General Notes

Each Array costs 70 bytes and is composed of `Array` and `(array)` object * `Array` javascript visible object: 32 bytes * `(array)` VM object where the array is actually stored in: 38 bytes

Each Object cost is 24 bytes plus 8 bytes per property.

For small arrays, it is more efficient to store the data as a linked list of items rather than small arrays. However, the array access is faster as shown here: https://jsperf.com/small-arrays-vs-linked-objects

Monomorphic vs Megamorphic code

Great reads: - What's up with monomorphism? - Impact of polymorphism on component-based frameworks like React

1) Monomorphic prop access is 100 times faster than megamorphic.
2) Monomorphic call is 4 times faster the megamorphic call.

See benchmark here.

Packed vs. holey Array

V8 represents arrays internally in a different way depending on: - type of elements in the array; - presence of holes (indexes that were never assigned).

Generally speaking packed arrays (a set of continuous, initialized indexes) perform better as compared to arrays with holes. To assure that arrays are packed follow those guidelines: * create array literals with known values whenever possible (ex. `a = [0];` is better than `a = []; a.push[0];`; * don't use `Array` constructor with the size value (ex. `new Array(5)`) - this will create a `HOLEY_ELEMENTS` array (even if this array is filled in later on!); * don't delete elements from an array (ex. `delete a[0]`) - this will create a hole; * don't write past the array length as this will create holes;

Great reads: - Elements kinds in V8

Exporting top level variables

Exporting top level variables should be avoided where possible where performance and code size matters:

```
// Typescript
export let exported = 0;
let notExported = 0;

notExported = exported;

// Would be compiled to
```

```
exports.exported = 0;
var notExported = 0;

notExported = exports.exported;
```

Most minifiers do not rename properties (closure is an exception here).

What could be done instead is:

```
let exported = 0;

export function getExported() { return exported; }
export function setExported(v) { exported = v; }
```

Also writing to a property of `exports` might change its hidden class resulting in megamorphic access.

Iterating over Keys of an Object.

https://jsperf.com/object-keys-vs-for-in-with-closure/3 implies that `Object.keys` is the fastest way of iterating over properties of an object.

```
for (var i = 0, keys = Object.keys(obj); i < keys.length; i++) {
  const key = keys[i];
}
```

Recursive functions

Avoid recursive functions when possible because they cannot be inlined. https://jsperf.com/cost-of-recursion

Function Inlining

VMs gain a lot of speed by inlining functions which are small (such as getters). This is because the cost of the value retrieval (getter) is often way less than the cost of making a function call. VMs use the heuristic of size to determine whether a function should be inline. Thinking is that large functions probably will not benefit inlining because the overhead of function call is not significant to the overall function execution.

Our goal should be that all of the instructions which are in template function should be inlinable. Here is an example of code which breaks the inlining and a way to fix it.

```
export function i18nStart(index: number, message: string, subTemplateIndex?: number): void {
  const tView = getTView();
  if (tView.firstCreatePass && tView.data[index + HEADER_OFFSET] === null) {
    // LOTS OF CODE HERE WHICH PREVENTS INLINING.
  }
}
```

Notice that the above function almost never runs because `tView.firstCreatePass` is usually false. The application would benefit from inlining, but the large code inside `if` prevents it. Simple refactoring will fix it.

```
export function i18nStart(index: number, message: string, subTemplateIndex?: number): void {
  const tView = getTView();
  if (tView.firstCreatePass && tView.data[index + HEADER_OFFSET] === null) {
    i18nStartfirstCreatePass(tView, index, message, subTemplateIndex)
  }
}
export function i18nStartfirstCreatePass(tView: TView, index: number, message: string, subTe
  // LOTS OF CODE HERE WHICH PREVENTS INLINING.
}
```

Loops

Don't use `forEach`, it can cause megamorphic function calls (depending on the browser) and function allocations. It is a lot slower than regular `for` loops

Limit global state access

Ivy implementation uses some variables in `packages/core/src/render3/state.ts` that could be considered "global state" (those are not truly global variables exposed on `window` but still those variables are easily accessible from anywhere in the ivy codebase). Usage of this global state should be limited to avoid unnecessary function calls (state getters) and improve code readability.

As a rule, the global state should be accessed *only* from instructions (functions invoked from the generated code).

Instructions should be only called from the generated code

Instruction functions should be called only from the generated template code. As a consequence of this rule, instructions shouldn't call other instructions.

Calling instructions from other instructions (or any part of the ivy codebase) multiplies global state access (see previous rule) and makes reasoning about code more difficult.