WHLSL Documentation

Robin Morisset, Filip Pizlo, Myles C. Maxfield

Jul 29, 2019

CONTENTS

CHAPTER

ONE

GENERAL

This section is non-normative.

A shader passes through a series of compilation steps, eventually ending up represented as machine code for the specific device the user is running. Any intermediate forms the source may take throughout this transformation are beyond the scope of this specification.

. Note:: The WebGPU Shading Language is designed to target other high-level shading languages as compilation targets.

WHLSL shaders are used with the WebGPU API. Specific WebGPU API entry points to compile and manipulate shaders are specified in the WebGPU specification, not this document. This document describes which programs are well-formed; the exact form of error reporting is not discussed in this specification.

WHLSL does not support extensions or optional behavior.

Terms in this document such as *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in normative parts of this document are to be interpreted as described in RFC 2119.

CHAPTER

TWO

BASICS

The WebGPU Shading Language is designed to be as portable as possible; therefore, implementations must reject any shader which does not strictly adhere to this specification. Optimizations must not affect the validity of a program. Implementations must not support functionality beyond the mandated parts of this specification.

Note: The fact that optimizations must not affect the validity of a program means errors in dead code must cause a compilation failure. However, optimizations may, in general, be observable, such as fusing a multiply followed by an add into a single operation which has higher intermediate precision than the distinct operations. This means that the same WHLSL program run on different machines, browsers, or operating systems may might not produce the exact same results bit-for-bit.

A shader is a compilation unit which includes type definitions and function definitions.

WHLSL is used to describe different types of shaders:

1. Vertex shaders
2. Fragment shaders
3. Compute shaders

Each shader type represents software which may execute on a specialized processor. Different draw calls, different shader types, or different invocations of the same shader, may execute on independent processors.

A WHLSL string passes through the following stages of processing before it is executed:

1. Tokenization
2. Parsing
3. Validation

Once a WHLSL string passes all the validation checks, it is then available to be used in a draw call or dispatch call. A WHLSL string contains zero or more shaders, and each shader is of a specific shader type. Compute shaders must only be used for dispatch calls, and vertex and fragment shaders must only be used in draw calls.

CHAPTER

THREE

PIPELINES

WebGPU includes two pipelines: the graphics pipeline and the compute pipeline.

3.1 Graphics Pipeline

The WebGPU graphics pipeline includes five stages, two of which are programmable. The graphics pipeline is invoked by WebGPU draw calls.

3.1.1 Input Assembler

The first stage of the graphics pipeline is the input assembler. This stage is not programmable. This stage may do a collection of many things, including collecting primitives, determining which vertices are actually referenced in the draw call, extruding points/lines, expanding adjacency information into adjacent triangles, and more. As this stage is not programmable, the exact behavior of this stage is not observable. This stage is configured with the WebGPU API.

3.1.2 Vertex Shader

After the vertices relevant to a particular draw call have been determined, they are run through the programmable vertex shader. A vertex shader is responsible for mapping a single input vertex to a single output vertex. Multiple vertex shaders may run in parallel; two arbitrary vertices may be mapped through the vertex shader concurrently or sequentially.

An input vertex may be any assortment of information, arranged into four forms:

1. Stage-in data. Each invocation of the vertex shader is associated with data from a particular index in a GPU buffer. The WebGPU API is responsible for describing the association between which invocation receives which index in which buffer. Stage-in data must only be of scalar, vector, or matrix type.
2. Resources. All invocations of the vertex shader may have access to one or more resources. All invocations share the same resource; therefore, data races may occur between read-write resources. Resources may be of type buffer, texture, or sampler.
3. Built-ins. Some information can be made available to the shader automatically (such as the vertex ID or the instance ID).
4. Specialization constants. These are scalar variables for which the WebGPU API specifies a value before the shader may be used.

Because vertex shaders may have write-access to resources, they are not “pure” in the functional sense. The order of execution of multiple invocations of the vertex shader may be observable. Execution of multiple invocations of the vertex shader may be multiplexed across multiple processing units at the entire shader level or the instruction level (or any level in between). Therefore, when using simple loads and stores, load tearing may occur, or any such artifacts. WHLSL authors must take care to create shaders which are portable. See below for advice on how to accomplish this.

Note: Specific GPU instructions are not within scope of this document; therefore, races may lead to surprising and interesting results.

An output vertex may be any assortment of information, arranged into two forms:

1. A position, in Clip Coordinates, represented by a float4. When a vertex shader emits a position in Clip Coordinates, the WebGPU runtime will divide this position by its “w” component, resulting in a position in Normalized Device Coordinates. In Normalized Device Coordinates, the “x” component represents horizontal distance across the screen (or other output medium), where -1 represents the left edge and 1 represents the right edge. Similarly, the “y” component represents the vertical distance between -1 and 1, and the “z” component represents depth, where -1 represents the minimum depth and 1 represents the maximum depth.
2. Other information, represented by a collection of scalar values, vector values, and matrix values.

3.1.3 Rasterizer

Once the relevant vertex shaders have been run, their positions have been emitted, and those positions have been transformed into Normalized Device Coordinates, the rasterizer now interpolates the values of the other information in the output vertex. For a particular primitive, the rasterizer iterates over all fragments on the interior of the primitive, and computes the barycentric coordinate of that particular fragment with respect to the vertices of the primitive. It then computes a weighted average of the other vertex information using the barycentric coordinates as weights. This stage is not programmable.

3.1.4 Fragment Shader

After the vertex output information has been interpolated across the face of each vertex, one invocation of the fragment shader runs for each of these sets of interpolated values. A fragment shader is responsible for mapping the interpolated result of the vertex shader into a single output fragment (which is usually a color in the framebuffer, but may be other information such as geometry in a G-buffer or lighting accumulation in a lighting buffer).

Similar to a vertex shader, a fragment shader input may be any assortment of information, arranged into four forms:

1. Interpolated output from the vertex shader. These variables are matched to vertex shader variables using the routine described below.
2. Resources. All invocations of the fragment shader may have access to one or more resources. All invocations share the same resource; therefore, data races may occur between read-write resources. Resources may be of type buffer, texture, or sampler.
3. Built-ins. Some information can be made available to the shader automatically (such as the sample ID or the primitive ID).
4. Specialization constants. These are scalar variables for which the WebGPU API specifies a value before the shader may be used.

Because vertex shaders may have write-access to resources, they are not “pure” in the functional sense. The order of execution of multiple invocations of the vertex shader may be observable. Execution of multiple invocations of the vertex shader may be multiplexed across multiple processing units at the entire shader level or the instruction level (or any level in between). Therefore, WHLSL authors must take care to create shaders which are portable. See below for advice on how to accomplish this.

Note: Specific GPU instructions are not within scope of this document; therefore, races may lead to surprising and interesting results.

Because each invocation of the fragment shader is associated with a particuluar fragment with respect to the geometry of the primitive being drawn, the fragment shader can output into a particular region into zero or more attachments of the framebuffer. The fragment shader does not choose which region of the framebuffer its results get outputted into; instead, the fragment shader only gets to choose which values get outputted into that region.

The destination region of the framebuffer may be a pixel on the screen (if the framebuffer is attached to a canvas element). It may also be a texel in a texture, or a particular sample or set of samples in a multisampled texture.

The type of this output data must match the type of the framebuffer attachments being written into. See below for a rigorous definition of "match."

3.1.5 Output Merger

Once the fragment shader outputs a particular value for a fragment, that value must be merged with whatever value the fragment already happens to hold. For example, the new color may be linearly blended with the existing framebuffer contents (possibly using the "w" channel of the new color to determine the weights).

The output merger for a particular fragment is guaranteed to occur in API submission order for all primitives that overlap that particular fragment.

Note: This is in contrast to the fragment shader stage of the pipeline, which has no such guarantee.

3.2 Compute pipeline

The compute pipeline only has a single stage, and is invoked by WebGPU dispatch calls. The compute pipeline and the graphics pipeline are thus mutually exclusive; a single WebGPU call will invoke either the graphics pipeline or the compute pipeline, but not both.

Compute shader invocations are arranged into a two-level hierarchy: invocations are grouped into blocks, and blocks are grouped into a single grid. Multiple invocations that share a block share threadgroup variables for both reading and writing.

The WebGPU API describes how many invocations of the compute shader to invoke, as well as how big the blocks should be within the grid.

The input to a compute shader may be any assortment of information, arranged into three forms:

1. Resources. All invocations of the compute shader may have access to one or more resources. All invocations share the same resource; therefore, data races may occur between read-write resources. Resources may be of type buffer, texture, or sampler.
2. Built-ins. Some information can be made available to the shader automatically (such as the invocation ID within the block or the block ID within the grid).
3. Specialization constants. These are scalar variables for which the WebGPU API specifies a value before the shader may be used.

3.3 Entry Points

All functions in WHLSL are either "entry points" or "non-entry points." An entry point is a function that may be associated with a particular programmable stage in a pipeline. Entry points may call non-entry points, non-entry points may call non-entry points, but entry points may not be called by any WHLSL function. When execution of a particular shader stage begins, the entry point associated with that shader stage begins, and when that entry point returns, the associated shader stage ends.

Exactly one WHLSL shader occupies one stage in the WebGPU pipeline at a time. Two shaders of the same shader type must not be used together in the same draw call or dispatch call. Every stage of the appropriate WebGPU pipeline must be occupied by a shader in order to execute a draw call or dispatch call.

All entry points must begin with the keyword "vertex", "fragment", or "compute", and the keyword describes which pipeline stage that shader is appropriate for. An entry point is only valid for one type of shader stage.

Built-ins are identified by name. WHLSL does not include annotations for identifying built-ins. If the return of a shader should be assigned to a built-in, the author should create a struct with a variable named according to to the built-in, and the shader should return that struct.

Vertex and fragment entry points must transitively never refer to the `threadgroup` memory space.

3.3.1 Arguments and Return Types

Arguments return types of an entry point are more restricted than arguments to an arbitrary WHLSL function. They are flattened through structs - that is, each member of any struct appearing in an argument to an entry point or return type is considered independently, recursively. Arguments to entry points are not distinguished by position or order.

Multiple members with the same name may appear inside the flattened collection of arguments. However, if multiple members with the same name appear, the entire variable (type, qualifiers, etc.) must be identical. Otherwise, the entire program is in error.

The items of the flattened structs can be partitioned into a number of buckets:

1. Built-in variables. These declaractions use the appropriate built-in semantic from the list below, and must use the appropriate type for that semantic.
2. Resources. These must be either the opaque texture types, opaque sampler types, or slices. Slices must only hold scalars, vectors, matrices, or structs containing any of these types. Nested structs are allowed. The packing rules for data inside slices are described below. All resources must be in the `device` or `constant` memory space, and use the appropriate semantic as described below.
3. Stage-in/out variables. These are variables of scalar, vector, or matrix type. Stage-in variables in a vertex shader must use the semantic ‘‘ : attribute(n)‘‘ where n is a nonnegative integer. Stage-out variables in a vertex shader and stage-in variables in a fragment shader must also use the semantic ‘‘ : attribute(n)‘‘. Stage-out variables in a vertex shader are matched with stage-in variables in a fragment shader by semantic. After these stage-in/stage-out varaibles match, their qualified type must also match. After discovering all these matches, any other left-over variables are simply zero-filled.
4. Specialization constants. These are scalar variables which must be specified by the WebGPU API before the shader is allowed to execute. These variables must use the `specialized` semantic.

Vertex shaders accept all four buckets as input, and allow only built-in variables and stage-out variables as output. Fragment shaders accept all four buckets as input, and allow only built-in variables as output. Compute shaders only accept built-in variables and resources, and do not allow any output.

If an entry-point returns a single built-in or stage-out variable, the semantic for that variable must be placed between the function signature and the function's opening `{` character.

Vertex shader stage-out variables and fragment-shader stage-in variables may be qualified with any of the following qualifiers: `nointerpolation`, `noperspective`, `centroid`, or `sample`. `nointerpolation` and `noperspective` must not both be specified on the same variable. `centroid` and `sample` must not both be specified on the same variable. If other variables are qualified with these qualifiers, the qualifiers are ignored.

`nointerpolation` configures the rasterizer to not interpolate the value of this variable across the geometry. `noperspective` configures the rasterize to not use perspective-correct interpolation, and instead use simple linear interpolation. `centroid` configures the rasterizer to use a position in the centroid of all the samples within the geometry, rather than the center of the pixel. `sample` configures the fragment shader to run multiple times per pixel, with the interpolation point at each individual sample.

The value used for variables qualified with the `nointerpolation` qualifier is the value produced by one vertex shader invocation per primitive, known as the "provoking vertex." When drawing points, the provoking vertex is the vertex associated with that point (since points only have a single vertex). When drawing lines, the provoking vertex is the initial vertex (rather than the final vertex). When drawing triangles, the provoking vertex is also the initial vertex. Strips and fans are not supported by WHLSL.

When not in the context of arguments or return values of entry points, semantics are ignored.

CHAPTER

FOUR

GRAMMAR

4.1 Lexical analysis

Shaders exist as a Unicode string, and therefore support all the code points Unicode supports.

WHLSL does not include any digraphs or trigraphs. WHLSL is case-sensitive. It does not include any escape sequences.

Note: WHLSL does not include a string type, so escape characters are not present in the language.

WHLSL does not include a preprocessor step.

Note: Because there is no processor step, tokens such as '#if' are generally considered parse errors.

Before parsing, the text of a WHLSL program is first turned into a list of tokens, removing comments and whitespace along the way. Tokens are built greedily, in other words each token is as long as possible. If the program cannot be transformed into a list of tokens by following these rules, the program is invalid and must be rejected.

A token can be either of:

- An integer literal
- A float literal
- Punctuation
- A keyword
- A normal identifier
- An operator name

4.1.1 Literals

An integer literal can either be decimal or hexadecimal, and either signed or unsigned, giving 4 possibilities.

- A signed decimal integer literal starts with an optional `-`, then a number without leading 0 or just the number 0.
- An unsigned decimal integer literal starts with a number without leading 0, or just the number 0, then `u`.
- A signed hexadecimal integer literal starts with an optional `-`, then the string `0x`, then a non-empty sequence of elements of [0-9a-fA-F] (non-case sensitive, leading 0s are allowed).

- An unsigned hexadecimal inter literal starts with the string `0x`, then a non-empty sequence of elements of [0-9a-fA-F] (non-case sensitive, leading 0s are allowed), and finally the character `u`.

Note: Leading 0s are allowed in hexadecimal integer literals, but not in decimal integer literals except for the 0, -0 and 0u.

A float literal is made of the following elements in sequence:

- an optional − character
- a sequence of 0 or more digits (in [0-9])
- a `.` character
- a sequence of 0 or more digits (in [0-9]). This sequence must instead have 1 or more elements, if the last sequence was empty.
- optionally a `f` character

In regexp form: ‘-‘? ([0-9]+ ‘.’ [0-9]* | [0-9]* ‘.’ [0-9]+) f?

4.1.2 Keywords and punctuation

The following strings are reserved keywords of the language:

Top level	struct typedef enum operator vertex fragment compute numthreads
Control flow	if else switch case default while do for break continue fallthrough return
Literals	null true false
Address space	constant device threadgroup thread
Qualifier	nointerpolation noperspective specialized centroid sample
‘Semantics’ qualifier	SV_InstanceID SV_VertexID PSIZE SV_Position SV_IsFrontFace SV_SampleIndex SV_InnerCoverage SV_Target SV_Depth SV_Coverage SV_DispatchThreadId SV_GroupID SV_GroupIndex SV_GroupThreadID attribute register specialized
Reserved for future extension	protocol auto const static restricted native space uniform

`null`, `true` and `false` are keywords, but they are considered literals in the grammar rules later.

Similarily, the following elements of punctuation are valid tokens:

Relational operators	`== != <= => < >`
Assignment operators	`= ++ -- += -= *= /= %= ^= &= |= >>= <<=`
Arithmetic operators	`+ - * / %`
Logic operators	`&& || & | ^ >> << ! ~`
Memory operators	`-> . & @`
Other	`? : ; , [ ] { } ( )`

4.1.3 Identifiers

An identifier is any sequence of characters or underscores, that does not start by a digit, that is not a single underscore (the single underscore is reserved for future extension), and that is not a reserved keyword.

4.1.4 Whitespace and comments

Any of the following characters are considered whitespace, and ignored after this phase: space, tabulation (`\t`), carriage return (`\r`), new line(`\n`).

WHLSL also allows two kinds of comments. These are treated like whitespace (i.e. ignored during parsing). The first kind is a line comment, that starts with the string `//` and continues until the next end of line character. The second kind is a multi-line comment, that starts with the string `/*` and ends as soon as the string `*/` is read.

Note: Multi-line comments cannot be nested, as the first `*/` closes the outermost `/*`

4.2 Parsing

In this section we will describe the grammar of WHLSL programs, using the usual BNF metalanguage (https://en.wikipedia.org/wiki/Backus\T1\textendash{}Naur_form). We use names starting with an upper case letter to refer to lexical tokens defined in the previous section, and names starting with a lower case letter to refer to non-terminals. These are linked (at least in the HTML version of this document). We use non-bold text surrounded by quotes for text terminals (keywords, punctuation, etc..).

4.2.1 Top-level declarations

A valid compilation unit is made of a sequence of 0 or more top-level declarations.

```
topLevelDecl  ::=   ";" | typedef | structDef | enumDef | funcDef
```

Todo: We may want to also allow variable declarations at the top-level if it can easily be supported by all of our targets. (Myles: We can emulate it an all the targets, but the targets themselves only allow constant variables at global scope. We should follow suit.) https://github.com/gpuweb/WHLSL/issues/310

```
typedef  ::=   "typedef" Identifier "=" type ";"
```

```
structDef      ::=   "struct" Identifier "{" structElement* "}"
structElement  ::=   type Identifier (":" semantic)? ";"
```

```
enumDef      ::=   "enum" Identifier (":" type)? "{" enumElement ("," enumElement)* "}"
enumElement  ::=   Identifier ("=" constexpr)?
```

```
funcDef              ::=   funcDecl "{" stmt* "}"
funcDecl             ::=   (entryPointDecl | normalFuncDecl | castOperatorDecl) (":" semant
entryPointDecl       ::=   ("vertex" | "fragment" | "[" numthreadsSemantic "]" "compute") ty
numthreadsSemantic   ::=   "numthreads" "(" IntLiteral "," IntLiteral "," IntLiteral ")"
normalFuncDecl       ::=   type (Identifier | operatorName) parameters
```

```
castOperatorDecl    ::=   "operator" type parameters
parameters          ::=   "(" ")" | "(" parameter ("," parameter)* ")"
parameter           ::=   type Identifier (":" semantic)?
```

Note: the return type is put after the "operator" keyword when declaring a cast operator, mostly because it is also the name of the created function.

```
operatorName  ::=   "operator" (">>" | "<<" | "+" | "-" | "*" | "/" | "%" | "&&" | "||" | '
```

Note: We call `operator.x` a getter for x, `operator.x=` a setter for x, and `operator&.x` an address taker for x.

```
semantic                          ::=   builtInSemantic | stageInOutSemantic | resourceSemant
builtInSemantic                   ::=   "SV_InstanceID" | "SV_VertexID" | "PSIZE" | "SV_Posit
stageInOutSemantic                ::=   "attribute" "(" IntLiteral ")"
resourceSemantic                  ::=   "register" "(" Identifier ")" | "register" "(" Identi
specializationConstantSemantic    ::=   "specialized"
```

4.2.2 Statements

```
stmt             ::=   "{" (stmt | variableDecls ";")* "}"
                       | compoundStmt
                       | terminatorStmt ";"
                       | maybeEffectfulExpr ";"
compoundStmt     ::=   ifStmt | ifElseStmt | whileStmt | doWhileStmt | forStmt | switchStmt
terminatorStmt   ::=   "break" | "continue" | "fallthrough" | "return" expr?
```

Note: The `fallthrough` statement is used at the end of switch cases to fallthrough to the next case. It is not valid to have it anywhere else, or to have a switch case where control-flow reaches the end without a fallthrough (see section *Typing statements*).

```
ifStmt       ::=   "if" "(" expr ")" stmt
ifElseStmt   ::=   "if" "(" expr ")" stmt "else" stmt
```

The first of these two productions is merely syntactic sugar for the second:

$$\textbf{if}(e)\, s \rightsquigarrow \textbf{if}(e)\, s\, \textbf{else}\, \{\}$$

```
whileStmt     ::=   "while" "(" expr ")" stmt
forStmt       ::=   "for" "(" (maybeEffectfulExpr | variableDecls) ";" expr? ";" expr? ")" 
doWhileStmt   ::=   "do" stmt "while" "(" expr ")" ";"
```

Similarly, we desugar all while loops into do while loops.

$$\textbf{while}(e)\, s \rightsquigarrow \textbf{if}(e)\textbf{do}\, s\, \textbf{while}(e)$$

We also partly desugar for loops:

1. If the second element of the for is empty we replace it by "true".
2. If the third element of the for is empty we replace it by "null". (any effect-free expression would work as well).
3. If the first element of the for is not empty, we hoist it out of the loop, into a newly created block that includes the loop:

$$\textbf{for}(X_{pre}; e_{cond}; e_{iter})\, s \rightsquigarrow \{X_{pre}; \textbf{for}(; e_{cond}; e_{iter})\, s\}$$

```
switchStmt  ::=  "switch" "(" expr ")" "{" switchCase* "}"
switchCase  ::=  ("case" constexpr | "default") ":" stmt*
```

```
variableDecls  ::=  type variableDecl ("," variableDecl)*
variableDecl   ::=  Identifier ("=" ternaryConditional)?
```

Complex variable declarations are also mere syntactic sugar. Several variable declarations separated by commas are the same as separating them with semicolons and repeating the type for each one. This transformation can always be done because variable declarations are only allowed inside blocks (and for loops, but these get desugared into a block, see above).

4.2.3 Types

```
type                       ::=  addressSpace Identifier typeArguments typeSuffixAbbreviate
                                | Identifier typeArguments typeSuffixNonAbbreviated*
addressSpace               ::=  "constant" | "device" | "threadgroup" | "thread"
typeSuffixAbbreviated      ::=  "*" | "[" "]" | "[" IntLiteral "]"
typeSuffixNonAbbreviated   ::=  "*" addressSpace | "[" "]" addressSpace | "[" IntLiteral "
```

Putting the address space before the identifier is just syntactic sugar for having that same address space applied to all type suffixes. `thread int *[]*[42]` is for example the same as `int *thread []thread *thread [42]`.

```
typeArguments  ::=  "<" (typeArgument ",")* addressSpace? Identifier "<"
                    (typeArgument ("," typeArgument)*)? ">>"
                    | "<" (typeArgument ("," typeArgument)* ">"
                    | ("<" ">")?
typeArgument   ::=  constepxr | type
```

The first production rule for typeArguments is a way to say that >> can be parsed as two > closing delimiters, in the case of nested typeArguments.

4.2.4 Expressions

WHLSL accepts three different kinds of expressions, in different places in the grammar.

- `expr` is the most generic, and includes all expressions.
- `maybeEffectfulExpr` is used in places where a variable declaration would also be allowed. It forbids some expressions that are clearly effect-free, such as `x*y` or `x < y`. As the name indicates, it may be empty. In that case it is equivalent to "null" (any other effect-free expression would be fine, as the result of such an expression is always discarded).
- `constexpr` is limited to literals and the elements of an enum. It is used in switch cases, and in type arguments.

```
expr                 ::=   (expr ",")? ternaryConditional
ternaryConditional   ::=   exprLogicalOr "?" expr ":" ternaryConditional
                           | exprPrefix assignOperator ternaryConditional
                           | exprLogicalOr
assignOperator       ::=   "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" | "^=" | ">>
exprLogicalOr        ::=   (exprLogicalOr "||")? exprLogicalAnd
exprLogicalAnd       ::=   (exprLogicalAnd "&&")? exprBitwiseOr
exprBitwiseOr        ::=   (exprBitwiseOr "|")? exprBitwiseXor
exprBitwiseXor       ::=   (exprBitwiseXor "^")? exprBitwiseAnd
exprBitwiseAnd       ::=   (exprBitwiseAnd "&")? exprRelational
exprRelational       ::=   exprShift (relationalBinop exprShift)?
relationalBinop      ::=   "<" | ">" | "<=" | ">=" | "==" | "!="
exprShift            ::=   (exprShift ("<<" | ">>"))? exprAdd
exprAdd              ::=   (exprMult ("*" | "/" | "%"))? exprPrefix
exprPrefix           ::=   prefixOp exprPrefix | exprSuffix
prefixOp             ::=   "++" | "--" | "+" | "-" | "~" | "!" | "*" | "&" | "@"
exprSuffix           ::=   callExpression limitedSuffixOp*
                           | term (limitedSuffixOp | "++" | "--")*
limitedSuffixOp      ::=   "." Identifier | "->" Identifier | "[" expr "]"
callExpression       ::=   Identifier "(" (ternaryConditional ("," ternaryConditional)*)? ")
term                 ::=   Literal | Identifier | "(" expr ")"
```

WHLSL matches the precedence and associativity of operators from C++, with one exception: relational operators are non-associative, so that they cannot be chained. Chaining them has sufficiently surprising results that it is not a clear reduction in usability, and it should make it a lot easier to extend the syntax in the future to accept generics.

The prefix form of increment and decrement (`++e` and `--e`) are syntactic sugar for `e += 1` and `e -= 1`.

`x -> y` is purely syntactic sugar for `(*x).y`, so we will ignore the `->` operator in the rest of this specification.

```
maybeEffectfulExpr   ::=   (effAssignment ("," effAssignment)*)?
effAssignment        ::=   exprPrefix assignOperator expr | effPrefix
effPrefix            ::=   ("++" | "--") exprPrefix | effSuffix
effSuffix            ::=   exprSuffix ("++" | "--") | callExpression | "(" expr ")"
```

The structure of maybeEffectfulExpr roughly match the structure of normal expressions, just with normally effect-free operators left off.

If the programmer still wants to use them in such a position (for example due to having overloaded an operator with an effectful operation), it can be done just by wrapping the expression in parentheses (see the last alternative for effSuffix).

```
constexpr  ::=   Literal | Identifier "." Identifier
```

CHAPTER

FIVE

VALIDATION

In this section we describe how to determine if a program is valid or not. If a program is invalid, a compliant implementation must reject it with an appropriate error message, and not attempt to execute it. If a program is valid, we describe its semantics later in this document.

Validation includes all of typing. If a program is valid, it is also annotated with typing information used by the execution semantics later (for example, accesses to fixed-size arrays are annotated with the size for the bounds-check).

The validation rules are presented in several steps:

- First we explain how the typing environment is built from the top-level declarations (*Building the global typing environment*)
- Then we provide global validation rules, including checking the absence of recursion (*Other validation steps*)
- Finally we provide the typing rules (*Typing of functions*)

5.1 Building the global typing environment

In this first step all top-level declarations are gathered into a global environment. More precisely they are gathered in three different mappings:

- A mapping from identifiers to types (typedefs, enums and structs)
- A mapping from identifiers to declarations of global (constant) variables
- A mapping from identifiers to sets of function declarations.

A type for the purpose of this mapping is either an enum characterized by a set of values, or it is a typedef characterized by its equivalent type, or it is a struct characterized by the types of its elements. A variable declaration for the purpose of this mapping is characterized by its type. A function declaration for the purpose of this mapping is characterised by a tuple of the return type, the number and types of the parameters, and the body of the function.

This environment is initialized with the types and function declarations from the standard library, see *Standard library*.

For each top-level declaration:

1. If it is a variable declaration
   (a) If there is already a variable of the same name in the environment, the program is invalid
   (b) Add it to the mapping with a type Left-value of its declared type in the Constant address space (see *Typing expressions* for details on types of values)
2. If it is a typedef
   (a) If there is already a type of the same name in the environment, the program is invalid
   (b) Add it to the mapping, as a new type, associated to its definition

3. If it is a structure
   - (a) If there is already a type of the same name in the environment, the program is invalid
   - (b) If two or more fields of the struct have the same name, the program is invalid
   - (c) Add the struct to the environment as a new type.
   - (d) **For each field of the struct, add to the environment a mapping from the name `operator&.field=` (where `field`**
     whose return type is a pointer to the type of the field, and whose argument type is a pointer to the struct itself. There is one such function declaration for each address space, that address space is used both by the pointer argument and by the return type
4. If it is an enum
   - (a) If there is already a type of the same name in the environment, the program is invalid
   - (b) If the enum has an explicit base type, and it is not one of `uint` or `int` then the program is invalid
   - (c) If the enum does not have an explicit base type, its base type is `int`
   - (d) A value is associated to each element of the enum, by iterating over them in source order:
     - i. If it has an explicit value, then this is its value
     - ii. Else if it is the first element of the enum, its value is 0
     - iii. Else its value is the value of the precedent element increased by one.
   - (e) If no element of the enum has the value 0, the program is invalid
   - (f) If two or more element of the enum have the same value, the program is invalid
   - (g) If one or more element of the enum have a value that is not representable in the base type of the enum, the program is invalid
   - (h) Add the enum to the environment as a new type, associated with the set of the values of its elements
   - (i) For each element of the enum, add a mapping to the variables mapping, from `EnumName.ElementName` (with `EnumName` and `ElementName` replaced) to the enum type
5. If it is a function declaration
   - (a) If the name of the function is `operator.field` for some name `field`
     - i. It must have a single argument
     - ii. That argument must not be a pointer, array reference or array
   - (b) Else if the name of the function is `operator.field=` for some name `field`
     - i. It must have exactly two arguments
     - ii. Its first argument must not be a pointer, array reference or array
   - (c) Else if the name of the function is `operator&.field` for some name `field`
     - i. It must have exactly one argument
     - ii. Its return type must be a pointer type
     - iii. Its argument must be a pointer type
     - iv. Both its return type and its argument type must be in the same address space
   - (d) Else if the name of the function is `operator[]`
     - i. It must have exactly two argument
     - ii. Its first argument must not be a pointer, array reference, or array.

        iii. Its second argument must be one of `uint` or `int`

   (e) Else if the name of the function is `operator[]=`

        i. It must have exactly three arguments

        ii. Its first argument must not be a pointer, array reference, or array

        iii. Its second argument must be one of `uint` or `int`

   (f) Else if the name of the function is `operator&[]`

        i. It must have exactly two arguments

        ii. Its return type must be a pointer type

        iii. Its first argument must be a pointer type

        iv. The type pointed at by this pointer cannot be a pointer, array reference, or array.

        v. Both its return type and its first argument type must be in the same address space

        vi. Its second argument must be one of `uint` or `int`

   (g) Else if the name of the function is `operator++` or `operator--`

        i. It must have exactly one argument

        ii. Its argument type and its return type must be the same

   (h) Else if the name of the function is `operator+` or `operator-`, it must have one or two arguments

   (i) Else if the name of the function is `operator*`, `operator/`, `operator%`, `operator&`, `operator|`, `operator^`, `operator<<` or `operator>>`, it must have exactly two arguments

   (j) Else if the name of the function is `operator~`, it must have exactly one argument

   (k) Else if the name of the function is `operator==`, `operator!=`, `operator<`, `operator>`, `operator<=` or `operator>=`

        i. It must have exactly two arguments

        ii. Its return type must be bool

   (l) If the environment already has a mapping from that function name to a set of declarations, add this declaration to that set

   (m) Otherwise add a new mapping from that function name to a singleton set containing that declaration

5.2 Other validation steps

We list here these validation steps that don't cleanly fit in either the building of the global typing environment, or the typing of each function.

5.2.1 Void type

The void type is a special type that can only appear as the return type of functions. It must not be part of a composite type (i.e. there is no pointer to void, no array reference to void, no array of void) It must not be the type of a variable (either at the top-level or in a function), the type of a field of a struct, the type of a function parameter, or the definition of a typedef.

5.2.2 Validating types

Every type name that appears in the program must be defined (i.e. have a mapping in the environment).

5.2.3 Resolving typedefs

We define a relation "depends on", as the smallest relation such that:

- A typedef that is defined as equal to a structure or another typedef "depends on" this structure or typedef.
- A structure "depends on" a typedef or structure if it has a member with the same name.

If this relation is cyclic, then the program is invalid.

Then each typedef must be resolved, meaning that each mention of it in the program and in the environment is replaced by its definition.

Note: This last step is guaranteed to terminate thanks to the acyclicity check before it.

5.2.4 Checking the coherence of operators and functions

For every declaration of a function with a name of the form `operator&.field` for some name `field` with argument type `thread T1*` and return type `thread T2*`:

1. Add a declaration of a function `operator.field=` for the same name `field`, with argument types `T1` and `T2`, and return type `T1`
2. Add a declaration of a function `operator.field` for the same name `field`, with argument type `T1` and return type `T2`

For every declaration of the function `operator&[]` with argument types `thread T1*` and `uint32`, and return type `thread T2*`:

1. Add a declaration of a function `operator[]=` with argument types `T1`, `uint32` and `T2`, and return type `T1`
2. Add a declaration of a function `operator[]` with argument types `T1` and `uint32`, and return type `T2`

For every function with a name of the form `operator.field=` for some name `field` which is defined:

1. There must be a function with the name `operator.field` (for the same name `field`) which is defined
2. For each declaration of the former with arguments type `(t1, t2)`, there must be a declaration of the latter with argument type `(t1)`, and return type `t2`

If a function with the name `operator[]=` is defined:

1. There must be a function with the name `operator[]` which is defined
2. For each declaration of the former with arguments type `(t1, t2, t3)`, there must be a declaration of the latter with argument type `(t1, t2)`, and return type `t3`

If there are two function declarations with the same names, number of parameters, and types of their parameters, then the program is invalid.

5.3 Typing of functions

Each function must be well-typed following the rules in this section.

To check that a function is well-typed:

1. Make a new copy of the global environment (built above)
2. For each parameter of the function, add a mapping to this typing environment, associating this parameter name to the corresponding type
3. Check that the function body is well-typed in this typing environment (treating it as a block of statement)
4. If the return type of the function is `void`, then the set of behaviours of the function body must be included in `{Nothing, Return Void}`
5. Else if the return type of the function is a type T, then the set of behaviours of the function body must be `{Return T}`

In this section we define the terms above, and in particular, what it means for a statement or an expression to be well-typed. More formally we define two mutually recursive judgments: "In typing environment Gamma, s is a well-typed statement whose set of behaviours is B" and "In typing environment Gamma, e is a well-typed expression whose type is Tau".

A type can either be:

- A left-value type with an associated right-value type and an address space
- An abstract left-value type with an associated right-value type
- A right-value type, which can be any of the following:
  - A basic type such as `bool` or `uint`
  - A structure type, defined by its name
  - An enum type, defined by its name
  - `void`
  - An array with an associated right-value type and a size (a number of elements). The size must be a positive integer that fits in 32 bits
  - A pointer with an associated right-value type and an address space
  - An array reference with an associated right-value type and an address space

Informally, a left-value type is anything whose address can be taken, whereas an abstract left-value type is anything that can be assigned to. Any value with a left-value type of non-constant address space can be given an abstract left-value type, and any value with an abstract left-value type (or left-value type even with a constant address space) can be given a right-value type, but the opposite to those is not true.

A behaviour is any of the following:

- Return of a right-value type
- Break
- Continue
- Fallthrough
- Nothing

We use these "behaviours" to check the effect of statements on the control flow.

5.3.1 Typing statements

To check an if-then-else statement:

1. Check that the condition is a well-typed expression of type bool
2. Check that the then and else branches are well-typed statements whose behaviours we will respectively call `B` and `B'`
3. Check that neither `B` nor `B'` contain a return of a pointer type, or of an array reference type
4. Then the if-then-else statement is well-typed, and its behaviours is the union of `B` and `B'`

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathbf{bool} \\ \Gamma \vdash s : B \\ \Gamma \vdash s' : B' \\ B'' = (B \cup B') \\ \mathbf{Return\,Ptr}\,(\tau^{val}, as) \notin B'' \\ \mathbf{Return\,Ref}\,(\tau^{val}, as) \notin B''\end{array}}{\Gamma \vdash \mathbf{if}\ dpid(e) s\ \mathbf{else}\ s' : B''}\ \text{IF}$$

To check a do-while or for statement:

1. Check that the condition is well-typed, of type `bool`
2. If it is a for statement, check that the expression that is executed at the end of each iteration is well-typed
3. Check that the body of the loop is a well-typed statement whose behaviours we will call `B`
4. Check that `B` does not contain a return of a pointer type, or of an array reference type
5. If Continue is in `B`, remove it
6. If Break is in `B`, remove it and add Nothing to `B`
7. Then the do-while statement is well-typed, and its behaviours is `B`

$$\frac{\begin{array}{l}\Gamma \vdash e : \textbf{bool} \\ \Gamma \vdash s : B \\ \textbf{Return Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return Ref}\,(\tau^{val}, as) \notin B \\ \textbf{Break} \in B\end{array}}{\Gamma \vdash \textbf{do}\ dpid\ s\ \textbf{while}\,(e);: (B \backslash \{\textbf{Break}, \textbf{Continue}\}) \cup \{\textbf{Nothing}\}} \ \text{DO\_WHILE\_BREAK}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \textbf{bool} \\ \Gamma \vdash s : B \\ \textbf{Return Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return Ref}\,(\tau^{val}, as) \notin B \\ \textbf{Break} \notin B\end{array}}{\Gamma \vdash \textbf{do}\ dpid\ s\ \textbf{while}\,(e);: B \backslash \{\textbf{Continue}\}} \ \text{DO\_WHILE\_NO\_BREAK}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \textbf{bool} \\ \Gamma \vdash e' : \tau \\ \Gamma \vdash s : B \\ \textbf{Return Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return Ref}\,(\tau^{val}, as) \notin B \\ \textbf{Break} \in B\end{array}}{\Gamma \vdash \textbf{for}\ dpid(; e; e')s : (B \backslash \{\textbf{Break}, \textbf{Continue}\}) \cup \{\textbf{Nothing}\}} \ \text{FOR\_BREAK}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \textbf{bool} \\ \Gamma \vdash e' : \tau \\ \Gamma \vdash s : B \\ \textbf{Return Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return Ref}\,(\tau^{val}, as) \notin B \\ \textbf{Break} \notin B\end{array}}{\Gamma \vdash \textbf{for}\ dpid(; e; e')s : B \backslash \{\textbf{Continue}\}} \ \text{FOR\_NO\_BREAK}$$

Note: We do not give rules for while loops, or for if-then statements without an else, because they are syntactic sugar that are eliminated during parsing (see *Parsing*).

To check a switch statement:

1. Check that the expression being switched on is well-typed
2. Check that its type is either an integer type (`int` or `uint`) or an enum type
3. Check that each value `v` in a `case v` in this switch is well-typed with the same type
4. Check that no two such cases have the same value
5. If there is a default, check that there is at least one value in that type which is not covered by the cases
6. Else check that for all values in that type, there is one case that covers it
7. Check that the body of each case (and default) is well-typed when treating them as blocks
8. Check that the behaviours of the last such body does not include Fallthrough
9. Make a set of behaviours that is the union of the behaviours of all of these bodies

10. Check that this set contains neither Nothing, nor a Return of a pointer type, nor a Return of an array reference type
11. If Fallthrough is in this set, remove it
12. If Break is in this set, remove it and add Nothing
13. Then the switch statement is well-typed, and its behaviours is this last set

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau^{val} \\ \Gamma \vdash \textbf{ isIntegerOrEnum}\,(\tau^{val}) \\ \Gamma \vdash sc_0 : \tau^{val} \land \,..\, \land \Gamma \vdash sc_n : \tau^{val} \\ \Gamma \vdash sc_0 \,..\, sc_n \,\textbf{fully covers}\, \tau^{val} \\ \Gamma \vdash sblock_0 : B_0 \land \,..\, \land \Gamma \vdash sblock_n : B_n \\ B = \bigcup B_0 \,..\, B_n \\ \textbf{Fallthrough} \notin B_n \\ \textbf{Nothing} \notin B \\ \textbf{Break} \in B \\ \textbf{Return}\,\textbf{Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return}\,\textbf{Ref}\,(\tau^{val}, as) \notin B\end{array}}{\Gamma \vdash \textbf{switch}\, dpid(e)\{sc_0 : sblock_0 \,..\, sc_n : sblock_n\} : (B \backslash \{\textbf{Break}, \textbf{Fallthrough}\}) \cup \{\textbf{Nothing}\}} \quad \text{SWITCH\_BREAK}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau^{val} \\ \Gamma \vdash \textbf{ isIntegerOrEnum}\,(\tau^{val}) \\ \Gamma \vdash sc_0 : \tau^{val} \land \,..\, \land \Gamma \vdash sc_n : \tau^{val} \\ \Gamma \vdash sc_0 \,..\, sc_n \,\textbf{fully covers}\, \tau^{val} \\ \Gamma \vdash sblock_0 : B_0 \land \,..\, \land \Gamma \vdash sblock_n : B_n \\ B = \bigcup B_0 \,..\, B_n \\ \textbf{Fallthrough} \notin B_n \\ \textbf{Nothing} \notin B \\ \textbf{Break} \notin B \\ \textbf{Return}\,\textbf{Ptr}\,(\tau^{val}, as) \notin B \\ \textbf{Return}\,\textbf{Ref}\,(\tau^{val}, as) \notin B\end{array}}{\Gamma \vdash \textbf{switch}\, dpid(e)\{sc_0 : sblock_0 \,..\, sc_n : sblock_n\} : B \backslash \{\textbf{Fallthrough}\}} \quad \text{SWITCH\_NO\_BREAK}$$

$$\frac{\Gamma \vdash rval : \tau^{val}}{\Gamma \vdash \textbf{case}\, rval : \tau^{val}} \quad \text{CASE}$$

$$\frac{}{\Gamma \vdash \textbf{default} : \tau^{val}} \quad \text{DEFAULT}$$

$$\frac{\Gamma \vdash \{\, s_0 \,..\, s_n \} : B}{\Gamma \vdash s_0 \,..\, s_n : B} \quad \text{SWITCH\_BLOCK}$$

The `break;`, `fallthrough;`, `continue;` and `return;` statements are always well-typed, and their behaviours are respectively {Break}, {Fallthrough}, {Continue} and {Return void}.

The statement `return e;` is well-typed if `e` is a well-typed expression with a right-value type T and its behaviours is then {Return T}.

$$\frac{}{\Gamma \vdash \mathbf{break}\,;:\{\mathbf{Break}\}}\quad\text{BREAK}$$

$$\frac{}{\Gamma \vdash \mathbf{continue}\,;:\{\mathbf{Continue}\}}\quad\text{CONTINUE}$$

$$\frac{}{\Gamma \vdash \mathbf{fallthrough}\,;:\{\mathbf{Fallthrough}\}}\quad\text{FALLTHROUGH}$$

$$\frac{}{\Gamma \vdash \mathbf{return}\,;:\{\mathbf{Return\,void}\}}\quad\text{RETURN\_VOID}$$

$$\frac{\Gamma \vdash e : \tau^{val}}{\Gamma \vdash \mathbf{return}\,e;:\{\mathbf{Return}\,\tau^{val}\}}\quad\text{RETURN}$$

To check a block:

1. If it is empty, it is well-typed and its behaviours is always {Nothing}
2. Else if it starts by a variable declaration:
   (a) Check that there is no other statement in that block is a variable declaration sharing the same name.
   (b) Check that the given address space is either `thread` or `threadgroup`
   (c) Make a new typing environment from the current one, in which the variable name is mapped to a left-value type of its given type and address-space
   (d) If there is no initializing expression, check that the type of this variable is neither a pointer type nor an array reference type.
   (e) Else if there is an initializing expression, check that it is well-typed in this new environment and that its type match the type of the variable
   (f) Check that the rest of the block, removing this first statement is well-typed in this new typing environment and has a set of behaviours B.
   (g) Then the block is well-typed and has the same set of behaviours B.
3. Else if this block contains a single statement, check that this statement is well-typed. If it is, then so is this block, and it has the same set of behaviours
4. Else
   (a) Check that this block's first statement is well-typed
   (b) Check that its set of behaviours B contains Nothing.
   (c) Remove Nothing from it.
   (d) Check that it does not contain Fallthrough
   (e) Check that the rest of the block, removing the first statement, is well-typed with a set of behaviours B'.
   (f) Then the whole block is well-typed, and its set of behaviour is the union of B and B'.

$$\frac{}{\Gamma \vdash \{\,\} : \{\textbf{Nothing}\}} \quad \text{EMPTY\_BLOCK}$$

$$\frac{\begin{array}{l}\tau^{val} \neq \textbf{Ptr}\,(\tau^{val'}, as)\\ \tau^{val} \neq \textbf{Ref}\,(\tau^{val'}, as)\\ \Gamma[x \mapsto \textbf{LVal}\,(\tau^{val}, \textbf{thread})] \vdash \{\, s_0 \mathinner{..} s_n \} : B\\ s_0 \neq \tau^{val'}\ x; \wedge \mathinner{..} \wedge s_n \neq \tau^{val'}\ x;\\ s_0 \neq \tau^{val'}\ x = e'; \wedge \mathinner{..} \wedge s_n \neq \tau^{val'}\ x = e';\end{array}}{\Gamma \vdash \{\, \tau^{val}\ x;\ s_0 \mathinner{..} s_n \} : B} \quad \text{VARIABLE\_DECL}$$

$$\frac{\begin{array}{l}\Gamma[x \mapsto \textbf{LVal}\,(\tau^{val}, \textbf{thread})] \vdash e : \tau^{val}\\ \Gamma[x \mapsto \textbf{LVal}\,(\tau^{val}, \textbf{thread})] \vdash \{\, s_0 \mathinner{..} s_n \} : B\\ s_0 \neq \tau^{val'}\ x; \wedge \mathinner{..} \wedge s_n \neq \tau^{val'}\ x;\\ s_0 \neq \tau^{val'}\ x = e'; \wedge \mathinner{..} \wedge s_n \neq \tau^{val'}\ x = e';\end{array}}{\Gamma \vdash \{\, \tau^{val}\ x = e;\ s_0 \mathinner{..} s_n \} : B} \quad \text{VARIABLE\_DECL\_INIT}$$

$$\frac{\Gamma \vdash s : B}{\Gamma \vdash \{\, s \} : B} \quad \text{TRIVIAL\_BLOCK}$$

$$\frac{\begin{array}{l}\Gamma \vdash s : B\\ \Gamma \vdash \{\, s_1 \mathinner{..} s_n \} : B'\\ n > 0\\ \textbf{Fallthrough} \notin B\\ \textbf{Nothing} \in B\end{array}}{\Gamma \vdash \{\, s\ s_1 \mathinner{..} s_n \} : (B \backslash \{\textbf{Nothing}\}) \cup B'} \quad \text{BLOCK}$$

Todo: Change the variable declaration ott rules to support threadgroup local variables https://github.com/gpuweb/WHLSL/issues/63

Finally a statement that consists of a single expression (followed by a semicolon) is well-typed if that expression is well-typed and its set of behaviours is then {Nothing}.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e; : \{\textbf{Nothing}\}} \quad \text{EXPR}$$

5.3.2 Typing expressions

Literals are always well-typed and of the following right-value types:

- `true` and `false` are always boolean.
- float literals are of any floating-point right-value type
- unsigned int literals (marked by an `u` at the end) are of any unsigned integer type large enough to contain them
- int literals are of any integer type large enough to contain them

`null` is always well-typed and its type can be any pointer or array reference type (depending on which is required for validation to succeed).

The type of an expression in parentheses, is the type of the expression in the parentheses

A comma expression is well-typed if both of its operands are well-typed. In that case, its type is the right-value type of its second operand.

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \quad \text{LITERAL\_TRUE}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \quad \text{LITERAL\_FALSE}$$

$$\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Ref}\,(\tau^{val}, as)} \quad \text{NULL\_LIT\_ARRAY\_REF}$$

$$\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Ptr}\,(\tau^{val}, as)} \quad \text{NULL\_LIT\_PTR}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \quad \text{PARENS}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \Gamma \vdash e' : \tau^{val'}\end{array}}{\Gamma \vdash e, e' : \tau^{val'}} \quad \text{COMMA}$$

To check that a boolean or, or a boolean and is well-typed, check that both of its operands are well-typed and of type bool.

To check that a ternary conditional is well-typed:

1. Check that its condition is well-typed and of type bool
2. Check that both of its branches are well-typed
3. Check that the types of its branches are both right-value types and the same
4. Check that this same type is neither a pointer type nor an array reference type.
5. Then the whole expression is well-typed, and of that type

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbf{bool} \\ \Gamma \vdash e' : \mathbf{bool}\end{array}}{\Gamma \vdash e \mid\mid e' : \mathbf{bool}} \quad \text{OR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbf{bool} \\ \Gamma \vdash e' : \mathbf{bool}\end{array}}{\Gamma \vdash e \,\&\&\, e' : \mathbf{bool}} \quad \text{AND}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_0 : \mathbf{bool} \\ \Gamma \vdash e_1 : \tau^{val} \\ \Gamma \vdash e_2 : \tau^{val} \\ \tau^{val} \neq \mathbf{Ptr}\,(\tau^{val'}, as) \\ \tau^{val} \neq \mathbf{Ref}\,(\tau^{val'}, as)\end{array}}{\Gamma \vdash e_0 \;?\; e_1 : e_2 : \tau^{val}} \quad \text{TERNARY}$$

To check that an assignment is well-typed:

1. Check that the expression on the right side of the = is well-typed with a right-value type “tval”
2. Check that “tval” is neither a pointer type nor an array reference type
3. Check that the expression on the left side is well-typed with an abstract left-value type

4. Check that the right-value type associated with this abstract left-value type is "tval"
5. Then the assignment is well-typed, and its type is "tval"

$$\frac{\Gamma \vdash e : \textbf{ALVal}\,(\tau^{val}) \quad \Gamma \vdash e' : \tau^{val} \quad \tau^{val} \neq \textbf{Ptr}\,(\tau^{val'}, as') \quad \tau^{val} \neq \textbf{Ref}\,(\tau^{val'}, as')}{\Gamma \vdash e = e' : \tau^{val}} \text{ ASSIGNMENT}$$

If an expression is well-typed and its type is an abstract left-value type, it can also be treated as if it were of the associated right-value type. If an expression is well-typed and its type is a left-value type, and its address space is not constant, it can also be treated as if it were of the associated abstract left-value type. If an expression is well-typed and its type is a left-value type, it can also be treated as if it were of the associated right-value type.

A variable name is well-typed if it is in the typing environment. In that case, its type is whatever it is mapped to in the typing environment,

An expression `&e` (respectively `*e`) is well-typed and with a pointer type (respectively with a left-value type) if `e` is well-typed with a left-value type (respectively of a pointer type). The associated right-value types and address spaces are left unchanged by these two operators.

An expression `@e` is well-typed and with an array reference type if `e` is well-typed with a left-value type. The associated right-value types and address spaces are left unchanged by this operator.

Note: The dynamic behaviour depends on whether the expression is a left-value array type or not, but it makes no difference during validation. `@x` for a variable `x` with a non-array type is valid, it will merely produce an array reference for which only the index 0 can be used.

$$\frac{\Gamma \vdash e : \textbf{ALVal}\,(\tau^{val})}{\Gamma \vdash e : \tau^{val}} \text{ ALVAL\_TO\_RVAL}$$

$$\frac{\Gamma \vdash e : \textbf{LVal}\,(\tau^{val}, as) \quad as \neq \textbf{constant}}{\Gamma \vdash e : \textbf{ALVal}\,(\tau^{val})} \text{ LVAL\_TO\_ALVAL}$$

$$\frac{\Gamma \vdash e : \textbf{LVal}\,(\tau^{val}, as)}{\Gamma \vdash e : \tau^{val}} \text{ LVAL\_TO\_RVAL}$$

$$\frac{x \mapsto \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ VARIABLE\_NAME}$$

$$\frac{\Gamma \vdash e : \textbf{LVal}\,(\tau^{val}, as)}{\Gamma \vdash \&e : \textbf{Ptr}\,(\tau^{val}, as)} \text{ ADDRESS\_TAKING}$$

$$\frac{\Gamma \vdash e : \textbf{Ptr}\,(\tau^{val}, as)}{\Gamma \vdash *e : \textbf{LVal}\,(\tau^{val}, as)} \text{ PTR\_DEREF}$$

$$\frac{\Gamma \vdash e : \textbf{LVal}\,(\tau^{val}, as)}{\Gamma \vdash @e : \textbf{Ref}\,(\tau^{val}, as)} \text{ TAKE\_REF\_LVAL}$$

To check a dot expression of the form `e.foo` (for an expression `e` and an identifier `foo`):

1. If `e` is well-typed
   (a) **If `e` has a left-value type, and there is a function called `operator&.foo` with a first parameter whose type is a poin**
   then the whole expression is well-typed, and has a left-value type corresponding to the right-value type and address-space of the return type of that function.
   (b) **Else if `e` has an abstract left-value type, and there is a function called `operator.foo=` with a first parameter whos**
   then the whole expression is well-typed, and has an abstract left-value type corresponding to the type of the second parameter of that function.
   (c) **Else if there is a function called `operator.foo` with a parameter whose type matches the type of `e`,**
   then the whole expression is well-typed and has the return type of that function.
   (d) Else the expression is ill-typed.
2. Else if `e` is an identifier
   (a) Check that there is an enum with that name in the global environment
   (b) Check that this enum has an element named `foo`
   (c) Then `e.foo` is well-typed, with the type of that enum
   (d) And replace it by the corresponding value

Note: Replacing e.foo by its value in the case of an enum is a bit weird of a thing to do at typing time, but it simplifies the writing of the execution rules if we can assume that every dot operator that we see corresponds to a getter, setter, or address-taker.

Note: Please note that a local variable declaration can shadow a global enum declaration.

To check that an array dereference `e1[e2]` is well-typed:

1. Check that `e2` is well-typed with the type `uint32`
2. Check that `e1` is well-typed
3. If the type of `e1` is an array reference whose associated type is `T`, then the whole expression is well-typed, and its type is a left-value with an associated type of `T`, and the same address space as the type of `e1`
4. Else if `e1` has a left-value type, and there is a function called `operator&[]` with a first parameter whose type is a pointer to the same right-value type with the same address space, then the whole expression is well-typed, and has a left-value type corresponding to the right-value type and address-space of the return type of that function.
5. **Else if `e1` has an abstract left-value type, and there is a function called `operator[]=` with a first parameter whose type i**
   then the whole expression is well-typed, and has an abstract left-value type corresponding to the type of the third parameter of that function.
6. **Else if there is a function called `operator.[]` with a parameter whose type matches the type of `e1`,**
   then the whole expression is well-typed and has the return type of that function.
7. Else the expression is ill-typed

$$\frac{\begin{array}{c}\Gamma \vdash e : \textbf{Ref}\,(\tau^{val}, as) \\ \Gamma \vdash e' : \textbf{uint32}\end{array}}{\Gamma \vdash e[e'] : \textbf{LVal}\,(\tau^{val}, as)} \quad \text{ARRAY\_REF\_INDEX}$$

To check that an expression `e++`, or `e--` is well-typed:

1. Check that `e` is well-typed, with an abstract left-value type
2. Check that a call to `operator+(e, 1)` (respectively `operator-(e, 1)`) would be well-typed, with a right-value type that matches `e`
3. Then the expression is well-typed, and of the right-value type of `e`

To check that an expression `e1 += e2`, `e1 -= e2`, `e1 *= e2`, `e1 /= e2`, `e1 %= e2`, `e1 ^= e2`, `e1 &= e2`, `e1 |= e2`, `e1 >>= e2`, or `e1 <<= e2`:

1. Check that `e1` is well-typed, with an abstract left-value type
2. Check that `e2` is well-typed
3. Check that a call to `operator+(e1, e2)` (respectively with the corresponding operators) would be well-typed, with a right-value type that matches `e1`
4. Then the expression is well-typed, and of the right-value type of `e1`

$$\frac{\Gamma \vdash e : \mathbf{ALVal}\left(\tau^{val}\right) \quad \Gamma \vdash \mathit{operator}+\left(e, \mathbf{integerLiteral}\right) : \tau^{val}}{\Gamma \vdash e++ : \tau^{val}} \text{ POSTFIX\_INCR}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{ALVal}\left({\tau^{val}}_1\right) \quad \Gamma \vdash e_2 : {\tau^{val}}_2 \quad \Gamma \vdash \mathit{operator}+\left(e_1, e_2\right) : {\tau^{val}}_1}{\Gamma \vdash e_1 {+=} e_2 : {\tau^{val}}_1} \text{ OPERATOR\_PLUS\_EQUAL}$$

To check that a function call is well-typed:

1. Check that each argument is well-typed
2. Make a set of all the functions in the global environment that share the same name and number of parameters, and that is not an entry point
3. For each function in that set:
   (a) Check that each argument can be given a type that match the type of the parameter
   (b) Otherwise, remove the function from the set
4. Check that the set now contains a single function
5. Then the function call is well-typed, and its type is the return type of that function

Note: Our overloading resolution is only this simple because this version of the language does not have generics.

Note: Entry points cannot be called from within a shader. A function is an entry point if and only if it is marked with one of `vertex`, `fragment` or `compute`.

Note: A consequence of the rule that overloading must be resolved without ambiguity is that if there are two implementations of a function `foo` that take respectively an int and a short, then the program `foo(42)` is invalid (as it could refer to either of these implementations). The programmer can easily make their intent clear with something like `int x = 42; foo(x);`.

Todo: This rule for overloading resolution is very different from the implementation, the two should be brought back in sync one way or another. https://github.com/gpuweb/WHLSL/issues/313

5.4 Phase 4. Annotations for execution

We resolved each overloaded function call in the previous section. They must now be annotated with which function is actually being called.

Every variable declaration, every function parameter, and every postfix increment/decrement must be associated with a unique store identifier. This identifier in turn refers to a set of contiguous bytes, of the right size; these sets are disjoint.

Each control barrier must be annotated with a unique barrier identifier.

Each branch, switch, loop, ternary expression, boolean or expression and boolean and expression must be annotated with a unique divergence point identifier.

Every variable declaration that does not have an initializing value, must get an initializing value that is the default value for its type. These default values are computed as follows:

- The default value for integer types is `0`
- The default value for floating point types is `0.0`
- The default value for booleans is `false`
- The default value for enums is the element of the enum whose associated integer values is 0
- The default value for pointers and array references is `null`
- The default value for an array is an array of the right size filled with the default values for its element type
- The default value for a structure type is a structure whose elements are all given their respective default values

Every load and store must also be annotated with a size in bytes.

- The size of primitive types, pointers and array references is implementation defined.
- The size of enums is the size of the underlying type
- The size of arrays is their number of elements multiplied by the size of one element
- The size of structs is computed in the same way as for C structs, and includes padding

Note: The fact that padding is included in the size, combined with the dynamic rules in the next section, means that copying a struct also copies any padding bits. This may be observable by the outside world depending on where the store occurs.

Finally, every array dereference (the `[]` operator) must be annotated with the stride, i.e. the size of the elements of the corresponding array. This size is computed in exactly the way described above. If the first operand is either an array or a left-value type associated with an array type, the access must also be annotated with the statically known size of the array.

5.5 Phase 5. Verifying the absence of recursion

WHLSL does not support recursion (for efficient compilation to GPUs). So once all overloaded function calls have been resolved, we must do one last check.

We create a relationship “may call” that connects two function declarations `f` and `g` if there is a call to `g` in the body of `f` (after resolving overloading). If this relationship is cyclic, then the program is invalid.

Note: This check is done on function declarations, not on function names, so if for example foo(int) calls foo(short), it is not considered recursion, as they are different functions after resolution of overloading.

CHAPTER

SIX

DYNAMIC RULES

6.1 Definitions

We split the semantics in two parts: a per-thread execution semantics that does not know anything about concurrency or the memory, and a global set of rules for loads, stores, barriers and the like.

The per-thread semantics is a fairly classic small-step operational semantics, meaning that it describes a list of possible transitions that the program can take in one step. The per-thread state is made of a few element:

- The program being executed. Each transition transforms it.
- A divergence stack. This is a stack of pairs of divergence point identifiers and values, which tracks whether we are in a branch, and is used by the rules for barriers and derivatives to check that control-flow is uniform.
- An environment. This is a mapping from variable names to values and is used to keep track of arguments and variables declared in the function.

Each transition is a statement of the form "With environment ρ, if some conditions are respected, the program may be transformed into the following, emitting the following memory events, and modifying the divergence stack in these ways."

In some of these rules we use `ASSERT` to provide some properties that are true either by construction or thanks to the validation rules of the previous section. Such assertions are not tests that must be done by any implementation, they are merely hints to our intent.

6.2 Execution of statements

6.2.1 Blocks and variable declarations

The program fragments that we use to define our semantics are richer than just the syntactically correct programs. In particular, we allow annotating blocks (sequences of statements between braces) with an environment. This is useful to formalize lexical scoping.

Here is how to reduce a block by one step:

1. If the block is not annotated, annotate it with the environment
2. If the first statement of the block is an empty block, remove it
3. Else if the first statement of the block is a terminator (break, continue, fallthrough, or return), replace the entire block by it.
4. Else if the first statement of the block is a variable declaration:

   (a) Make a new environment from the one that annotates the block, mapping the variable name to its store identifier.
   (b) If the variable declaration has an initializing expression that can be reduced, reduce it using the new environment
   (c) Else:
      i. Change the annotation of the block to the new environment.
      ii. Emit a store to the store identifier of the declaration, of the initializing value
      iii. Remove this variable declaration from the block

5. Else reduce the first statement of the block, using the environment that the block was annotated with (not the top-level environment)

$$\frac{}{\rho \vdash \{\, s_0 \mathrel{..} s_n \} \rightarrow \{_\rho\, s_0 \mathrel{..} s_n \}} \quad \text{BLOCK\_ANNOTATE}$$

$$\frac{}{\rho_{out} \vdash \{_\rho\, \{_{\rho'}\, \}\, s_1 \mathrel{..} s_n \} \rightarrow \{_\rho\, s_1 \mathrel{..} s_n \}} \quad \text{BLOCK\_NEXT\_STMT}$$

$$\frac{s = \textbf{break}\,; \vee s = \textbf{continue}\,; \vee s = \textbf{fallthrough}\,; \vee s = \textbf{return}\, rval; \vee s = \textbf{return}\,;}{\rho_{out} \vdash \{_\rho\, s\, s_1 \mathrel{..} s_n \} \rightarrow s} \quad \text{BLOCK\_TERMINATOR}$$

$$\frac{\rho[x \mapsto \textbf{LVal}\,(\textbf{sid})] \vdash e \xrightarrow[S]{E} e'}{\rho_{out} \vdash \{_\rho\, \tau^{val}\, x : \textbf{sid} \ = e;\ s_1 \mathrel{..} s_n \} \xrightarrow[S]{E} \{_\rho\, \tau^{val}\, x : \textbf{sid} \ = e';\ s_1 \mathrel{..} s_n \}} \quad \text{BLOCK\_VDECL\_REDUCE}$$

$$\frac{\rho' = \rho[x \mapsto \textbf{LVal}\,(\textbf{sid})]}{\rho_{out} \vdash \{_\rho\, \tau^{val}\, x : \textbf{sid} \ = rv;\ s_1 \mathrel{..} s_n \} \xrightarrow{\textbf{sid} \leftarrow rv} \{_{\rho'}\, s_1 \mathrel{..} s_n \}} \quad \text{BLOCK\_VDECL\_COMPLETE}$$

$$\frac{\begin{array}{c}\rho' = \rho[x \mapsto \textbf{LVal}\,(\textbf{sid})] \\ rv = \textbf{Default}\,(\tau^{val})\end{array}}{\rho_{out} \vdash \{_\rho\, \tau^{val}\, x : \textbf{sid}\,;\ s_1 \mathrel{..} s_n \} \xrightarrow{\textbf{sid} \leftarrow rv} \{_{\rho'}\, s_1 \mathrel{..} s_n \}} \quad \text{BLOCK\_VDECL}$$

$$\frac{\rho \vdash s \xrightarrow[S]{E} s'}{\rho_{out} \vdash \{_\rho\, s\, s_1 \mathrel{..} s_n \} \xrightarrow[S]{E} \{_\rho\, s'\, s_1 \mathrel{..} s_n \}} \quad \text{BLOCK\_REDUCE}$$

6.2.2 Branches

We add another kind of statement: the `Join(s)` construct, that takes as argument another statement `s`.

Here is how to reduce a branch (if-then-else construct, remember that if-then is just syntactic sugar that was eliminated during parsing) by one step:

1. If the expression in the if is `true` or `false`.
   (a) Push that value and the divergence point identifier of the branch on the divergence stack
   (b) Replace the branch by the statement in the then (for `true`) or else (for `false`) branch, wrapped in the `Join` construct
2. Else reduce that expression

$$\frac{}{\rho \vdash \textbf{if } \mathit{dpid}(\textbf{true}) s \textbf{ else } s' \xrightarrow[\textbf{push}\,(\textbf{true}, \mathit{dpid})]{} \textbf{Join}\,(s)} \quad \text{IF\_TRUE}$$

$$\frac{}{\rho \vdash \textbf{if } \mathit{dpid}(\textbf{false}) s \textbf{ else } s' \xrightarrow[\textbf{push}\,(\textbf{false}, \mathit{dpid})]{} \textbf{Join}\,(s')} \quad \text{IF\_FALSE}$$

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash \textbf{if } \mathit{dpid}(e) s \textbf{ else } s' \xrightarrow[S]{E} \textbf{if } \mathit{dpid}(e') s \textbf{ else } s'} \quad \text{IF\_REDUCE}$$

Here is how to reduce a `Join(s)` statement:

1. If the argument of the `Join` is a terminator (`break;`, `continue;`, `fallthrough;`, or `return e?;`) or an empty block
   (a) ASSERT(the divergence stack is not empty)
   (b) Pop the last element from the divergence stack
   (c) Replace the `Join` statement by its argument
2. Else reduce its argument

$$\frac{s = \textbf{break}\,; \lor s = \textbf{continue}\,; \lor s = \textbf{fallthrough}\,; \lor s = \textbf{return}\,; \lor s = \textbf{return}\, e; \lor s = \{_\rho\}}{\rho \vdash \textbf{Join}\,(s) \xrightarrow[\textbf{pop}\,()]{} s} \quad \text{JOIN\_ELIM}$$

$$\frac{\rho \vdash s \xrightarrow[S]{E} s'}{\rho \vdash \textbf{Join}\,(s) \xrightarrow[S]{E} \textbf{Join}\,(s')} \quad \text{JOIN\_REDUCE}$$

Note: Popping the last element from the divergence stack never fails, as a Join only appears when eliminating a branch, which pushes a value on it.

6.2.3 Switches

We add another kind of statement: the `Cases(..)` construct that takes as argument a sequence of statements. Informally it represents the different cases of a switch, and deals with the `fallthrough;` and `break;` statements.

Here is how to reduce a switch statement by one step:

1. If the expression in the switch can be reduced, reduce it by one step
2. Else if it is an integer or enum value `val` and there is a `case val:` in the switch:
   (a) Wrap the corresponding sequence of statements into a block (turning it into a single statement)
   (b) Do the same for each sequence of statements until the end of the switch
   (c) Replace the entire switch by a `Cases` construct, taking as argument these resulting statements in source order
   (d) Push `val` and the divergence point identifier of the switch on the divergence stack
3. Else
   (a) ASSERT(the expression in the switch is an integer or enum value `val`)

   (b) ASSERT(there is a `default:` case in the switch)

   (c) Find the `default` case, and wrap the corresponding sequence of statements into a block (turning it into a single statement)

   (d) Do the same for each sequence of statements until the end of the switch

   (e) Replace the entire switch by a `Cases` construct, taking as argument these resulting statements in source order

   (f) Push `val` and the divergence point identifier of the switch on the divergence stack

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash \textbf{switch}\ dpid(e)\{sc_0 : sblock_0 \mathinner{..} sc_n : sblock_n\} \xrightarrow[S]{E} \textbf{switch}\ dpid(e')\{sc_0 : sblock_0 \mathinner{..} sc_n : sblock}$$

$$\frac{s = \{sblock\} \wedge s_0 = \{sblock'_0\} \wedge \mathinner{..} \wedge s_m = \{sblock'_m\}}{\rho \vdash \textbf{switch}\ dpid(rv)\{sc_0 : sblock_0 \mathinner{..} sc_n : sblock_n\ \textbf{case}\ rv : sblock\ sc'_0 : sblock'_0 \mathinner{..} sc'_m : sblock'_m\} \xrightarrow[\textbf{push}\,(rv, dpid)]{} \textbf{Cases}\,(s, s_0, \mathinner{..}, s}$$

$$\frac{\begin{array}{l} rv\ \textbf{not} \in sc_0 \mathinner{..} sc_n \\ rv\ \textbf{not} \in sc'_0 \mathinner{..} sc'_m \\ s = \{sblock\} \wedge s_0 = \{sblock'_0\} \wedge \mathinner{..} \wedge s_m = \{sblock'_m\} \end{array}}{\rho \vdash \textbf{switch}\ dpid(rv)\{sc_0 : sblock_0 \mathinner{..} sc_n : sblock_n\ \textbf{default} : sblock\ sc'_0 : sblock'_0 \mathinner{..} sc'_m : sblock'_m\} \xrightarrow[\textbf{push}\,(rv, dpid)]{} \textbf{Cases}\,(s, s_0, \mathinner{..}, s}$$

Here is how to reduce a `Cases` construct by one step:

1. ASSERT(the construct has at least one argument)
2. If the first argument is the `fallthrough;` statement, remove it (reducing the total number of arguments by 1)
3. Else if the first argument is the `break;` statement:
   (a) ASSERT(the divergence stack is not empty)

   (b) Pop the last element from the divergence stack

   (c) Replace the entire construct by an empty block
4. Else if the first argument is another terminator statement, that cannot be reduced (i.e. `continue;`, `return value;` or `return;`)
   (a) ASSERT(the divergence stack is not empty)

   (b) Pop the last element from the divergence stack

   (c) Replace the entire construct by its first argument
5. Else reduce the first argument by one step

$$\frac{}{\rho \vdash \textbf{Cases}\,(\textbf{fallthrough}\,;\,,s_0,\,..\,,s_n) \to \textbf{Cases}\,(s_0,\,..\,,s_n)} \quad \text{CASES\_FALLTHROUGH}$$

$$\frac{}{\rho \vdash \textbf{Cases}\,(\textbf{break}\,;\,,s_0,\,..\,,s_n) \xrightarrow[\textbf{pop}\,()]{} \{\,\}} \quad \text{CASES\_BREAK}$$

$$\frac{s = \textbf{continue}\,; \vee s = \textbf{return}\,; \vee s = \textbf{return}\; rval;}{\rho \vdash \textbf{Cases}\,(s, s_0,\,..\,,s_n) \xrightarrow[\textbf{pop}\,()]{} s} \quad \text{CASES\_OTHER\_TERMINATOR}$$

$$\frac{\rho \vdash s \xrightarrow[S]{E} s'}{\rho \vdash \textbf{Cases}\,(s, s_0,\,..\,,s_n) \xrightarrow[S]{E} \textbf{Cases}\,(s', s_0,\,..\,,s_n)} \quad \text{CASES\_REDUCE}$$

6.2.4 Loops

We add yet another kind of statement: the `Loop(s, s', s'')` construct that takes as arguments three statements. Informally, its first argument represents the current iteration of a loop, its second argument is to be executed at the end of the iteration, and its third argument is a continuation for the rest of the loop.

Any `do s while(e);` statement is reduced to the following in one step: `Loop(s, {}, if(e) do s while(e); else {})`, keeping the same divergence point identifier. Any `for (;e;e') s` statement is reduced to the following in one step `if (e) Loop(s, e';, for(;e;e') s) else {}`, keeping the same divergence point identifier.

$$\frac{}{\rho \vdash \textbf{do}\; dpid\; s\; \textbf{while}\,(e); \to \textbf{Loop}\,(s, \{\,\}, \textbf{if}\; dpid(e)\textbf{do}\; dpid\; s\; \textbf{while}\,(e);\; \textbf{else}\,\{\,\})} \quad \text{DO\_WHILE\_LOOP}$$

$$\frac{}{\rho \vdash \textbf{for}\; dpid(;e;e')s \to \textbf{if}\; dpid(e)\textbf{Loop}\,(s, e';, \textbf{for}\; dpid(;e;e')s)\;\textbf{else}\,\{\,\}} \quad \text{FOR\_LOOP}$$

Note: while loops are desugared into do while loops, see *Parsing*.

Note: we only treat the case where for loops have no initialization, because it is desugared, see *Parsing*.

Here is how to reduce a `Loop(s, s')` statement by one step:

1. If `s` is the `break;` statement, replace the whole construct by the empty block: `{}`
2. Else if `s` is the empty block or the `continue;` statement
   (a) If the second argument `s'` is the empty statement, replace the whole construct by its third argument `s''`
   (b) Else reduce `s'` by a step
3. Else if `s` is another terminator (`fallthrough;`, `return;` or `return rval;`), replace the whole construct by it
4. Else reduce `s` by one step

$$\frac{}{\rho \vdash \mathbf{Loop}\,(\mathbf{break}\,;, s_2, s_3) \to \{\,\}}\quad \text{LOOP\_BREAK}$$

$$\frac{s_1 = \{_{\rho'}\,\} \vee s_1 = \mathbf{continue}\,;}{\rho \vdash \mathbf{Loop}\,(s_1, \{_{\rho''}\,\}, s_2) \to s_2}\quad \text{LOOP\_NEXT\_ITERATION}$$

$$\frac{\begin{array}{l} s_1 = \{_{\rho'}\,\} \vee s_1 = \mathbf{continue}\,; \\ \rho \vdash s_2 \xrightarrow[S]{E} s_2' \end{array}}{\rho \vdash \mathbf{Loop}\,(s_1, s_2, s_3) \xrightarrow[S]{E} \mathbf{Loop}\,(s_1, s_2', s_3)}\quad \text{LOOP\_INCREMENT}$$

$$\frac{s_1 = \mathbf{fallthrough}\,; \vee s_1 = \mathbf{return}\,; \vee s_1 = \mathbf{return}\,rval;}{\rho \vdash \mathbf{Loop}\,(s_1, s_2, s_3) \to s_1}\quad \text{LOOP\_OTHER\_TERMINATOR}$$

$$\frac{\rho \vdash s_1 \xrightarrow[S]{E} s_1'}{\rho \vdash \mathbf{Loop}\,(s_1, s_2, s_3) \xrightarrow[S]{E} \mathbf{Loop}\,(s_1', s_2, s_3)}\quad \text{LOOP\_REDUCE}$$

Note: These operations do not need to explicitly modify the divergence stack, because each iteration of a loop executes an `if` statement that does it.

6.2.5 Barriers and uniform control flow

There is no rule in the per-thread semantics for *control barriers*. Instead, there is a rule in the global semantics, saying that if all threads are at a control barrier instruction with the same identifier, and their divergence stacks are identical, then they may all advance atomically, replacing the barrier by an empty block.

6.2.6 Other

If a statement is just an expression (`effectfulExpr` in the grammar), it is either discarded (if it is a value) or reduced by one step (otherwise).

If a statement is a return followed by an expression, and the expression can be reduced, then the statement can as well by reducing the expression.

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash e; \xrightarrow[S]{E} e';}\quad \text{EFFECTFUL\_EXPR\_REDUCE}$$

$$\frac{}{\rho \vdash rval; \to \{\,\}}\quad \text{EFFECTFUL\_EXPR\_ELIM}$$

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash \mathbf{return}\,e; \xrightarrow[S]{E} \mathbf{return}\,e';}\quad \text{RETURN\_REDUCE}$$

The standard library also offers atomic operations and fences (a.k.a. *memory barriers*, not to be confused with *control barriers*). Each of these emit a specific memory event when they are executed, whose semantics is described in the memory model section.

6.3 Execution of expressions

We define the following kinds of values:

- Integers, floats, booleans and other primitives provided by the standard library
- Pointers. These have an address and an address space
- Left values. These also have an address and an address space
- A special Invalid left-value, used to represent the dereferencing of out-of-bounds accesses and the dereferencing of `null`
- Array references. These have a base address, an address space and a size
- Struct values. These are a sequence of bytes of the right size, and can be interpreted as a tuple of their elements (plus padding bits)

Note: Abstract left-value types were used in the typing section to represent things that can be assigned to. At runtime they are either left-values, or normal values with a setter applicable to them.

In this section we describe how to reduce each kind of expression to another expression or to a value. Left values are the only kind of values that can be further reduced.

6.3.1 Operations affecting control-flow

Just like we added `Join`, `Cases` and `Loop` construct to deal with control-flow affecting statements, we add a `JoinExpr` construct to deal with control-flow affecting expressions. `JoinExpr` takes as argument an expression and return an expression. Its only use is (informally) as a marker that the divergence stack will have to be popped to access its content.

There are three kinds of expressions that can cause a divergence in control-flow: the boolean and (i.e. `&&`, that short-circuits), the boolean or (i.e. `||`, that also short-circuits), and ternary conditions.

To reduce a boolean and by one step:

1. If its first operand can be reduced, reduce it
2. Else if its first operand is `false`, replace the whole operation by `false`.
3. Else
   (a) ASSERT(its first operand is `true`)
   (b) Push `true` and the corresponding divergence point identifier on the divergence stack.
   (c) Replace the whole operation by its second operand wrapped in a `JoinExpr` construct.

$$\frac{\rho \vdash e_0 \xrightarrow[S]{E} e_0'}{\rho \vdash e_0 \mathbin{\&\&} e_1 \xrightarrow[S]{E} e_0' \mathbin{\&\&} e_1} \quad \text{AND\_REDUCE}$$

$$\frac{}{\rho \vdash \textbf{false} \mathbin{\&\&} e \rightarrow \textbf{false}} \quad \text{AND\_FALSE}$$

$$\frac{}{\rho \vdash \textbf{true} \mathbin{\&\&} e \xrightarrow[\textbf{push}\,(\textbf{true}, dpid)]{} \textbf{JoinExpr}\,(e)} \quad \text{AND\_TRUE}$$

Very similarly, to reduce a boolean or by one step:

1. If its first operand can be reduced, reduce it
2. Else if its first operand is `true`, replace the whole operation by `true`.
3. Else
   (a) ASSERT(its first operand is `false`)
   (b) Push `false` and the corresponding divergence point identifier on the divergence stack.
   (c) Replace the whole operation by its second operand wrapped in a `JoinExpr` construct.

$$\frac{\rho \vdash e_0 \xrightarrow[S]{E} e_0'}{\rho \vdash e_0 \mid\mid e_1 \xrightarrow[S]{E} e_0' \mid\mid e_1} \quad \text{OR\_REDUCE}$$

$$\frac{}{\rho \vdash \mathbf{true} \mid\mid e \rightarrow \mathbf{true}} \quad \text{OR\_TRUE}$$

$$\frac{}{\rho \vdash \mathbf{false} \mid\mid e \xrightarrow[\mathbf{push}\,(\mathbf{false}, dpid)]{} \mathbf{JoinExpr}\,(e)} \quad \text{OR\_FALSE}$$

To reduce a ternary condition by one step:

1. If its first operand can be reduced, reduce it
2. Else if its first operand is `true`
   (a) Push `true` and the corresponding divergence point identifier on the divergence stack.
   (b) Replace the whole operation by its second operand wrapped in a `JoinExpr` construct
3. Else
   (a) ASSERT(its first operand is `false`)
   (b) Push `false` and the corresponding divergence point identifier on the divergence stack.
   (c) Replace the whole operation by its third operand wrapped in a `JoinExpr` construct.

$$\frac{\rho \vdash e_0 \xrightarrow[S]{E} e_0'}{\rho \vdash e_0 \;?\; e_1 : e_2 \xrightarrow[S]{E} e_0' \;?\; e_1 : e_2} \quad \text{TERNARY\_REDUCE}$$

$$\frac{}{\rho \vdash \mathbf{true} \;?\; e_1 : e_2 \xrightarrow[\mathbf{push}\,(\mathbf{true}, dpid)]{} \mathbf{JoinExpr}\,(e_1)} \quad \text{TERNARY\_TRUE}$$

$$\frac{}{\rho \vdash \mathbf{false} \;?\; e_1 : e_2 \xrightarrow[\mathbf{push}\,(\mathbf{false}, dpid)]{} \mathbf{JoinExpr}\,(e_2)} \quad \text{TERNARY\_FALSE}$$

To reduce a `JoinExpr` by one step:

1. If its operand is not a lvalue, and can be reduced, then reduce it by one step
2. Else:
   (a) ASSERT(the divergence stack is not empty)
   (b) Pop the last element from the divergence stack
   (c) Replace the whole expression by its operand

$$\frac{}{\rho \vdash \textbf{JoinExpr}\,(val) \xrightarrow[\textbf{pop}\,()]{} val}\quad \text{JOIN\_EXPR\_ELIM}$$

$$\frac{e \neq val \qquad \rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash \textbf{JoinExpr}\,(e) \xrightarrow[S]{E} \textbf{JoinExpr}\,(e')}\quad \text{JOIN\_EXPR\_REDUCE}$$

6.3.2 Variables

A variable name can be reduced in one step into whatever that name binds in the current environment. This does not require any memory access: it is purely used to represent scoping, and most names just bind to lvalues.

To reduce a (valid) lvalue:

1. Emit a load to the corresponding address, of a size appropriate for the type of the value
2. If the type of the expression was an enum type, and the value loaded is not a valid value of that type, replace it by an unspecified valid value of that type
3. Replace the whole expression by this value

Note: The 2nd step is to prevent races from allowing the creation of invalid enum values, which could cause problems to switches without default cases. We don't need a similar rules for pointers or array references, because we do not allow potentially racy assignments to variables of these types.

To reduce an invalid left-value, any of the following is acceptable:

- Trap
- Replace it by the default value of that type.

Todo: We should extend this possible behavior to also accept (0,0,0,X) for some specific values of X for "vector reads" to match https://github.com/gpuweb/spirv-execution-env/blob/master/execution-env.md I just have to figure out what exactly these vector reads map to in WHLSL. https://github.com/gpuweb/WHLSL/issues/316

6.3.3 Reduction to an abstract left-value

Todo: my naming here is utterly terrible, I really should find better names for these things.

We now define a notion of "reducing `e` one step to an abstract left-value". This will be used to define how much to reduce things on the left-side of assignments. For example, in "x = y", we do not want to reduce "x" all the way to a load, although we do want to reduce "y" to a load. Here is the definition:

1. If `e` is of the form `e1.foo`
   (a) If `e1` can be reduced one step to an abstract left-value, do it
   (b) Else if `e` had a left-value type and `e1` is a left value (valid or not), replace the whole expression by `* operator&.foo(&e1)`, using the instance of `operator&.foo` that was used to give a left-value type to `e`.
   (c) Else fail

2. Else if `e` is of the form `e1[e2]`
   (a) If `e1` can be reduced one step to an abstract left-value, do it
   (b) Else if `e1` had an array reference type and can be reduced one step (normally), do it
   (c) Else if `e2` can be reduced one step (normally), do it
   (d) Else if `e1` is `null`, replace the whole expression by an invalid left-value.
   (e) Else if `e1` is an array reference:
      i. ASSERT(`e2` is an integer)
      ii. If `e2` is out of the bounds of `e1`, either replace the whole expression by an invalid left-value, or replace `e2` by an unspecified in-bounds value.
      iii. Else replace the whole expression by a left-value, to an address computed by adding the address in `e1` to the product of `e2` and the stride computed from the type of `e1`'s elements.
   (f) Else if `e` had a left-value type and `e1` is a left value (valid or not), replace the whole expression by `* operator&[](&e1, e2)` using the instance of `operator&[]` that was used to give a left-value type to `e`.
   (g) Else fail
3. Else if `e` is not a lValue (valid or not)
   (a) ASSERT(`e` can be reduced)
   (b) Reduce `e`

Note: in the rules we say "e is an abstract left-value" as a short hand for "e cannot be reduced further to an abstract left value"

$$\frac{\rho \Vdash e \xrightarrow[S]{E} e'}{\rho \Vdash e.foo \xrightarrow[S]{E} e'.foo} \quad \text{ALVAL\_DOT\_REDUCE}$$

$$\frac{\begin{array}{c} e.foo \textbf{ has a left value type} \\ e = \textbf{LVal}\,(addr) \lor e = \textbf{Invalid\_LVal} \end{array}}{\rho \Vdash e.foo \to *operator\&.foo(\&e)} \quad \text{ALVAL\_DOT\_ANDER}$$

$$\frac{\rho \Vdash e_1 \xrightarrow[S]{E} e_1'}{\rho \Vdash e_1[e_2] \xrightarrow[S]{E} e_1'[e_2]} \quad \text{ALVAL\_ARRAY\_REDUCE\_LEFT}$$

$$\frac{\begin{array}{l} e_1 \textbf{ is an abstract left value} \\ e_1 \textbf{ has an array reference type} \\ \rho \vdash e_1 \xrightarrow[S]{E} e_1' \end{array}}{\rho \Vdash e_1[e_2] \xrightarrow[S]{E} e_1'[e_2]} \quad \text{ALVAL\_ARRAY\_REF\_REDUCE}$$

$$\frac{\begin{array}{l} \rho \vdash e_2 \xrightarrow[S]{E} e_2' \\ e_1 \textbf{ is an abstract left value} \\ e_1 \textbf{ does not have an array reference type} \lor e_1 = rval \end{array}}{\rho \Vdash e_1[e_2] \xrightarrow[S]{E} e_1[e_2']} \quad \text{ALVAL\_ARRAY\_REDUCE\_RIGHT}$$

$$\frac{}{\rho \Vdash \textbf{null}[i] \to \textbf{Invalid\_LVal}} \quad \text{ARRAY\_NULL\_ACCESS}$$

$$\frac{i \geq j}{\rho \Vdash \textbf{Ref}\,(addr, j)[i] \to \textbf{Invalid\_LVal}} \quad \text{ARRAY\_REF\_INVALID}$$

$$\frac{\begin{array}{c} i \geq k \\ j < k \end{array}}{\rho \Vdash \textbf{Ref}\,(addr, k)[i] \to \textbf{Ref}\,(addr, k)[j]} \quad \text{ARRAY\_REF\_CLAMPED}$$

$$\frac{\begin{array}{l} addr' = addr + i * \textbf{stride} \\ i < j \end{array}}{\rho \Vdash \textbf{Ref}\,(addr, j)[i] \to \textbf{LVal}\,(addr')} \quad \text{ARRAY\_REF\_VALID}$$

$$\frac{\begin{array}{c} e_1[rval] \textbf{ has a left value type} \\ e_1 = \textbf{LVal}\,(addr) \lor e_1 = \textbf{Invalid\_LVal} \end{array}}{\rho \Vdash e_1[rval] \to *operator\&[](\&e_1, rval)} \quad \text{ALVAL\_ARRAY\_ANDER}$$

$$\frac{\begin{array}{l} \rho \vdash e \xrightarrow[S]{E} e' \\ e \neq \textbf{LVal}\,(addr) \\ e \neq \textbf{Invalid\_LVal} \\ e \neq e_1[e_2] \end{array}}{\rho \Vdash e \xrightarrow[S]{E} e'} \quad \text{ALVAL\_GENERIC\_REDUCE}$$

6.3.4 Assignment

To reduce an assignment `e1 = e2`:

1. If `e1` can be reduced to an abstract left-value, do it
2. Else if `e2` can be reduced, reduce it.
3. Else if `e1` is a valid lvalue
   (a) Emit a store to the address of the lvalue, of the value on the right of the equal, of a size appropriate for the type of that value
   (b) Replace the entire expression by the value on the right of the equal.
4. Else if `e1` is an invalid lvalue, either replace the whole expression by `e2` or trap
5. Else if `e1` is of the form `e3.foo`
   (a) ASSERT(`e1` had an abstract left-value type)
   (b) Replace the whole expression by an assignment to `e3` of the result of a call to `operator.foo=` with the arguments `e3` and `e2`, using the instance of `operator.foo=` that was used to give an abstract left-value type to `e1`.
6. Else
   (a) ASSERT(`e1` is of the form `e3[e4]`)
   (b) ASSERT(`e1` had an abstract left-value type)
   (c) Replace the whole expression by an assignment to `e3` of the result of a call to `operator[]=` with the arguments `e3`, `e4`, and `e2`, using the instance of `operator[]=` that was used to give an abstract left-value type to `e1`.

$$\frac{\rho \Vdash e_0 \xrightarrow[S]{E} e_0'}{\rho \vdash e_0 = e_1 \xrightarrow[S]{E} e_0' = e_1} \quad \text{ASSIGN\_LEFT\_REDUCE}$$

$$\frac{\begin{array}{c}\rho \vdash e_1 \xrightarrow[S]{E} e_1' \\ e_0 \textbf{ is an abstract left value}\end{array}}{\rho \vdash e_0 = e_1 \xrightarrow[S]{E} e_0 = e_1'} \quad \text{ASSIGN\_RIGHT\_REDUCE}$$

$$\frac{}{\rho \vdash \textbf{LVal}\,(addr) = rval \xrightarrow{addr \leftarrow rval} rval} \quad \text{ASSIGN\_EXECUTE}$$

$$\frac{}{\rho \vdash \textbf{Invalid\_LVal} = rval \rightarrow rval} \quad \text{ASSIGN\_INVALID\_IGNORE}$$

$$\frac{}{\rho \vdash \textbf{Invalid\_LVal} = rval \rightarrow \textbf{TrapValue}} \quad \text{ASSIGN\_INVALID\_TRAP}$$

$$\frac{e_3.foo \textbf{ is an abstract left value}}{\rho \vdash e_3.foo = rval \rightarrow e_3 = operator.foo = (e_3, rval)} \quad \text{ASSIGN\_SETTER}$$

$$\frac{e_3[e_4] \textbf{ is an abstract left value}}{\rho \vdash e_3[e_4] = rval \rightarrow e_3 = operator[] = (e_3, e_4, rval)} \quad \text{ASSIGN\_INDEXED\_SETTER}$$

6.3.5 Pointers and references

WHLSL has both pointers and array references. Pointers let the programmer access a specific memory location, but do not allow any pointer arithmetic. Array references are actually bounds-checked fat-pointers.

The `&` and `*` operators simply convert between left-values and pointers. To reduce `& e`:

1. If `e` can be reduced to an abstract left-value, do it
2. Else if `e` is an invalid lvalue, either replace the whole expression with null, or trap
3. Else ASSERT(`e` is a valid lvalue), and replace the whole expression by a pointer to the same address

To reduce `* e`:

1. If `e` is null, either trap or replace the whole expression by an invalid left-value
2. Else if `e` is a pointer, replace the whole expression by a lvalue to the same address in the same address-space
3. Else reduce `e`

$$\frac{\rho \Vdash e \xrightarrow[S]{E} e'}{\rho \vdash \&e \xrightarrow[S]{E} \&e'} \quad \text{TAKE\_PTR\_REDUCE}$$

$$\frac{}{\rho \vdash \&\mathbf{Invalid\_LVal} \rightarrow \mathbf{null}} \quad \text{TAKE\_PTR\_INVALID\_NULL}$$

$$\frac{}{\rho \vdash \&\mathbf{Invalid\_LVal} \rightarrow \mathbf{TrapValue}} \quad \text{TAKE\_PTR\_INVALID\_TRAP}$$

$$\frac{}{\rho \vdash \&\mathbf{LVal}\,(addr) \rightarrow \mathbf{Ptr}\,(addr)} \quad \text{TAKE\_PTR\_LVAL}$$

$$\frac{}{\rho \vdash *\mathbf{null} \rightarrow \mathbf{TrapValue}} \quad \text{DEREF\_NULL\_TRAP}$$

$$\frac{}{\rho \vdash *\mathbf{null} \rightarrow \mathbf{Invalid\_LVal}} \quad \text{DEREF\_NULL\_INVALID}$$

$$\frac{}{\rho \vdash *\mathbf{Ptr}\,(addr) \rightarrow \mathbf{LVal}\,(addr)} \quad \text{DEREF\_PTR}$$

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash *e \xrightarrow[S]{E} *e'} \quad \text{DEREF\_REDUCE}$$

6.3.6 Arrays

The `@` operator is used to turn a lvalue into an array reference, using the size information computed during typing to set the bounds. More precisely, to reduce `@ e`:

1. If `e` is an LValue and was of type LValue of an array of size `n` during typing, replace it by an array reference to the same address, same address space, and with a bound of `n`
2. Else if it is an LValue and was of type LValue of a non-array type during typing, replace it by an array reference to the same address, same address space, and with a bound of `1`
3. Else if it is an invalid lvalue, either replace the whole expression by null, or trap

4. Else reduce it

$$\frac{}{\rho \vdash @\mathbf{LVal}\,(addr) \rightarrow \mathbf{Ref}\,(addr, i)}\ \text{MAKE\_REF\_LVAL}$$

$$\frac{}{\rho \vdash @\mathbf{Invalid\_LVal} \rightarrow \mathbf{null}}\ \text{MAKE\_REF\_INVALID\_NULL}$$

$$\frac{}{\rho \vdash @\mathbf{Invalid\_LVal} \rightarrow \mathbf{TrapValue}}\ \text{MAKE\_REF\_INVALID\_TRAP}$$

$$\frac{\begin{array}{c} e \neq \mathbf{LVal}\,(addr) \\ e \neq \mathbf{Invalid\_LVal} \\ \rho \vdash e \xrightarrow[S]{E} e' \end{array}}{\rho \vdash @e \xrightarrow[S]{E} @e'}\ \text{MAKE\_REF\_REDUCE}$$

There is no explicit dereferencing operator for array references: they can just be used with the array syntax. The `[]` dereferencing operator is polymorphic: its first operand can be either an array reference, or a value for which the relevant operators (`operator&[]`, `operator[]=`, or `operator[]`) are defined. To reduce `e1[e2]` by one step:

1. If the whole expression can be reduced to an abstract left-value, do it
2. Else replace the whole expression by `operator[](e1, e2)`, using the instance of `operator[]` that was used during the typing of this array dereference.

Note: In the case where `operator&[]` can be used, it will be used through the rules for reduction to an abstract left-value (see *Reduction to an abstract left-value*). For `operator[]=`, see the section on assignment (*Assignment*).

$$\frac{\rho \Vdash e_1[e_2] \xrightarrow[S]{E} e_3}{\rho \vdash e_1[e_2] \xrightarrow[S]{E} e_3}\ \text{ARRAY\_DEREF\_REDUCE}$$

$$\frac{e_1[e_2]\ \mathbf{is\ an\ abstract\ left\ value}}{\rho \vdash e_1[e_2] \rightarrow operator[](e_1, e_2)}\ \text{ARRAY\_DEREF\_GETTER}$$

6.3.7 Dot operator

The dot operator is used for two purposes: accessing the fields of structs (or custom `operator&.foo`, `operator.foo=`, `operator.foo`), and getting an element of an enum. Since we already eliminated the case where it is used to get an element of an enum (see *Typing expressions*), we only have to deal with the getters/setters/address-takers. Additionally, it can be overloaded (through `operator&.foo`, `operator.foo=` and `operator.foo`). To reduce `e.foo` for some identifier `foo`:

1. If the whole expression can be reduced to an abstract left-value, do it
2. Else replace the whole expression by `operator.foo(e1)`, using the instance of `operator.foo` that was used during the typing of this dot operator.

Note: In the case where `operator&.foo` can be used, it will be used through the rules for reduction to an abstract left-value (see *Reduction to an abstract left-value*). For `operator.foo=`, see the section on assignment (*Assignment*).

$$\frac{\rho \Vdash e_1.foo \xrightarrow[S]{E} e_2}{\rho \vdash e_1.foo \xrightarrow[S]{E} e_2} \text{ DOT\_REDUCE}$$

$$\frac{e_1.foo\,\textbf{is an abstract left value}}{\rho \vdash e_1.foo \rightarrow operator.foo(e_1)} \text{ DOT\_GETTER}$$

6.3.8 Read-modify-write expressions

To reduce an expression `e++` or `e--`:

1. If `e` can be reduced to an abstract left value, do it
2. Else:
   (a) Let `addr` be a fresh address
   (b) Replace the whole expression by `LVal(addr) = e, e = LVal(addr) + 1, LVal(addr)` (replacing the + by - for `e--`)

Note: depending on `e`, this can lead to calls to getters/setters or address takers.

$$\frac{\rho \Vdash e \xrightarrow[S]{E} e'}{\rho \vdash e++ \xrightarrow[S]{E} e'++} \text{ POSTFIX\_INCR\_REDUCE}$$

$$\frac{e\,\textbf{is an abstract left value} \quad addr = freshAddress()}{\rho \vdash e++ \rightarrow \textbf{LVal}\,(addr) = e, e = operator+(\textbf{LVal}\,(addr), 1), \textbf{LVal}\,(addr)} \text{ POSTFIX\_INCR\_ELIM}$$

To reduce an expression `e1 += e2`, `e1 -= e2`, `e1 *= e2`, `e1 /= e2`, `e1 %= e2`, `e1 ^= e2`, `e1 &= e2`, `e1 |= e2`, `e1 >>= e2`, or `e1 <<= e2`:

1. If `e1` can be reduced to an abstract left-value, do it
2. Else replace the whole expression by an assignment to `e1` of the result of the corresponding operator, called on `e1` and `e2`

$$\frac{\rho \Vdash e_1 \xrightarrow[S]{E} e_1'}{\rho \vdash e_1 += e_2 \xrightarrow[S]{E} e_1' += e_2} \text{ PLUS\_EQUAL\_REDUCE}$$

$$\frac{e_1\,\textbf{is an abstract left value}}{\rho \vdash e_1 += e_2 \rightarrow e_1 = operator+(e_1, e_2)} \text{ PLUS\_EQUAL\_ELIM}$$

6.3.9 Calls

Overloaded function calls have already been resolved to point to a specific function declaration during the validation phase.

Like we added `Loop` or `JoinExpr`, we add a special construct `Call` that takes as argument a statement and return an expression. Informally, it is a way to transform a return statement into the corresponding value.

To reduce a function call by one step:

1. If there is at least an argument that can be reduced, reduce the left-most argument that can be reduced.
2. Else:
   (a) ASSERT(the number of arguments and parameters to the function match)
   (b) Create a new environment from the current environment
   (c) For each parameter of the function, from left to right:
      i. Lookup the address of that parameter
      ii. Emit a store of the value of the corresponding argument to that address, of a size appropriate to the type of it. That store is po-after any other store emitted by this step for previous parameters.
      iii. Modify the new environment to have a binding from that parameter name to that address
   (d) Make a block statement from the body of the function, annotated with this new environment
   (e) Wrap that block in the `Call` construct
   (f) Replace the entire expression by that construct.

Note: Contrary to C/C++, execution order is fully specified: it is always left-to-right.

Note: The new environment binds the parameter names to the argument values, regardless of whether there was already a binding for that name. This allows shadowing global variables.

$$\frac{\rho \vdash e \xrightarrow[S]{E} e'}{\rho \vdash \mathit{fid}(rv'_0, \,..\,, rv'_m, e, e_0, \,..\,, e_k) \xrightarrow[S]{E} \mathit{fid}(rv'_0, \,..\,, rv'_m, e', e_0, \,..\,, e_k)} \quad \text{CALL\_REDUCE}$$

$$\frac{\begin{array}{l} \mathit{fid} \mapsto (y_0 : addr_0, \,..\,, y_m : addr_m)\{s_0 \,..\, s_k\} \\ E = \textbf{Sequence}\,(addr_0 \leftarrow rv'_0, \,..\,, addr_m \leftarrow rv'_m) \\ \rho' = \rho[y_0 \mapsto \textbf{LVal}\,(addr_0), \,..\,, y_m \mapsto \textbf{LVal}\,(addr_m)] \end{array}}{\rho \vdash \mathit{fid}(rv'_0, \,..\,, rv'_m) \xrightarrow{E} \textbf{Call}\,\{_{\rho'}\, s_0 \,..\, s_k\}} \quad \text{CALL\_RESOLVE}$$

To reduce a `Call` construct by one step:

1. If its argument can be reduced, reduce it
2. Else if its argument is `return;` or an empty block, replace it by a special `Void` value. Nothing can be done with such a value, except discarding it (see Effectful Expression).
3. Else if its argument is `return val;` for some value `val`, then replace it by this value.

$$\frac{\rho \vdash s \xrightarrow[S]{E} s'}{\rho \vdash \textbf{Call}\, s \xrightarrow[S]{E} \textbf{Call}\, s'} \quad \text{CALL\_CONSTRUCT\_REDUCE}$$

$$\frac{}{\rho \vdash \textbf{Call}\, \textbf{return}\, ; \to \textbf{Void}} \quad \text{CALL\_RETURN\_VOID}$$

$$\frac{}{\rho \vdash \textbf{Call}\, \{_{\rho'}\ \} \to \textbf{Void}} \quad \text{CALL\_END\_FUNCTION}$$

$$\frac{}{\rho \vdash \textbf{Call}\, \textbf{return}\, rval; \to rval} \quad \text{CALL\_RETURN}$$

6.3.10 Other

Parentheses have no effect at runtime (beyond their effect during parsing).

The comma operator simply reduces its first operand as long as it can, then drops it and is replaced by its second operand.

$$\frac{\rho \vdash e_0 \xrightarrow[S]{E} e_0'}{\rho \vdash e_0, e_1 \xrightarrow[S]{E} e_0', e_1} \quad \text{COMMA\_REDUCE}$$

$$\frac{}{\rho \vdash rval, e_1 \to e_1} \quad \text{COMMA\_NEXT}$$

$$\frac{}{\rho \vdash (e) \to e} \quad \text{PARENS\_EXEC}$$

6.4 Generated functions

We saw in the validation section that many functions can be automatically generated:

- address-takers for each field of each struct
- indexed address-takers for each array type
- (indexed) getters and setters for each (indexed) address-taker in the thread address space

In this section we will describe how they behave at runtime.

For each field `foo` with type `T` of a struct `Bar`, 4 address-takers are generated, one for each address-space. Each of them return a pointer to an address that is the sum of the address of their parameter and the offset required to hit the corresponding field.

Note: We describe these functions in this way, because they are not writable directly in the language.

For each type of the form `T[n]` which is used in the program, the following declarations are generated:

```
thread T* operator&[](thread T[n]* a, uint32 i) { return &((@a)[i]); }
threadgroup T* operator&[](threadgroup T[n]* a, uint32 i) { return &((@a)[i]); }
device T* operator&[](device T[n]* a, uint32 i) { return &((@a)[i]); }
constant T* operator&[](constant T[n]* a, uint32 i) { return &((@a)[i]); }
```

For each declaration of the form `address-space T* operator&.foo(thread Bar* b)` for some `address-space`, the following declarations are generated:

```
T operator.foo(Bar b) { return b.foo; }
Bar operator.foo=(Bar b, T newval) { b.foo = newval; return b; }
```

Note: The `b.foo` part in both of the above uses the address-taker, as `b` is a function parameter and thus a left value

For each declaration of the form `address-space T2* operator&[](thread T1* a, uint32 i)` for some `address-space`, the following declarations are generated:

```
T2 operator[](T1 a, uint32 i) { return a[i]; }
T1 operator[]=(T1 a, uint32 i, T2 newval) { a[i] = newval; return a; }
```

Note: Similarily, `a[i]` in both of the above use the indexed address-taker, as `a` is a function parameter, and thus a left-value. Such generated getters and setters may look useless, but they are used when something is not a left-value, for example because of nested calls to getters/setters. For example you could have a struct Foo, with a getter for the field bar, returning a struct Bar, with an ander for the field baz. When using foo.bar.baz, it is not possible to use the ander for Bar, as foo.bar is not a left-value. So we instead use the generated getter (that behind the scene copies foo.bar into its parameter, and then uses the ander).

6.5 Memory model

Our memory model is strongly inspired by the Vulkan memory model, as presented in https://github.com/KhronosGroup/Vulkan-MemoryModel/blob/master/alloy/spirv.als as of the git commit f9110270e1799041bdaaf00a1db70fd4175d433f and in https://github.com/KhronosGroup/Vulkan-Docs/blob/master/appendices/memorymodel.txt as of the git commit 56e0289318a4cd23aa5f5dcfb290ee873be53b82. That memory model is under Creative Commons Attribution 4.0 International License per the comment at the top of both files: http://creativecommons.org/licenses/by/4.0/ and is Copyright (c) 2017-2018 Khronos Group or Copyright (c) 2017-2019 Khronos Group depending on the file.

The main difference between the two models is that we avoid undefined behaviour by making races merely make reads return unspecified results. This is in turn safe, as our execution semantics for loads (see above) clamp any enum value to a valid value of that type, and there can be no race on pointers or array references as they are limited to the `thread` address space.

Apart from that, we only removed parts of the model, since some operations supported by Vulkhan are not supported by WHLSL, and renamed some elements for consistency with the rest of this specification.

6.5.1 Memory locations

A memory location identifies unique storage for 8 bits of data. Memory operations access a set of memory locations consisting of one or more memory locations at a time, e.g. an operation accessing a 32-bit integer in memory would read/write a set of four memory locations. Two sets of memory locations overlap if the intersection of their sets of memory locations is non-empty. A memory operation must not affect memory at a memory location not within its set of memory locations.

6.5.2 Memory events and program order

Some steps in the execution rules provided in the previous section emit memory events. There are a few possible such events: - A store of a value to some set of (contiguous) memory locations, that may be atomic - A load of a value from some set of (contiguous) memory locations, that may be atomic - A memory barrier (a.k.a. memory fence), that may either ensure synchronization at the threadgroup or whole device scope. - A control barrier, with a scope that may be threadgroup or device, that may optionally act as a memory barrier at the threadgroup level, or at the device level

Note: A write whose value is the same as what was already in those memory locations is still considered to be a write and has all the same effects.

Todo: Add a note here giving an informal mapping of these to Vulkan/MSL/HLSL.

There is furthermore a total order `po` (program order) on all such events by any given thread. An event is before another by `po` if it is emitted by an execution rule that is executed by this thread before the rule that emitted the other event. Additionally the store events emitted by the call execution rule are ordered by `po` in the order of the corresponding parameters (as written in that rule).

Note: `po` is guaranteed to be a total order for a given thread because the call rule is the only one that emits several memory events.

Todo: Rewrite the rest of the model here, translating the kinds of atomics provided; and formalizing what we mean about races.

CHAPTER

SEVEN

STANDARD LIBRARY

7.1 Built-in Types

7.1.1 Built-in Scalars

Type Name	Description	Representable values
void	Must only be used as a return type from functions which don't return anything.	None
bool	A conditional type.	true or false
uint	An unsigned 32-bit integer.	0, 1, 2, … 4294967295
int	A signed 32-bit integer.	-2147483648, -2147483647, … -1, 0, 1, … 2147483647
float	An IEEE 32-bit floating-point number.	All values of a IEEE 754 single-precision binary floating-point number

Note: The following types are not present in WHLSL: dword, min16float, min10float, min16int, min12int, min16uint, string, size_t, ptrdiff_t, double, float64, int64, uint64

7.1.2 Built-in Atomic Types

1. atomic_int
2. atomic_uint

7.1.3 Built-in aggregate types

The following are vector types, which list the name of a scalar type and the number of elements in the vector. Each item below includes two types, which are synonyms for each other.

- bool2, or vector<bool, 2>
- bool3, or vector<bool, 3>
- bool4, or vector<bool, 4>
- uint2, or vector<uint, 2>
- uint3, or vector<uint, 3>

- uint4, or vector<uint, 4>
- int2, or vector<int, 2>
- int3, or vector<int, 3>
- int4, or vector<int, 4>
- float2, or vector<float, 2>
- float3, or vector<float, 3>
- float4, or vector<float, 4>

The following are matrix types, which list the name of a scalar type, the number of columns, and the number of rows, in that order. Each item below includes two types, which are synonyms for each other.

- float2x2, or matrix<float, 2, 2>
- float2x3, or matrix<float, 2, 3>
- float2x4, or matrix<float, 2, 4>
- float3x2, or matrix<float, 3, 2>
- float3x3, or matrix<float, 3, 3>
- float3x4, or matrix<float, 3, 4>
- float4x2, or matrix<float, 4, 2>
- float4x3, or matrix<float, 4, 3>
- float4x4, or matrix<float, 4, 4>

Todo: Should we have int or bool matrices?

7.1.4 Samplers

Samplers must only be passed into an entry point inside an argument. All samplers are immutable and must be declared in the "constant" address space. There is no constructor for samplers; it is impossible to create or destory one in WHLSL. The type is defined as `native typedef sampler;`. Samplers are impossible to introspect. Arrays must not contain samplers anywhere inside them. Functions that return samplers must only have one return point. Ternary expressions must not return references.

Todo: The last sentence does not seem related to samplers. Or should we s/references/samplers/g in it? https://github.com/gpuweb/WHLSL/issues/332

Todo: Robin: I have not put the `native typedef` syntax in the grammar or the semantics so far, should I? https://github.com/gpuweb/WHLSL/issues/332

7.1.5 Textures

The following types represent textures:

- Texture1D<T>

- RWTexture1D<T>
- Texture1DArray<T>
- RWTexture1DArray<T>
- Texture2D<T>
- RWTexture2D<T>
- Texture2DArray<T>
- RWTexture2DArray<T>
- Texture3D<T>
- RWTexture3D<T>
- TextureCube<T>
- TextureDepth2D<float>
- RWTextureDepth2D<float>
- TextureDepth2DArray<float>
- RWTextureDepth2DArray<float>
- TextureDepthCube<float>

Todo: Texture2DMS<T>, TextureDepth2DMS<float> https://github.com/gpuweb/WHLSL/issues/333

Each of the above types accepts a "type argument". The "T" types above may be any scalar or vector integral or floating point type.

If the type argument, including the <> characters is missing, is is assumed to be `float4`.

Textures must only be passed into an entry point inside an argument. Therefore, textures must only be declared in either the `constant` or `device` address space. A texture declared in the `constant` address space must never be modified. There is no constructor for textures; it is impossible to create or destroy one in WHLSL. Arrays must not contain textures anywhere inside them. Functions that return textures must only have one return point. Ternary expressions must not return references.

Todo: "Therefore": it is not clear to me how it is a consequence (Robin). "Ternary expressions must not return references": What kind of references are you referring to? If it is array references, I don't think it is the right place to mention this (but thank you for the reminder to put it in the validation section). Similarily, most of these constraints should probably be either duplicated in or moved to the validation section, I will take care of it.

They are not copyable, ie they are references. https://github.com/gpuweb/WHLSL/issues/334

7.2 Built-in Variables

Built-in variables are represented by using semantics. For example, `uint theInstanceID : SV_InstanceID`. Variables with these semantics must have the type associated with that semantic.

The following built-in variables, as identified by their semantics, are available inside arguments to vertex shaders:

Semantic Name	Type
SV_InstanceID	uint
SV_VertexID	uint

The following built-in variables, as identified by their semantics, are available inside the return value of a vertex shader:

Semantic Name	Type
PSIZE	float
SV_Position	float4

The following built-in variables, as identified by their semantics, are available inside arguments to fragment shaders:

Semantic Name	Type
SV_IsFrontFace	bool
SV_SampleIndex	uint
SV_InnerCoverage	uint

The following built-in variables, as identified by their semantics, are available inside the return value of a fragment shader:

Semantic Name	Type
SV_Target[n]	float4
SV_Depth	float
SV_Coverage	uint

The following built-in variables, as identified by their semantics, are available inside arguments to compute shaders:

Semantic Name	Type
SV_DispatchThreadID	uint3
SV_GroupID	uint3
SV_GroupIndex	uint
SV_GroupThreadID	uint3

7.3 Built-in Functions

Todo: Fill in this section, including all of the basic arithmetic operators, explaining what behaviors are allowed on overflow.

Some of these functions only appear in specific shader stages.

We should figure out if atomic handling goes here.

7.3.1 Integer arithmetic

Note: Many of the functions described in this section have other overloads for floating point arguments, or for vector/matrix arguments. These overloads will be described in the corresponding sections.

`operator+`, `operator-` and `operator*` are defined as binary functions on integers, both signed and unsigned. Their return type is the same as the type of their arguments, and they respectively implement integer addition, substraction and multiplication. They follow 2-complement semantics in the case of overflow or underflow. In other terms, they behave as if they computed the result as integers with a large enough width for avoiding both overflow and underflow, then truncated to the 32 low-bits of the result. `mul` is defined on integers both signed and unsigned as a synonym of operator*.

`operator++` and `operator--` are defined as unary functions on integers, both signed and unsigned. Their return type is the same as the type of their argument. They return the addition or substraction (respectively) of `1` to their argument, behaving like `operator+` and `operator-` in the case of overflow/underflow.

Todo: Decide whether we want to have operator++ at all, or whether it should just be sugar for +1. We are currently not consistent, not only between spec and implementation, but between sections of this spec. https://github.com/gpuweb/WHLSL/issues/336

`operator/` and `operator%` are defined as binary functions on integers, both signed and unsigned. Their return type is the same as the type of their arguments. They respectively return the algebraic quotient truncated towards zero (fractional part discarded), and the remainder of the division such that `(a/b)*b + a%b == a`. If the second operand is `0`, both return an unspecified value. If there is an underflow (e.g. from `INT_MIN / (-1)`), they follow 2-complement semantics. In other terms, they behave as if they computed the result as integers with a large enough width for avoiding both overflow and underflow, then truncated to the 32 low-bits of the result.

`abs` is defined as an unary function on integers, both signed and unsigned. Its return type is the same as its argument type. It returns its argument if it is non-negative, and the opposite of its argument otherwise. In case of an underflow/overflow it follows 2-complement semantics, so abs(INT_MIN) is INT_MIN.

`sign` is defined as an unary function on integers, both signed and unsigned. Its return type is int. If its argument is positive, it returns `1`. If its argument is negative, it returns `-1`. If its argument is `0`, it returns `0`.

Todo: In the current implementation, its return type is always the same as its argument type. https://bugs.webkit.org/show_bug.cgi?id=200252

`min` and `max` are defined as binary functions on integers, both signed and unsigned. Their return type is the same as the type of their arguments. They return respectively the minimum and the maximum of their arguments.

`clamp` is defined as a ternary function on integers, both signed and unsigned. Its return type is the same as the type of its arguments. It returns `min(max(first, second), third)`. In particular, if its third argument is less than its second argument it will always return its third argument.

Note: This behavior for `clamp` matches that of HLSL (where it is undocumented).

Note: In the HLSL documentation, `abs`, `sign`, `min`, `max` and `clamp` are not defined on unsigned integers. But we found them defined on unsigned integers in the actual implementation, so they are supported in WHLSL on unsigned integers for portability.

7.3.2 Bit manipulation

`operator&`, `operator|` and `operator^` are defined as binary functions on booleans, signed integers, and unsigned integers. Their return type is the same type as their arguments. They respectively implement bitwise and, or and exclusive or; treating booleans as if they were integers of size 1, with `true` being `1` and `false` being `0`.

`operator~` is defined as an unary function on booleans, signed integers, and unsigned integers. Its return type is the same type as its argument. It implements bitwise negation on integers, and negation on booleans.

`operator<<` and `operator>>` are defined as binary functions, whose first argument can be integers either signed or unsigned, and whose second argument is an unsigned integer. Their return types are the same type as their first argument. They respectively shift their first argument left/right by their second argument modulo the bit-width of their first argument. So for example `x << 33` is the same as `x << 1` if x is a 32-bit integer. In the case of a right shift on an unsigned integer or any left shift, the vacated bits are replaced by 0. In the case of a right shift on a signed integer, the vacated bits are replaced by the sign bit of the first argument (i.e. it is an arithmetic right-shift, not a logical one).

`all` and `any` are defined as unary functions on booleans and integers both signed and unsigned. Their return type is always `bool`. They return false on zero and true on any non-zero integer. They are simply the identity function on booleans.

`countbits`, `reversebits`, `firstbithigh` and `firstbitlow` are all unary functions on unsigned integers. Their return type is also `unsigned int`. `countbits` returns the number of bits set to 1 in the binary representation of its argument. `reversebits` reverses the order of the bits in the binary representation of its argument. `firsbithigh` and `firstbitlow` return `32` if their argument is `0`. Otherwise they return the index of respectively the lowest and the highest bit that is set in the binary representation of their argument.

7.3.3 Floating point arithmetic

Also includes a bunch of special functions like cos, isNaN, . . .

7.3.4 Numerical Compliance

Todo: Decide on what precision guarantees we can make. MSL got two different tables, depending on whether it is running in fast-math mode or not. I did not find the equivalent table for SPIR-V, but it has the nice property of tagging each operation with the different components of fast-math. We should probably measure how costly forbidding fast-math would be, since NotNaN and NotInf introduce undefined behavior. https://github.com/gpuweb/WHLSL/issues/335

7.3.5 Fences and atomic operations

7.3.6 Cast operators

Also include the `as` function.

7.3.7 Comparison operators

7.3.8 Vector and matrix operations

Includes arithmetic operators (per-element), various swizzles, length, getDimensions, operator[] and operator[]=, determinant

7.3.9 Sampler and texture operations

Sample, Load, Gather, etc..

CHAPTER

EIGHT

INTERFACE WITH JAVASCRIPT

Shaders are supplied to the Javascript WebGPU API as a single argument which is understood to be of type 'DOM-String'.

CHAPTER

NINE

RESOURCE LIMITS

1. How many inputs
2. How many outputs
3. How many intermediate variables

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search