shape 4 times even though there is nothing between checks that could affect the `v0`'s shape. Attentive reader will also spot that loads `v2` and `v5` are redundant too, as nothing is writing into corresponding properties. Fortunately the GVN pass that is later applied to the IR will eliminate redundant checks and loads:

```
    ;; After GVN
    CheckMap v0, {x,y}
v1  Load v0, @12
i3  Mul v1, v1
v4  Load v0, @16
i6  Mul v4, v4
i7  Add i3, i6
```

However as noted above such elimination is only possible because there are no interfering side effects between redundant operations: if there were a call between `v1` and `v2` we would have to conservatively assume that callee might potentially have access to `v0` and thus can add, remove and change properties - this would make impossible to eliminate access `v2` or `CheckMap` that guards it.

Now that we have a basic understanding of optimizing compiler and know what it likes (specialized ops) and dislikes (calls and generic ops) there is only one thing left to discuss: handling of non-monomorphic operations by the optimizing compiler.

If operation is non-monomorphic optimizing compiler obviously can't use that simple specialization rule `type-guard + specialized-op` we were discussing before. It simply will not be able to select a single

type for a type guard and single specialized operation. IC tells the compiler that this operation sees values of different types/shapes thus picking a single one of them and ignoring the rest means risking deoptimization, which is highly undesirable. Instead optimizing compiler will usually try to build a *decision tree*. For example a polymorphic property access `o.x` that saw shapes `A`, `B`, `C` will be expanded into something like this (note this is pseudocode - optimizing compiler would build a CFG structure instead):

```
var o_x
if ($GetShape(o) === A) {
  o_x = $LoadByOffset(o, offset_A_x)
} else if ($GetShape(o) === B) {
  o_x = $LoadByOffset(o, offset_B_x)
} else if ($GetShape(o) === C) {
  o_x = $LoadByOffset(o, offset_C_x)
} else {
  // o.x saw only A, B, C so we assume
  // there can be *nothing* else
  $Deoptimize()
}
// Note: at this point we can only say that
// o is either A, B or C. But we lost information
// which one.
```

One thing to notice here is that polymorphic accesses lack useful property that monomorphic accesses have. After specialized monomorphic access and until an interfering side-effect we can guarantee that object has a certain shape. This allows to eliminate redundancy between monomorphic accesses. Polymorphic accesses give a very weak guarantee “object’s shape is one of A, B, C”. We can’t use this information to eliminate much redundancy between two similar polymorphic accesses that follow each other (at most we could use it to eliminate the last comparison and deoptimization block - but V8 does not try to do this).

V8 however does build a more efficient IR if property is located in the same place in *all* shapes. In this case a polymorphic type guard will be emitted instead of the decision tree:

```
// Check that o's shape is one of A, B or C - deoptimize otherwise.
$TypeGuard(o, [A, B, C])
// Load property. It's in the same place in A, B and C.
var o_x = $LoadByOffset(o, offset_x)
```

This IR has one important benefit for redundancy elimination: if there are no interfering side effects between two `$TypeGuard(o, [A, B, C])` instructions then the second one is redundant just like in the monomorphic case.

If type feedback tells optimizing compiler that property access saw more different shapes than optimizing compiler considers feasible to handle inline then optimizer will instead build a slightly different decision tree that ends with a generic operation instead of deoptimization:

```
var o_x
if ($GetShape(o) === A) {
  o_x = $LoadByOffset(o, offset_A_x)
} else if ($GetShape(o) === B) {
  o_x = $LoadByOffset(o, offset_B_x)
} else if ($GetShape(o) === C) {
  o_x = $LoadByOffset(o, offset_C_x)
} else {
  // We know that o.x is too polymorphic (megamorphic).
  // to avoid deoptimizations leave escape hatch to handle
  // arbitrary object:
  o_x = $LoadPropertyGeneric(o, 'x')
  //    ^^^^^^^^^^^^^^^^^^^^ arbitrary side effects
}
// Note: at this point nothing is known about
// o's shape and furthermore arbitrary
// side-effects could have happened.
```

Finally in some cases optimizing compiler can completely give up on specializing operations:

- if it does not know how to efficiently specialize it;

- operation is polymorphic and optimizer does not know how to build a decision tree correctly for this operation (e.g. used to be the case for polymorphic keyed accesses `arr[i]` in V8 - but not anymore);
- operation does not have any actionable type feedback to specialize upon (operation were never executed, GC cleared collected type feedback, etc).

In all such (arguably rare) cases the optimizer just emits a generic variant of the operation in the IR.

Performance implications

Lets summarize what we learned:

- monomorphic operations are easiest to specialize, give optimizer most actionable information and enable further optimizations. Hulk-style summary **ONE TYPE CLOSE TO METAL**!
- moderately polymorphic operations which require a polymorphic type guard or in the worst case a decision tree are slower then monomorphic ones.
  - Decision trees complicate control flow and make it harder for optimizer to propagate types and eliminate redundancies. Memory dependent conditional jumps that constitute those decision trees might be bad news if polymorphic operation is right in the middle of the tight number crunching loop;
  - Polymorphic type guards don't obstruct type flow that much and still allow for some redundancy elimination - but each polymorphic type guard is still somewhat slower than monomorphic type guard (that checks against one shape). Performance penalty for a polymorphic type guard depends on how well CPU handles conditional branches;
- highly polymorphic/megamorphic operations are not specialized entirely and result in a generic operation being emitted as part of the optimized IR. This generic operation is a call - with all associated bad consequences for both optimizations and raw CPU performance.

Take a look at this microbenchmark trying to capture the difference between all these cases for a property access: monomorphic, polymorphic with matching property offsets (requires polymorphic type guard), polymorphic with different property offsets (requires decision tree), megamorphic.

Undiscussed

I have intentionally ignored some implementation details when writing this post to avoid making it too broad in scope.

Shapes

We did not discuss at all how shapes (aka hidden classes) are represented, computed and attached to objects. Check out my post on inline caches, some of my talks e.g. AWP2014 one to get the basic idea.

One important thing to realize here that the notion of *shape* in JavaScript VMs is a heuristical approximation. It's an attempt to dynamically approximate static structure hidden in the program. Things that are shaped the same for a human might not necessary have the same shape for the VM:

```
function A() { this.x = 1 }
function B() { this.x = 1 }

var a = new A,
    b = new B,
    c = { x: 1 },
    d = { x: 1, y: 1 }

delete d.y
// a, b, c, d all have DIFFERENT SHAPES for V8
```

Fluffy dictionary nature of JavaScript objects also makes it easy to create *accidental* polymorphism:

```
function A() {
  this.x = 1;
}
```

```
var a = new A(), b = new A(); // same shape

if (something) {
  a.y = 2;  // shape of a no longer matches b.
}
```

Intentional polymorphism

Even if you are programming a language that only allows you to instantiate fixed shape objects (Java, C#, Dart, C++, etc) from rigid classes you can still write polymorphic code:

```
interface Base {
  void doX();
}

class A implements Base {
  void doX() { }
}

class B implements Base {
  void doX() { }
}

void handle(Base obj) {
  obj.foo();
}

handle(new A());
handle(new B());
// obj.foo() callsite is polymorphic
```

Being able to write code against the *interface* and have this code process objects of different *implementation* is an important abstraction mechanism. Polymorphism in statically typed programming languages has similar performance implications to the ones discussed above.

Not all caches are the same

It might be good to keep in mind that some caches are not shape based and/or have lower *capacity* than others. For example cache associated with a function invocation is either uninitialized, monomorphic or megamorphic with no intermediate polymorphic state. Instead of caching function's shape which is irrelevant for an invocation it caches *invocation target* - that is function itself.

```
function inv(cb) {
  return cb(0)
}

function F(v) { return v }
function G(v) { return v + 1 }

inv(F)
// inline cache is monomorpic, points to F
inv(G)
// inline cache is megamorphic
```

If `inv` is optimized when inline cache for `cb(...)` invocation is monomorphic then optimizing compiler can potentially inline this call (which is very important for small functions on hot paths). When this cache is megamorphic optimizer will not be able to inline anything (it does not know *what* - as there is no single target) and will just leave a generic invocation operation in the IR.

This comes in contrast with method invocation expressions `o.m(...)` that are handled similarly to property accesses. ICs at method invocation sites have intermediate polymorphic state between monomorphic and megamorphic state. V8 is capable of inlining at monomorphic, polymorphic and megamorphic sites and it builds IR in the same way as for properties: choosing between a decision tree or a single polymorphic type guard before inlined function body. There is one limitation however: for V8 to be able to inline method call it needs *it to be part of the shape*.

```
function inv(o) {
```

```
  return o.cb(0)
}

var f = {
  cb: function F(v) { return v },
};

var g = {
  cb: function G(v) { return v + 1 },
};

inv(f)
inv(f)
// here inline cache is monomorpic, have seen only objects with
// a shape like f.
inv(g)
// here inline cache is polymorphic, seen objects with two different
// shapes: like f and like g
```

It might be surprising that `f` and `g` have different shapes above. This happens because when we assign a function to a property V8 tries (if possible) to attach it to object's shape instead of saving it directly on the object. In this example `f` has a shape like this `{cb: F}` i.e. the shape itself is pointing directly to the closure. In our previous examples we only had shapes that simply declared the presence of the property at a certain offset, however this shape also captures the value of the property. This makes V8's shapes similar to classes in an language like Java or C++ where class is essentially a set of fields *and methods*.

Of course if you later overwrite functional property with a different function V8 decides that this doesn't look like a class-method relationship and switches to a shape that reflects this:

```
var f = {
  cb: function F(v) { return v },
};

// Shape of f is {cb: F}

f.cb = function H(v) { return v - 1 }

// Shape of f is {cb: *}
```

Overall the topic of how V8 builds and maintains shapes (hidden classes) is worthy of a large separate post by itself.

Representing *path-to-the-property*

At this point it might seem that IC associated with property `o.x` is simply a dictionary mapping shapes to property offsets, something like `Dictionary<Shape, int>`. However this representation is way too narrow to be useful: property can reside on one of the objects within the prototype chain or be an *accessor property* (one with a getter and/or a setter). An interesting observation to make here is that accessor properties are in the certain sense more generic than normal data properties.

For example `o = {x: 1}` can be perceived as an object with an accessor property `x` that has a getter/setter accessing a hidden internal slot using VM intrinsics:

```
// pseudo-code reimagining o = { x: 1 }
var o = {
  get x () {
    return $LoadByOffset(this, offset_of_x)
  },
  set x (value) {
    $StoreByOffset(this, offset_of_x, value)
  }
  // both getter and setter are generated internally by VM
  // and are invisible to normal JS code.
};
$StoreByOffset(this, offset_of_x, 1)
```

In the light of this observation it becomes clear that IC should be more akin to `Dictionary<Shape, Function>`: mapping shapes to accessor functions that should be executed IC is hit. Such IC

representation would allow to optimize cases that were impossible to cover with a simplistic representation from the above (properties on the prototype chain, accessor properties and even ES6 proxy objects).

Premonomorphic state

Some ICs in V8 actually have so called *premonomorphic* state between *unitialized* and *monomorphic*. It exists to avoid compiling IC stubs for ICs that are only executed once. I decided to avoid discussing this state because it is a somewhat obscure implementation detail.

Final performance advice

The best performance advice is hidden in the title of Dale Carnegie's book "How to Stop Worrying and Start Living".

Indeed worrying about polymorphism is usually futile. Instead benchmark your code on a realistic dataset, profile it for hotspots and if any of them are JS related - check out IR that optimizing compiler produces.

If in the middle of your tight number crunching loop you see IR instruction called `XYZGeneric` or anything marked with red `changes[*]` (aka "changes everything") marker - then (only then!) it might be the right time to start worrying.

THE END