How to lose weight?

Aapo Alasuutari, FOSDEM 2025

About me

- Work at Valmet Automation
  - Software architect for a browser based automation UI platform
  - TypeScript developer by day
- Avid choir singer
- OpenSource enthusiast, contributor, albatross
- Data-oriented design zealot
- Developing Nova JavaScript engine
  - Rust developer by night!

What do I mean by weight?

- Amount of memory used at runtime
- Amount of code
- Complexity of program and required context knowledge to understand it
- For any program, there exist an ideal weight
- **Losing weight is about reasonably approaching ideal memory weight**
- We can calculate the ideal memory weight!
  - **Shannon entropy** or **“How surprising is the data?”**
  - How many Y/N questions must be answered correctly to understand data?
  - = How many bits of information does the data hold!

A dose of ground truth

- A JavaScript Value is usually a reference to heap data
- Size of a Value depends only on the “engine”
  - 4 bytes in Chrome, 8 bytes in Firefox, Safari, Node.js, Deno, Bun
- Booleans and 32-bit integer numbers are always stack-only
  - All numbers are stack only in Firefox, Safari, Bun; Chrome supports up to 31-bit integers on stack
- Heap data always has an “header” reference + data - *(size in Node v23.4.0, size in Chrome)*
  - Number: 8 byte IEEE-754 double precision floating point value - *(16b, 12b)*
  - String: Length + string bytes - *(16b+, 12b+)*
  - Symbol: Description string reference + engine specific data - *(24b, 16b)*
  - Object: 2 + N * property count references - *(24b, 12b)*
  - Array: Object + 32-bit integer length (rounds up to reference) - *(32b, 16b)*
  - Map/Set: Object + 1 + N * prop count * C - *(32b, 16b)*
  - ArrayBuffer: Object + buffer reference, byte length, max byte length, detach key, … - *(96b, 52b)*
  - Uint8Array: Object + ArrayBuffer reference, byte length, byte offset, length, .. - *(104b, 60b)*

Cut to the chase! How to lose weight?

- Remove things you don’t need
- Get rid of booleans
- Use context knowledge
- Choose appropriate data sizes / storage
- Get rid of heap “headers” and object data mostly or entirely
- **Struct-of-Arrays enables these!**

Let’s look at code!

In conclusion

- **Losing weight is about reasonably approaching ideal memory weight**
- **Struct of Arrays enables using the tools that JavaScript has**
  - Real life examples: 80 MiB => 9 MiB, 350 MiB => 15 MiB, 10 MiB => 3 MiB
- Wait: TypedArray is massive compared to an Array! Isn't that bad?
  - Yes! Fundamental problem with OOP: Subclass is can only grow in size
  - Without ArrayBuffer sharing, a TA breaks even at between 20 and 50 items
- **What if the JavaScript engine lost weight?**
  - Enter Nova JavaScript engine!
  - Approach ideal memory weight at the cost of code and complexity
  - Subclass objects no longer grow in size
    - 32b (12-16b) for ArrayBuffer, 20b (12-16b) for TypedArrays

Q&A

https://github.com/aapoalas/losing-weight

Nova JavaScript Engine

https://trynova.dev/

https://github.com/trynova/nova

Post-scriptum 1: General weight-loss tricks

- Removing booleans:
    - Why? 1 bit stored in 32 or 64 bits! 1.6-3.1% memory utilization!
    - Why? Adding booleans produces combinatorial state explosion with impossible states
    - Split group of objects with boolean property to two groups of objects
        - Remember to remove properties made unnecessary by the split, if reasonable
    - Use Set/WeakSet membership to signal rare boolean true
- Choosing column type in Struct-of-Arrays
    - Numeric data: Appropriate TypedArray
    - Unique strings: Array of strings (normal Arrays are perfectly fine sometimes)
    - Enums: Appropriate TypedArray
    - Index to Array / another SoA: Uint32Array when in doubt, use smaller if you can
    - Boolean: Uint8Array of bytes or bits, or a Set of non-default indexes for if only rarely true/false
    - Rarely set value: Map of indexes to values
    - Rarely non-default value: Map of indexes to non-default values
    - Rarely non-small value: TypedArray + Map<index, value>, use sentinel in TA if full value is in Map
    - Nullable values: Use a sentinel value (eg. maximum value) in TypedArray as “null”
- Hint: Allocate common ArrayBuffer for all TypedArrays to share in the same SoA

Post-scriptum 2: String weight-loss tricks

- Repetitive strings:
  - Store in an array of unique strings
  - Refer to array by index using the smallest possible TypedArray
  - Eg. 200 strings => Uint8Array, 300 strings => Uint16Array
  - Re-check size when adding strings, change to larger TypedArray if necessary
  - Equality check is just index equality check
  - Effectively: Turns strings into a dynamically generated enum
- Unique strings made out of repetitive parts:
  - Split strings into parts to get to the repetitive parts (remove statically known parts!)
  - Store each “kind” of part separately in an array of unique strings and refer by index
  - If number of parts is known, works excellently: As many indexes as there are parts
  - If number of parts is variable, store indexes in an Array or side-table and refer by index + length
  - Equality check is length equality and each index’ equality check
  - Effectively: Turns strings into a dynamically generated sum/combination/struct of enums

Post scriptum 3: Does it make sense?

- Not every memory optimisation makes sense! Depends on your use-case
- As data becomes more dynamic, SoA and other related optimisations become harder to manage
    - At some point you might have to start thinking about implementing your own garbage collection!
- Separating data into columns makes iterating over rows of a column faster but makes cross-column work slower.
    - Eg. Think of a box defined as { x0, x1, y0, y1 }:
    - SoA makes looking for a box by a point faster but calculating the box’s circumference slower
    - Know your common case!
- Converting a boolean to Set membership makes lookup slower (maybe)
- Measure, measure, measure!