Uniformity analysis for WebGPU shading language

Top-level validation

- Build the call-graph
- For each leaf of the call-graph:
    - validate that function using the global environment
    - Add information about its uniformity requirements to the global environment
    - Remove it from the call-graph, updating the set of leaves

This is guaranteed to eventually validate every function, as we do not have recursion and so the call-graph is acyclic.

And during the validation of each function, we are also guaranteed to have the uniformity requirements of all callees.

Function validation

The validation of a function results in either an error (if the function is never valid), or in metadata that is used to check all calls to this function. This metadata is made of the following flags:
- CF_mustBeUniform if the function can only be called in uniform control-flow (like control_barrier())
- mustBeUniform_i for each parameter i, if the function can only be called by passing a uniform value at position i.
- canBeUniform_return, if the function is not-void and there is a context where its return value is uniform
- conditionallyUniform_i for each parameter i, if it must be a uniform value for the return value to be uniform.

The validation of a function goes in the following phases:

- generate a new CF id (CF stands for control-flow, an id is just an integer that corresponds to a new node in the graph of implications)
- generate two new special ids: mustBeUniform and cannotBeUniform
- generate a new id for each parameter
- generate a new environment (global environment + the right types/id for the parameters)
- validate the body of the function as a statement
- Starting from the special node mustBeUniform, do a DFS through the implications computed while validating the body
    - If this hits the cannotBeUniform node, return a validation error; the path between the two special nodes can be used for crafting the error message.
    - Otherwise the function is valid. Now we want to compute its annotations.
    - if CF was visited, then set CF_mustBeUniform for that function
    - for each parameter id that was visited, set the relevant mustBeUniform_i
- If the behaviors of the function include a “Return C”:
    - Note: behaviors are just a way in WHLSL validation to keep track of the ways that control-flow can exit a statement. All of that information should be trivially available in SPIRV when structured control-flow is used.
    - Note 2: C above is a set of constraints, i.e. a set of integers that correspond to nodes in the implication graph. It is used to keep track of which values/control-flow points must be uniform for the return value to be uniform.
    - do a DFS from the elements of C through the implications computed while validating the body, ignoring any node that was already visited in the previous DFS.
    - If this does *not* hit the cannotBeUniform node, set canBeUniform_return for that function

- And for each parameter id that was visited, set the relevant conditionallyUniform_i

Note: ignoring the already visited nodes in the second DFS is not required, but is likely to provide a nice speedup, not only in the DFS itself, but also by avoiding setting conditionallyUniform for parameters that already have mustBeUniform.

Validating a statement

I start by giving the pseudo-C++ code, followed by the formal rules.
The pseudo-code and the formal rules match closely, the main difference is that everything in the formal rules is immutable, whereas we can mutate the implications graph in-place in the pseudo-code.

```
// This computes a set of “Behaviors”, i.e. ways for control-flow to exit it (Return/Break/Continue/Nothing/InfiniteLoop), as well as a node that has to be uniform for the control-flow at the exit of the statement to be uniform.
// cf is the node corresponding to the uniformity of the control-flow at the entry of the statement.
std::tuple<Vector<Behavior>, int> validateStmt(const Environment& g, int cf, const Statement& s, Map<int, Set<int>>& implications, int& highestIDGenerated) {
  Vector<Behavior> behaviorsResult;
  int constraintResult
  switch(s.type) {
    case IfThenElse:
      auto constraint = validateExprRVal(g, cf, s.condition, implications, highestIDGenerated);
      int cf2 = ++highestIDGenerated;
      implications.insertWithUnion(cf2, { cf, constraint });
      auto (behaviors1, constraint1) = validateStmt(g, cf2, s.then, implications, highestIDGenerated);
      auto (behaviors2, constraint2) = validateStmt(g, cf2, s.else, implications, highestIDGenerated);
      constraintsResult = ++highestIDGenerated;
      implications.insertWithUnion(constraintResult, { constraint1, constraint2 });
      behaviorsResult = merge(behaviors1, behaviors2);
      break;
    case ForLoop: // assuming that the initializing expression got desugared away, so for(;e1;e2) s
      int cf2 = ++highestIDGenerated;
      auto constraint1 = validateExprRVal(g, cf2, s.condition, implications, highestIDGenerated);
      (void) validateExprRVal(g, cf2, s.increment, implications, highestIDGenerated);
      auto (behaviors, constraint3) = validateStmt(g, cf2, s.body, implications, highestIDGenerated);
      implications.insertWithUnion(cf2, {cf, constraint1, constraint3});
      constraintResult = cf2;
      behaviors.remove(Break);
      behaviors.remove(Continue);
      behaviors.insert(InfiniteLoop);
      behaviors.insert(Nothing);
      behaviorsResult = behaviors;
      break;
    case Block: // i.e. a sequence of statements between curly braces
      Environment env;
      if (s.isEmptyBlock) {
        return { {Nothing}, cf };
      for (auto& stmt : s.stmts) {
        if (stmt.isVariableDeclaration()) {
          auto x = stmt.variableName();
          auto tval = stmt.variableType();
          int newVariableID = ++highestCFGenerated;
```

```
                env = g.clone().insert(x, { LeftValue(tval, threadAddressSpace), newVariableID);
                auto constraint = validateExprRVal(env, cf, stmt.variableInitializer, implications, highestIDGenerated);
                implications.insertWithUnion(newVariableID, {cf, constraints});
            } else {
                auto (behaviors, constraint) = validateStmt(env, cf, stmt, implications, highestIDGenerated)
                if (stmt.isLastInBlock())
                    constraintResult = constraint;
                else {
                    behaviors.remove(Nothing);
                    cf = constraint;
                }
                behaviorsResult = union(behaviorsResult, behaviors);
            }
        }
        break;
    case Return: // return e;
        auto constraint = validateExprRVal(g, cf, stmt.expr, implications, highestIDGenerated);
        int result = ++highestIDGenerated;
        implications.insertWithUnion(result, { cf, constraint });
        constraintResult = cf;
        behaviorsResult = { Return(result) };
    case EffectfulExpr: // e;
        (void) validateExprRVal(g, cf, stmt.expr, implications, highestIDGenerated);
        constraintResult = cf;
        behaviorsResult = { Nothing };
    case Break: // break;
        constraintResult = cf;
        behaviorsResult = { Break };
  }
  if (behaviors.size() == 1) // This is responsible for reconvergence, WebSPIRV can do it directly without bothering
with behaviors.
    constraintResult = cf;
  return { behaviorsResult, constraintResult };
}
```

Note about reading the formal rules:
"G, CF |- e : t, C, I" reads “In environment G, with control-flow which corresponds to the id ‘CF’, the expression e has type t, has a result whose uniformity corresponds to the id ‘C’, and adds the edges in ‘I’ to the graph of implications”.
Similarly, “G, CF |- s : B, C, I” reads “In environment G, with control-flow which corresponds to the id ‘CF’, the statement s has the set of behaviors B, has control-flow that exits it with uniformity that corresponds to the id ‘C’, and adds the edges in ‘I’ to the graph of implications”.
Finally,

```
  a
  b
  c
  ——
  d
```

simply means that d is true if a, b, and c are all true.
So all of the formal rules below are simply a purely declarative equivalent of the imperative pseudo-code above, and are not required to understand the rest of this email.

G, CF |- e : bool, C, I
CF' = freshID()
G, CF' |- s1 : B1, C1, I1
G, CF' |- s2 : B2, C2, I2
C' = freshID()
I' = I U I1 U I2 U {CF' => {CF, C}, C' => {C1, C2}}

G, CF |- if (e) s1 else s2 : B1 U B2, C', I'

CF' = freshID()
G, CF' |- e1: bool, C1, I1
G, CF' |- e2: tval, C2, I2
G, CF' |- s: B, C3, I3
I = I1 U I2 U I3 U {CF' => {CF, C1, C3}}
B' = B \ {Break, Continue} U {Nothing, InfiniteLoop}

G, CF |- for (;e1;e2) s : B', CF', I

n > 0
G, CF |- s0 : B0, C0, I0
G, C0 |- {s1 .. sn} : B, C, I
I' = I0 U I

G, CF |- {s0 s1 .. sn} : B U (B0\{Nothing}), C, I'

n > 0
x_id = freshID()
G' = G [x -> (LVal(tval, thread), x_id)]
G', CF |- e : tval, C1, I1
G', CF |- {s1 .. sn} : B, C2, I2

G, CF |- {tval x = e; s1 .. sn} : B, C2, I1 U I2 U {x_id => {CF U C1}}

G, CF |- s : B, C, I

G, CF |- {s} : B, C, I

G, CF |- {} : {Nothing}, CF, {}

G, CF |- e : tval, C, I
result = freshID()

G, CF |- return e; : {Return(result)}, CF, I U {result => {CF, C} }

G, CF |- e: tval, C, I

G, CF |- e; : {Nothing}, CF, I

G, CF |- break; : {Break}, CF, {}

G, CF |- s: {b}, C, I

G, CF |- s: {b}, CF, I

Expressions

We treat expressions differently depending on whether they are in a left-value position or a right-value position. Note: for this purpose we do not treat abstract left-values differently from left-value, the existence of setters in WHLSL is orthogonal from the uniformity problem.

```
// Return the identifier matching whatever variable owns the storage for this left-value
int validateExprLVal(const Environment& g, int cf, const Expression& e, Map<int, Set<int>>& implications, int&
highestIDGenerated) {
    switch(e.type) {
        case VariableAccess: // x
            auto (LeftValue(tval, as), x_id) = g.get(e.variableName);
            return x_id;
        case ArrayDereference: // e1[e2]
            auto x_id = validateExprLVal(g, cf, e.base, implications, highestIDGenerated);
            auto constraint2 = validateExprRVal(g, cf, e.index, implications, highestIDGenerated);
            implications.insertWithUnion(x_id, { constraint2 });
            return x_id;
        case PointerDereference: // * e
            (void) validateExprRVal(g, cf, e.expr, implications, highestIDGenerated);
            return cannotBeUniform;
    }
}
```

Note: this whole algorithm only infers the uniformity of local variables and arguments. It intrinsically treats anything that comes from a pointer dereference as non-uniform, and anything that escapes as also being non-uniform. I don't see a way to do better without introducing alias analysis, and it feels massively overkill to specify.

x -> (LVal(tval, as), x_id) in G

G, CF |- x : LVal(tval, as), x_id, {}

G, CF |- e1 : LVal(tval[], as), x_id, I1
G, CF |- e2: uint, C2, I2

G, CF |- e1[e2] : LVal(tval, as), x_id, I1 U I2 U {x_id => {C2} }

G, CF |- e : Ptr(tval, as), C, I

G, CF |- * e : LVal(tval, as), cannotBeUniform, I

```
// Returns a node ID. That node is what must be uniform for the expression's result to be uniform
int validateExprRVal(const Environment& g, int cf, const Expression& e, Map<int, Set<int>>& implications, int&
highestIDGenerated) {
```

```
    switch(expression.type) {
        case BooleanOr: // e1 || e2
            // This is not symmetric, because || can short-circuit.
            // So for example (control_barrier(), true) || (tid == 42) can be valid whereas (tid == 42) || (control_barrier(),
true) never is.
            auto constraint1 = validateExprRVal(g, cf, e.lhs, implications, highestIDGenerated);
            int cf2 = ++highestIDGenerated;
            implications.insertWithUnion(cf2, {cf, constraint1});
            auto constraint2 = validateExprRVal(g, cf2, e.rhs, implications, highestIDGenerated);
            return constraint2;
        case Assignment: // e1 = e2
            int x_id = validateExprLVal(g, cf, e.lhs, implications, highestIDGenerated);
            auto constraint2 = validateExprRVal(g, cf, e.rhs, implications, highestIDGenerated);
            implications.insertWithUnion(x_id, { constraint2 });
            return constraint2;
        case VariableAccess: // x
            auto (LeftValue(tval, _), x_id) = g.get(e.variableName);
            int result = ++highestIDGenerated;
            implications.insertWithUnion(result, {x_id, cf});
            return result;
        case ArrayDereference: // e1[e2]
            auto constraint1 = validateExprRVal(g, cf, e.base, implications, highestIDGenerated);
            auto constraint2 = validateExprRVal(g, cf, e.index, implications, highestIDGenerated);
            int result = ++highestIDGenerated;
            implications.insertWithUnion(result, {constraint1, constraint2});
            return result;
        case AddressTaking: // & e
            int x_id = validateExprLVal(g, cf, e.expr, implications, highestIDGenerated);
            implications.insertWithUnion(x_id, { cannotBeUniform });
            return cf;
        case PointerDereference: // *e
            (void) validateExprRVal(g, cf, e.expr, implications, highestIDGenerated);
            return cannotBeUniform;
        case Literal: // 42 or 3.14
            return cf;
        case FunctionCall: // f(e_1, e_2, …, e_n)
            auto functionInfo = g.getFunction(e.function);
            if (functionInfo.mustCFBeUniform())
                implications.insertWithUnion(mustBeUniform, { cf });
            int result = ++highestIDGenerated;
            implications.insertWithUnion(result, functionInfo.canReturnBeUniform() ? { cf } : { cannotBeUniform });
            for (int i = 0; i < e.argumentsCount; ++i) {
                auto argConstraint = validateExprRVal(g, cf, e.arguments[i], implications, highestIDGenerated);
                if (functionInfo.parameterUniformity[i] == MustBeUniform)
                    implications.insertWithUnion(mustBeUniform, {argConstraint});
                else if (functionInfo.canReturnBeUniform() && functionInfo.parameterUniformity[i] == ConditionallyUniform)
                    implications.insertWithUnion(result, {argConstraint});
            }
            return result;
    }
```

}

G, CF |- e1 : bool, C1, I1
CF' = freshID()
G, CF' |- e2 : bool, C2, I2
I = I1 U I2 U {CF' => {CF U C1} }
———————————————
G, CF |- e1 || e2 : bool, C2, I

G, CF |- e1 : LVal(tval), x_id, I1
G, CF |- e2 : tval, C2, I2
I = I1 U I2 U {x_id => C2}
———————————————
G, CF |- e1 = e2 : tval, C2, I

x -> (LVal(tval, as), x_id) in G
result = freshID()
———————————————
G, CF |- x : tval, result, {result => {x_id, CF} }

G, CF |- e1 : tval[], C1, I1
G, CF |- e2: uint, C2, I2
result = freshID()
———————————————
G, CF |- e1[e2] : tval, result, I1 U I2 U {result => {C1, C2} }

G, CF |- e : LVal(tval, as), x_id, I
———————————————
G, CF |- & e : Ptr(tval, as), CF, I U {x_id => {cannotBeUniform}}

G, CF |- e : Ptr(tval, as), C, I
———————————————
G, CF |- * e : tval, cannotBeUniform, I

———————————————
G, CF |- Literal : tval(Literal), CF, {}

f : CF_mustBeUniform (tval_1 mustBeUniform_1 conditionallyUniform_1, …, tval_n mustBeUniform_n conditionallyUniform_n) -> tval_return canBeUniform_return in G
G, CF |- e1: tval_1, C1, I1
…
G, CF |- en: tval_n, Cn, In
if canBeUniform_return
  C = {CF} U {C_i for each i such that conditionallyUniform_i}
else
  C = {cannotBeUniform}
if CF_mustBeUniform
  I = {mustBeUniform => {CF} }
else
  I = {}

result = ++highestIDGenerated();
I = I U I1 U … U In U {mustBeUniform => { C_i for each i such that mustBeUniform_i}, result => C }
———

G, CF |- f (e1, …, en) : tval_return, result, I

Note: this rule is not very nice to read, and not at all idiomatic. But doing it the idiomatic way without the if/then/else would make it 4 rules, and probably be even less readable.

Complexity

The DFS are linear-time in the number of edges in the implications graph.
All rules add at most three edges, except for function calls that may add up to n+1, where n is the number of arguments.
So the number of edges in the implications graph is linear in the program size, and so is the time taken by the DFS.

The only potentially costly step in the rules themselves is the adding of new edges to the graph.
If the set of edges leaving a node is represented as a balanced tree, giving us O(log degreeOfNode) per rule, then we have a total time bound of O(n*log n).
If it is instead represented by some kind of HashSet and we assume that insertion is O(1) (true as long as we have less than 2^64 buckets in the hash table..), then we have a total time bound of O(n).
If this union operation proves to be a bottleneck, it can easily be optimized, by using small vectors and linear search in the (common) case where there are few elements in the set, and a HashSet otherwise.