










# Turboshift Frontend - Preliminary Design Elements

Authors: dmercadier@  
 Date created: 02/11/2023  
 Last edited: 15/11/2023  
 Visibility: public

## Turbofan Frontend Overview

The following table lists all of the phases that Turbofan's front-end currently contains, along with their current status in Maglev and Turboshift, and some notes about their implementations in Turboshift and Maglev.

Main phase	Sub-phases	TF SLOC	Description / Notes	Maglev Status	Maglev Notes	TS Status	Turboshift Notes
Graph Builder	Graph building	3543	Walks bytecode and builds the graph from bytecode		Written for Maglev, would need to be extended for Turboshift		
	JSTypeHintLowering	704	Early feedback-based specialization				
Inlining & ContextSpecialization	NativeContextSpecialization ContextSpecialization	3600	Lowers Loads/Stores into specialized operations based on Context		Written for Maglev, would need to be extended for Turboshift		
	JSCallReducer	6950	Inlines builtins fast-paths				Turboshift has good tools to generate IR
		Inlining	1428	Inline JS functions		Only allows greedy inlining, would need to be rewritten for priority-based inlining	
Typer & Typed Lowering	Typer	2102			Partially written		Only machine-level typer and typed-optimizations currently exist
	JSCreateLowering	1701	Lowers CreateXXX nodes		Does not compute fix-point, no loop backedge information		
	JSTypedLowering	2124	Type-based lowering of generic operators (eg, JSAdd)		Probably doesn't need/want an expensive fix-point analysis		
	TypedOptimization	947	Type-based eliminations of Checks (eg, CheckString) and type-based specializations of				

			generic operators				Part of these are implemented in Machine Optimization Reducer
	SimplifiedOperatorReducer	349	Peephole optimizations on Simplified operators				
	CommonOperatorReducer	505	CFG optimizations, mostly replaces Branches by Gotos				
Loop Peeling & Unrolling	LoopPeeling	990	Peels the 1st iteration of innermost loops		Only allows greedy loop peeling.		Required for Wasm as well
	LoopUnrolling	655	Exists in Turbohaft but not Turbofan		Probably not hard to write		
Load Elimination	BranchElimination	357	CFG optimizations, mainly double-diamond elimination and <code>if(c){if(c){}}</code> optimization		Partially written		Easy to write
	RedundancyElimination	452	Eliminates impure redundant operations, like redundant Checks.		Does not compute fix-point, no loop backedge information		
	LoadElimination	1551	Eliminates redundant loads		Probably doesn't need/want an expensive fix-point analysis		Mostly already written Needs small modifications for front-end Needs to do Check-elimination as well — Required for CSA and Wasm as well
	ValueNumberingReducer	157	Eliminates redundant pure operations				
	CommonOperatorReducer		<a href="#">See above</a>				Mostly already written
	TypedOptimization		<a href="#">See above</a>				
Escape Analysis	EscapeAnalysis	1418	Avoid constructing objects that don't escape the current function		Needs tracking of dematerialized object for the deopt info		Deopt support already in place
<hr/>							
Simplified Lowering	SimplifiedLowering, RepresentationChange	6353	Lowers JS operators to Simplified operators		Maglev has some representation analysis (in particular around Phis), but this is a fast pass that is less powerful than what Turbohaft will need (and Maglev probably doesn't need more)		

## Legend

- ✓ implemented (or almost fully implemented)
- ◻ partially implemented (would need non-trivial modifications/extensions to work for Turbohaft)
- ✗ not implemented

Total TF SLOC (excluding Simplified Lowering): 28878

# New Frontend Phases and High-level Considerations

Which phases should be grouped together, and which shouldn't be?

**Graph Building** should be done before inlining, because we want to have access to the whole graph before choosing what to inline. But it's nice to do some `NativeContextSpecialization` while building the graph, so that it doesn't need to be done before starting Inlining.

**Inlining and `NativeContextSpecialization`** need to be done together so that after inlining a function, its users can be optimized with `NativeContextSpecialization`, which would in turn open inlining opportunities.

**Typing** (+ typed lowering, typed optimizations), **Load Elimination and Escape Analysis** have to run an analysis over the whole graph. This should be done after inlining is finished. Still, doing a bit of load elimination and typing during graph building could be useful in order to guide the inlining decisions (but "a bit" = not the full analysis).

**Loop peeling and unrolling** are probably somewhat flexible: we might be able to do them during graph building or later (doing them later is probably a little bit easier because they can then operate on a graph rather than having to look at bytecode). It's somewhat important to do Loop peeling before Load Elimination, since this might allow some loads to be hoisted out of loops.

We would end up with 3 main distinct phases:

- Graph Building. It would be nice if it contained `NativeContextSpecialization`.
- Inlining. This has to include `NativeContextSpecialization`.
- Optimizations (Type-based, LoadElimination, Escape Analysis, Loop Peeling/Unrolling).  
This phase could easily be split into multiple sub-phases.

(+ a final `SimplifiedLowering`-like phase)

## What about Maglev's Typing, Load Elimination and Escape Analysis?

These 3 phases will require some kind of fairly expensive analysis to be optimal (iterating at least twice over loops, or maybe until a fixpoint), which could be too expensive for Maglev.

Additionally, Maglev's current approach to Typing and Load Elimination is to do them online while building the graph. This is not suitable for Turbohaft, where loops would need to be revisited (and their content changed based on what we learned on the backedge).

That being said, if we wanted to reuse Maglev's graph builder, we could keep these Maglev optimizations, given that they still improve the graph. However, I'm not sure this is the best solution w.r.t. code complexity; see [Mixing Phases or Splitting Phases](#).

## Lowering Early or Late

**Benefits of lowering early:** lowering early can enable subsequent optimizations to do more. For instance, lowering an `Array.map` before inlining is probably a good idea, because it could allow inlining the callback function.

Similarly, lowering early can reduce the effects of an operation, which in turn can make subsequent optimizations/analysis (such as Load Elimination) more effective. For instance, lowering a `JSAdd` into a `Word32Add` during graph building based on feedback is probably beneficial, because it reduces the effect of the operation (`JSAdd` can probably lead to arbitrary JS code execution, while `Word32Add` is pure).

**Benefits of lowering late:** lowering late can also lead to more optimization opportunities in the Frontend, in particular dead-code elimination (it's easier to remove an unused `StringConcat` than a subgraph allocating a string and copying 2 strings in it), GVN (it's easier to GVN 2 consecutive identical `StringConcat` than 2 subgraphs performing the same `StringConcat`) and BranchElimination (branches whose conditions are high-level nodes are easier to eliminate than branches whose conditions are the result of some large subgraph computation).

**Disadvantages of lowering early:** lowering early tends to obfuscate the graph, which can make optimizations harder to do / spot. For instance, it's quite obvious that `TagSmi(UntagSmi(x))` should be optimized to `x`. However, once lowered, this becomes `OverflowCheckedAdd(ShiftRightArithmeticShiftOutZeros(x, 1))`, which wasn't optimized until recently.

Additionally, information can be "lost" through lowerings. For instance, something that was known to be a `Smi` can look like any Tagged value after a lowering, which means that subsequent optimizations won't be able to take advantage of this fact.

## Mixing Phases or Splitting Phases

Here is a Maglev loop for the lowering of `ArrayForEach`:

<https://source.chromium.org/chromium/chromium/src/+main:v8/src/maglev/maglev-graph-builder.cc;l=5605-5797;drc=7b59899e869fd1520ca45c6eb9f418402ec9ce59>

And here is the equivalent Turbofan loop:

<https://source.chromium.org/chromium/chromium/src/+main:v8/src/compiler/js-call-reducer.cc;l=1517-1558;drc=6ee16b764a2e267e20a3b25163b5077656eda9a3>

The Maglev one does more (tracking maps and types), and produces better code. The Turbofan one is much simpler and relies on subsequent passes (CheckElimination, LoadElimination, SimplifiedLowering) to optimize the not-so-optimal code it generates.

Maglev doesn't have the luxury of relying on subsequent passes to optimize poor code, but for Turbohaft, we could consider taking a similar approach as Turbofan, in order to simplify the code (since, anyways, we should have a powerful load elimination and representation selection coming later, which should be able to optimize away everything that Maglev optimized away).

## Inlining with Maglev/Turbohaft IR

The way inlining should work is how it currently works in Turbofan: pick the hottest function call, inline the function, and optimize (recursively) the uses based on what the inlined function body contains; repeat this process until the inlining budget is exhausted.

This does not work naturally well with Maglev and Turbohaft IR, since they don't support many in-place mutations (in Maglev, nodes can be inserted in some places and sometimes mutated, but changing control flow is hard, and in Turbohaft we can just overwrite a node with a node of the same size or smaller).

Still, it's possible to adapt Turbohaft (and Maglev) to support in-place mutation. [Turbohaft JS Inlining and In-place mutation](#) explains how to achieve this, and a [prototype CL](#) demonstrates feasibility of in-place mutations of the IR. The proposed solution has the downside of increasing the complexity of the Assembler (which was already fairly complex), which makes it not so ideal for long-term maintenance.

Another thing to keep in mind: if we plan on inlining JS in Wasm and vice-versa, then doing the inlining in Turbohaft would be convenient. However, such general-purpose JS-Wasm inlining is not in our short-to-medium term plans (Jakob Kummerow said that pattern-based inlining for specific JS/Wasm functions might be useful, but it's not clear whether the general thing is useful).

## Useful Turbohaft Features

Some phases of the front-end do generate quite a lot of code (in particular [JSCallReducer](#) but also [NativeContextSpecialization](#)). Turbohaft has some features that make it easier/safer to write code; whichever IR we choose for the beginning of the frontend, similar features would be nice.

### Static C++ node types

Instead of using `OpIndex` (the regular "node" type), one can also use `V<type>` (defined in [turbohaft/index.h](#)), where `type` is any JS type + some machine-level types like `Word32`, `Float64`, etc. If `type1` is implicitly convertible to `type2`, then `V<type1>` is implicitly convertible to `V<type2>`. This is quite useful to 1) document the code and the types of the variables, and 2)

detect bugs early (if a `V<Tagged>` is used as input for a function that expects a `V<Float64>`, something is obviously wrong, and we'll get a Clang compile-time error). (For those familiar with them: this is all very similar to `TNode<type>` in CSA.)

## AssemblerOpInterface helpers

The `AssemblerOpInterface` (in [turboshift/asmblar.h](#)) defines a lot of helpers to emit operations. Some are just syntactic sugar (like `Word32Add(a, b)` which expands to `WordBinop(a, b, Kind::kAdd, RegisterRepresentation::kWord32)`), and others have more complex lowering (like `CallBuiltin`, which generates a `Call` based on a `CallDescriptor`, doing some checking on the arguments along the way). This allows to have fairly generic Opcodes (like `kWordBinop` rather than `kWord32Add`, `kWord64Add`, `kWord32Sub`, ...), which in turns allows to generically handle multiple low-level cases, while at the same time being able to easily emit seemingly low-level code.

Additionally, thanks to the `ConstOrV` extension of `V<type>`, constants can often be passed as C++ constants and are automatically wrapped in Turboshift nodes (eg, one can write `__ Word32And(x, 1)` rather than `__ Word32And(x, __ Word32Constant(1))`).

## GraphGen Macros

Turboshift has a number of macros to help generate Turboshift graphs (defined in [turboshift/define-asmblar-macros.inc](#), although the most of the implementation is in [turboshift/asmblar.h](#)). The best examples of what can be done with these macros are in [turboshift/machine-lowering-reducer-inl.h](#). The main features are:

- `LABEL` to automatically generate phis and ensure that all predecessors of a label are feeding correctly-typed inputs for all of the phis.
- `IF/ELSE` to write structured code without using `GotoIf` and `Branch` all over the place.
- `LOOP` to define loops without having to manually handle loop phis or to manually maintain the only-1-backedge invariant.
- `LIKELY/UNLIKELY` to annotate branches are likely or unlikely.

## VariableReducer (SSA generation)

The `VariableReducer` provides 3 functions: `NewVariable`, `SetVariable` and `GetVariable` (see [turboshift/variable-reducer.h](#)). Tracking changes of a global(ish) value can be done by using these functions: the `VariableReducer` will automatically insert `Phi`s on merges when predecessors of a block have different values for a `Variable`.

A good example of this is the `MemoryOptimizationReducer` ([turboshift/memory-optimization-reducer.h](#)). For allocation folding, it uses a `Variable` to store the allocation top, and thus `Phi`s are automatically inserted after calling `Allocate` to update the allocation top.

Variables can also be used for small lowerings to avoid having to deal with `Phi`s manually (see for instance [turboshift/select-lowering-reducer.h](#)).

Here is a CL showing how having the GraphGen Macros and the VariableReducer can simplify lowerings: <https://crrev.com/c/4675295>.

## Safe unreachable code emission

Most of the time, lowerings don't have to check whether they are emitting unreachable code or not. In particular, if a lowering L contains something like

`IF (c) { ... } ELSE { ... }` and a reducer replaces the conditional branch by a Goto because `c` is actually a Constant, L doesn't need to check anything at the beginning of the IF or ELSE block: the Assembler will allow the reducer to keep trying to emit code and will just not emit anything.

As a bonus, in some cases, code looks unreachable because a BIND was forgotten; in such cases, the Assembler will most of the time crash rather than silently not emit anything.

Without this feature, every Branch would have to be followed by a Check checking whether the destinations are reachable.

(see [this useful comment in turboshaft/assembler.h](#))

## Automatic Edge Splitting

Turboshaft requires the graph to be in split-edge form (amongst other things, to allow memory-efficient storage of predecessors). Whenever emitting a Branch would break the split-edge form, the Assembler automatically splits the edge by inserting an intermediate block (see [turboshaft/assembler.h](#)). Note that to avoid creating unnecessary blocks, edges are split lazily when emitting a Branch would break the split-edge form rather than for all Branches. (only relevant if the IR requires split-edge form of course)