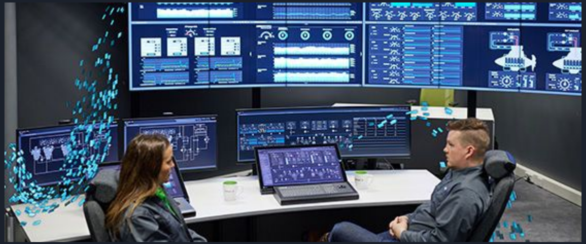
Nova JavaScript Engine - Exploring a Data-Oriented Engine Design

Web Engines Hackfest 2024
June 3rd, 2024
Aapo Alasuutari

About me and Nova

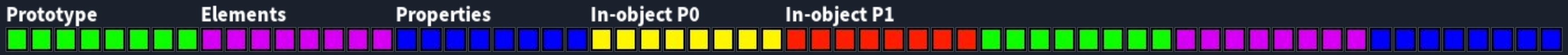


- Work at Valmet Automation as the Chief Design Engineer of the UI team
    - Also a freelance contractor for Deno Land Inc
- Not exceedingly performance critical, but user experience is very important
    - Poor animation performance really hurts us
    - See: Hummingbird HTML renderer by Coherent Labs
- Nova engine started from Andreu Botella's joke nearly 2 years ago
- 1 year ago I heard and got interested in data-oriented design and wanted to apply it to the engine

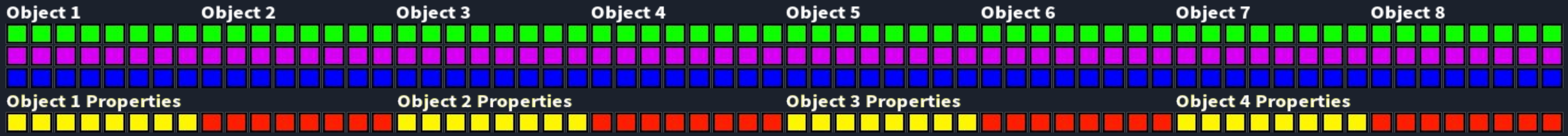
Data-oriented design

- Know your data, and how it is used
- Design your data structures for the most common use case
- Your program is not a one-off that touches one thing once
  - Loops, iterations, algorithms form the majority of your program's work: Think in multiples
- Aim to get the most out of your cache lines on the most common cases
- Ignore the singular case: It's a one-off and its performance is thus essentially meaningless

How quick is it to get P0 out of this object? What about P2?



How quick is it to map over P0's of 8 of these objects? What about P2's?



Improved cache line usage but at what cost?

- Object is no longer just a heap pointer, as its data is spread out over multiple cache lines
- Object need not be bigger: Use parallel vectors to store object data! Object is an index!
- All objects' heap data must now be same size for the parallel vector to work
  - Either all objects heap data is as big as the biggest, or…
  - All exotic objects get their own heap vectors!
  - Embedder slots are not a thing
- Value is a tagged union containing a byte-size tag and a heap index or stack data
  - Currently 64 bits in size, can take down to 32 bits if indexes only go up to 16.7 million
- Better cache line usage, that's it I guess?
  - No! Demand more! Think of the common use cases!
  - Array? Prototype not needed! Properties not needed!
  - ArrayBuffer? None of the usual stuff is needed!

Array heap data in Nova

```
struct ArrayHeapData {
    pub object_index: BackingObjectOrRealm,
    pub elements: SealableElementsVector,
}
```

- 16 (4 + 12) bytes, 4 on a cache line; can be split and minimized into 4 + 8 bytes
- Common case is to access elements or length
  - Pessimise prototype and properties access into “ordinary object” backing store
  - Optimise for mapping over multiple items
  - If split and minimized, that’s 8 elements pointers + lengths per cache line read!
- Elements also live in heap vectors!
  - Imagine length 2 element arrays: Object.entries() of 8 keys-values on the same cache line

```
struct ElementArray2Pow8 {
    pub values: Vec<Option<[Option<Value>; usize::pow(2, 8)]>>,
    pub descriptors: HashMap<ElementIndex, HashMap<u32, ElementDescriptor>>,
}
```

Improved cache line usage but only if the items are on the same cache line

- Axiom of GC systems: Most objects die young
  - Corollary: Most objects live together!
- All heap data of type T is created in the same Vec<T>
- Upon GC, the Vec<T> is drained of unreachable items and items are shifted down
  - Vectors are always packed
  - Data that was created together stays together
- All intra-heap indexes (references) must be realigned after GC
  - Small mercies: This is simple to calculate and is an embarrassingly parallel algorithm
- As a consequence, the heap does not generally fragment over time
- Nice benefit: post-GC high water mark acts as the nursery separator for Nth GC after current
  - No need for a separate nursery!

In conclusion, the upsides

- Excellent cache line usage (potential) for a dynamically typed language
- Vector-based heap is simple to reason about and provides interesting opportunities
- Tagged union based Value requires no pointer shenanigans, does not leak heap pointers and does not suffer from type confusion attacks
  - The 64-bit version can also carry quite a bit of on-stack data (i56, char[7])
- Properly exotic objects have a really easy time separating their concerns from general object concerns

And then the downsides

- Pessimising odd cases does pessimise them!
  - Objects hold no elements; indexed properties are just properties
  - Arrays with named properties need an extra indirection to get to the backing object
- Special internal slot cases, shared ownership of internal data forces either yet another heap vector or a pessimising of the common case
  - Promise resolving, rejecting functions: Do all BuiltinFunctions have the extra internal slots? Or are these special functions in the engine?
  - Promise Capability Record is either a reference counted pointer, or yet another heap vector
- Each exotic object requires a new implementation of internal methods
  - No inheritance, accessing heap data is always different: Implementations are all very similar but different
- Performance of GC remains to be proven
  - Can the heap vector compaction and index realigning be fast enough to be competitive?
  - As the heap size grows, this only gets worse

Q&A

https://github.com/trynova/nova

https://www.youtube.com/watch?v=WKGo1k47eYQ

Come talk to me and/or Andreu afterwards

Bonus: Where are we?

- Andreu working on test262 runner
  - ~3% passing! :)
  - Those are likely just the parser tests, oxc_parser passes them for us :(
- Can run trivial scripts with loops and if-elses
- Cannot run for-let / for-const
- No Promises, no jobs (callbacks)

Bonus: Rebels without a cause?

- In order of importance / complexity:
  - Have fun ✅
  - Try out and prove data-oriented design in a JavaScript engine context
  - Get a simplified, embeddable JS engine that can be feature-flagged to eg. drop out unneeded complexity like array-object features
  - Serve our personal websites using Nova
  - Secret (shh): Become Servo’s JavaScript engine
  - Super-secret (shhhh!): Become Deno’s JavaScript engine
  - Cause a revolution in engine and ECMAScript design
  - Take over the world

Bonus: Potential BuiltinFunction call API

- Usual function call API is uniform, and only optimised code may have some “Fast API” that is more exact
- What if instead we enumerate up to 2^8 different call APIs?
  - Takes this_value
  - Takes new_target (optional or not?)
  - 0..N parameters
- Interpreter copies the enumeration and the function pointer, and optimises work based on this
- Could this provide a “Fast API light” by default?
- What about parameter type definitions?

Bonus: GC marking, sweeping, Rust, and thread-safety

- Marking: N marker threads run through vectors of incoming indexes for Vec<T>, generate new work of based on items found in these items. New work goes into Vec<Index<T>>, finally gets sent to the proper thread / put into a work pool.
- Sweeping: Stop the world, re-mark from dirty items, then calculate for each Vec<T> which items need to shift down and by how much. Farm this data out to N threads and shift items down; dropped items get overridden, live items have their internal indexes shifted down.
- One writer, multiple readers working on data; Rust doesn't like this! The Heap must be filled with locks or Atomics
  - Choose Atomics!
- Define own wrappers around Atomics
  - MutatorOnly (non-atomic, only mutator can read&write)
  - Mutable (atomic, only mutator can write, readers can read)
  - Take advantage of wrapper-provided proofs to generally allow safe usage of Relaxed (though not always)
- Three states of the engine:
  - Mutator only, readers do not exist: Mutator can do work without atomics, but this is mostly not useful
  - Mutator & Readers, stop-the-world allowed: Values cannot live on stack, require indirection
  - Mutator & Readers, stop-the-world disallowed: Values can live directly on stack, use this for tight algorithms
- Use Rust type system to separate the states, prove Value lifetime on stack