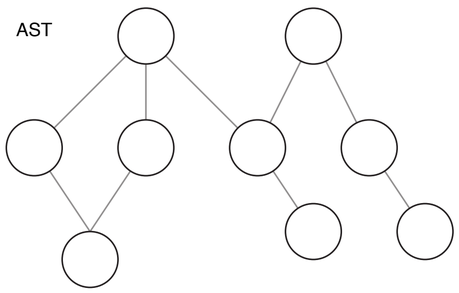
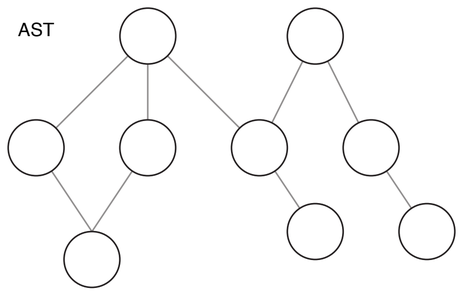
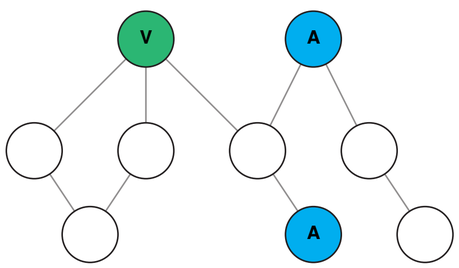
WEFT Render Graph

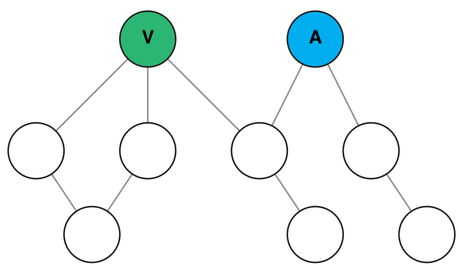




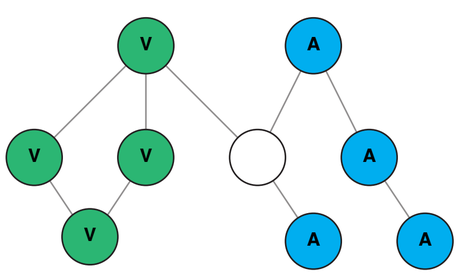
THE PROBLEM:
WEFT is by design domain-agnostic. You can talk about signals, and they dont have an intrinsic “medium” until they are displayed. But this isn’t how computers work. In order for WEFT to be useful at all, we need visual stuff to be computed on the GPU, audio stuff to be computed in CoreAudio or WebAudio, etc. This means that different parts of the AST have to be transpiled to different languages to be executed. Even worse, all of these different contexts can call each other. So I may have a chunk of visual code that relies on an audio sample. That means I need to compute the audio before the visual. You get the gist of the problem.

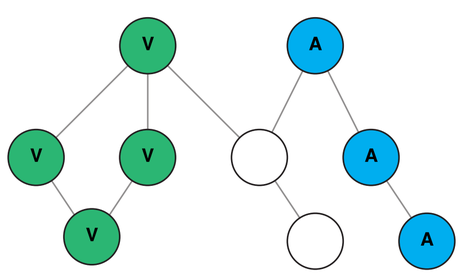
THE SOLUTION:
Walk the AST and sort what we can, divvying up chunks of AST to be sent to their respective transpilers and executed. Specifically:



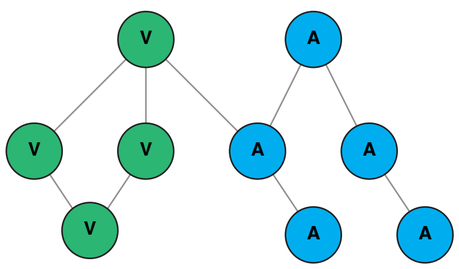


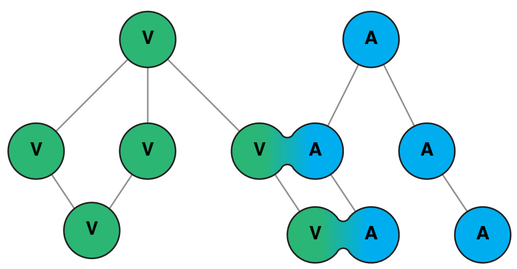
STEP 1
Assign a type to known nodes (things like microphone_in, render_pixel, etc) where the type is known beforehand (we should not try to compute render_pixel on an audio thread).



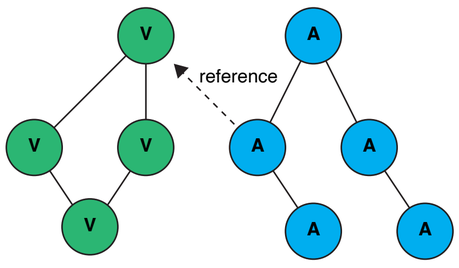


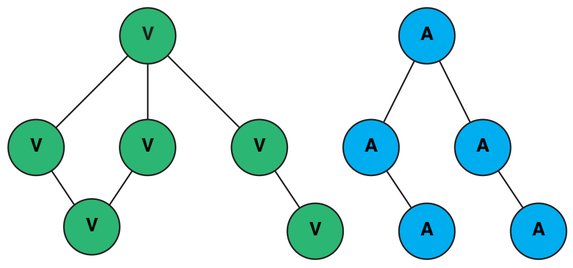
STEP 2
Propagate contexts bidirectionally through the dependency graph:
a. Bottom-up: if all your dependents (nodes that use you) have the same context, you inherit it
b. Top-down: if all your dependencies (nodes you use) have the same context, you inherit it



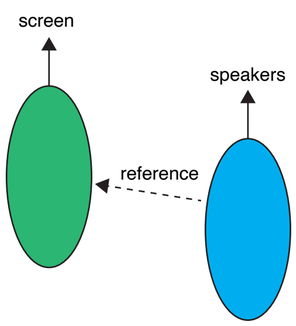


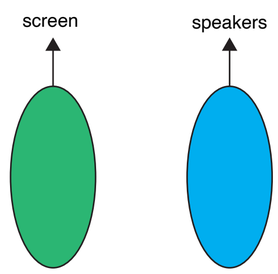
STEP 3
Handle remaining untyped nodes by finding connected subgraphs of untyped nodes:
a. If the subgraph depends on any typed node, assign the whole subgraph to that context
b. If the subgraph is “pure” i.e. has no typed dependencies and multiple contexts depend on it, duplicate the entire subgraph for each dependent type.





STEP 4
Now that we have duplicated some nodes, rebuild the edges with the new type information
a. Normal edge: both nodes are the same type, so this is regular data flow within that context
b. Reference edge: nodes of different types. This requires a cross-context data transfer





STEP 5
Extract subgraphs: divide the graph into normally-connected components (ignoring the reference edges). Each connected component is a chunk of computation that can be done in one context (i.e. a shader).
Then, build a meta-graph where nodes are these typed subgraphs, and edges of the meta-graph are the reference edges. Topological sort of this graph gives execution order. Each typed subgraph is a chunk of execution that can run independently in its own context. Reference edges are the bindings between these chunks of computation– they say “this shader code needs an audio sample”.