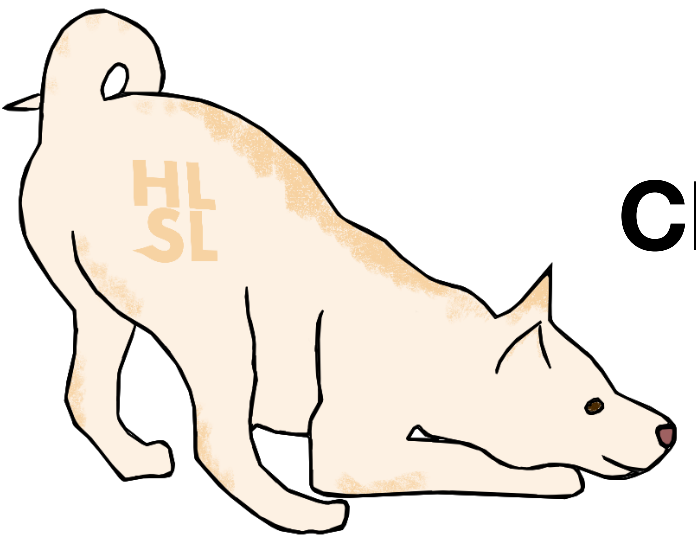
Two Compilers, One Language, No Specification

Defining a Language from Disagreeing References



Chris Bieneman

What is HLSL?

- GPU programming language introduced in 2002
- Evolved from NVIDIA's C for Graphics (Cg)
- Mostly used for computer graphics
  - Also used in GPGPU and ML applications

State of HLSL

- Reference Compiler #1 - FXC
  - Out of active support
  - DirectX 9-11 (and early 12)
  - HLSL versions 2002->2015
- Reference Compiler #2 - DXC
  - Fork of LLVM 3.7
  - DirectX 12 Ultimate
  - HLSL Versions 2016, 2017, 2018, 2021 & 202x
  - 202x & critical support only

The Problem

- We need a modern HLSL compiler
- We have a large base of existing HLSL code
- We have a lot of users that need to transition
- We don't want a bug-for-bug compatible replacement
    - GPU code portability is a common problem
- We need a written specification!

Why do we need a spec?

```
static int Count;

void increment(inout int C) {
  C += 1;
  C += Count;
}

int Fn() {
  Count = 0;
  increment(Count);
  return Count;
}
```

Same Compiler, Different Results

DirectX stores 1, SPIR-V stores 2

COMPILER EXPLORER
Add...
More
Templates
Sponsors
Share
Policies
Other
HLSL source #1
DXC (trunk) (Editor #1)
Output...
Filter...
Libraries
Overrides
Add new...
Add tool...

HLSL

```
static int Count;

void increment(inout int C) {
  C += 1;
  C += Count;
}

int Fn() {
  Count = 0;
  increment(Count);
  return Count;
}

RWBuffer<int> Out;
[numthreads(1,1,1)]
void main() {
  Out[0] = Fn();
}
```

DXC (trunk) -T cs_6_6

```
%dx.types.Handle = type { i8* }
%dx.types.ResBind = type { i32, i32, i32, i8 }
%dx.types.ResourceProperties = type { i32, i32 }
%"class.RWBuffer<int>" = type { i32 }

define void @main() {
  %Out_UAV_buf = call %dx.types.Handle @dx.op.createHandleFromBi
  call void @llvm.dbg.value(metadata i32 1, i64 0, metadata !42,
  %1 = call %dx.types.Handle @dx.op.annotateHandle(i32 216, %dx.
  call void @dx.op.bufferStore.i32(i32 69, %dx.types.Handle %1,
  ret void, !dbg !46
}

declare void @llvm.dbg.value(metadata, i64, metadata, metadata)

declare void @dx.op.bufferStore.i32(i32, %dx.types.Handle, i32,

declare %dx.types.Handle @dx.op.annotateHandle(i32, %dx.types.Ha

declare %dx.types.Handle @dx.op.createHandleFromBinding(i32, %dx

attributes #0 = { nounwind readnone }
attributes #1 = { nounwind }
```

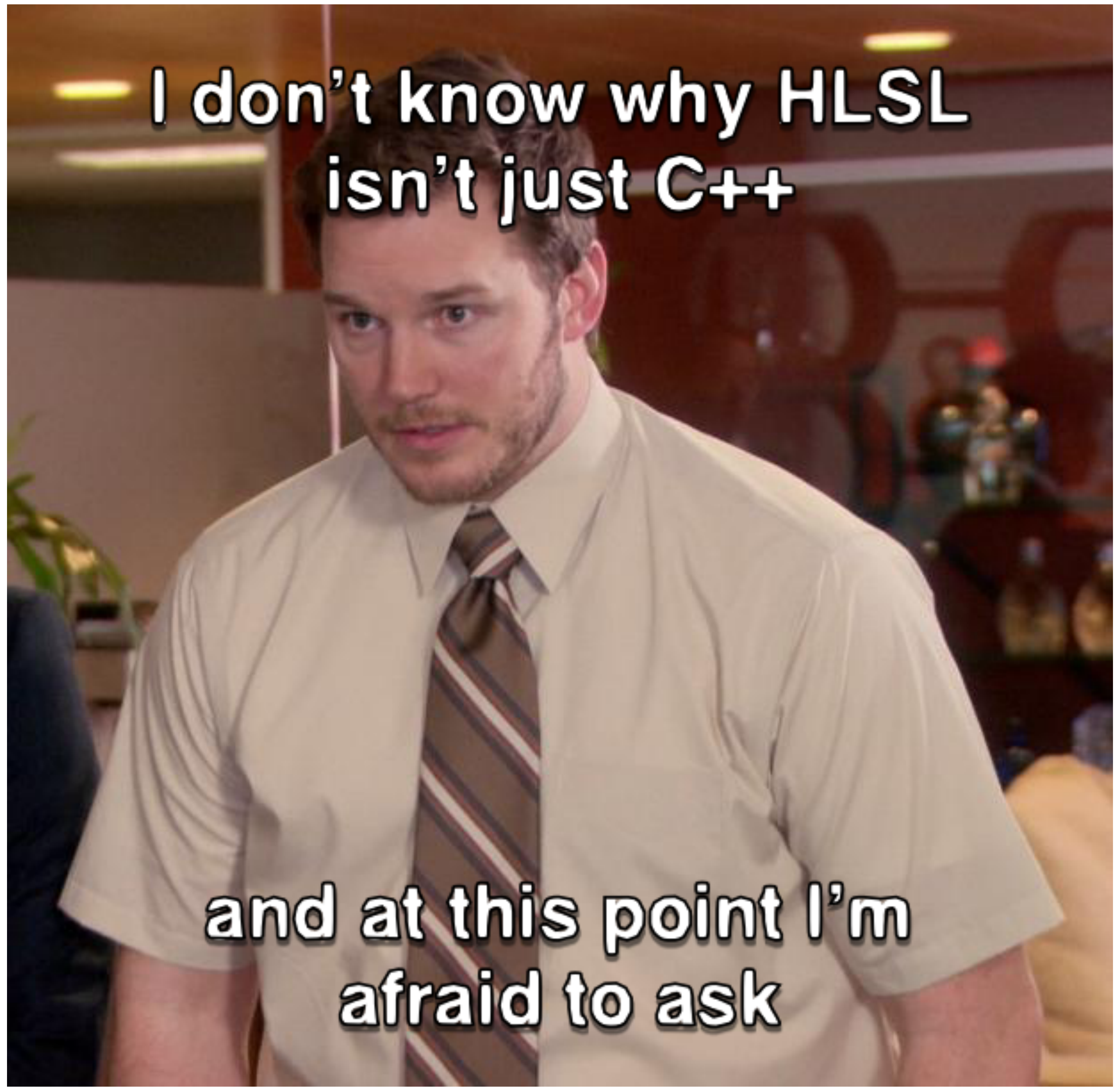
DXC (trunk) -T cs_6_6 -spirv

```
                   OpCapability SampledBuffer
                   OpCapability ImageBuffer
                   OpMemoryModel Logical GLSL450
                   OpEntryPoint GLCompute %main "main"
                   OpExecutionMode %main LocalSize 1 1 1
                   OpDecorate %Out DescriptorSet 0
                   OpDecorate %Out Binding 0
            %int = OpTypeInt 32 1
           %uint = OpTypeInt 32 0
         %uint_0 = OpConstant %uint 0
%type_buffer_image = OpTypeImage %int Buffer 2 0 0 2 R32i
%_ptr_UniformConstant_type_buffer_image = OpTypePointer Uniform(
           %void = OpTypeVoid
             %10 = OpTypeFunction %void
            %Out = OpVariable %_ptr_UniformConstant_type_buffer_imag
          %int_2 = OpConstant %int 2
           %main = OpFunction %void None %10
             %12 = OpLabel
             %13 = OpLoad %type_buffer_image %Out
                   OpImageWrite %13 %uint_0 %int_2 None
                   OpReturn
                   OpFunctionEnd
```

Why isn't HLSL just C or C++?

TL;DR: (1) History, (2) GPUs are weird

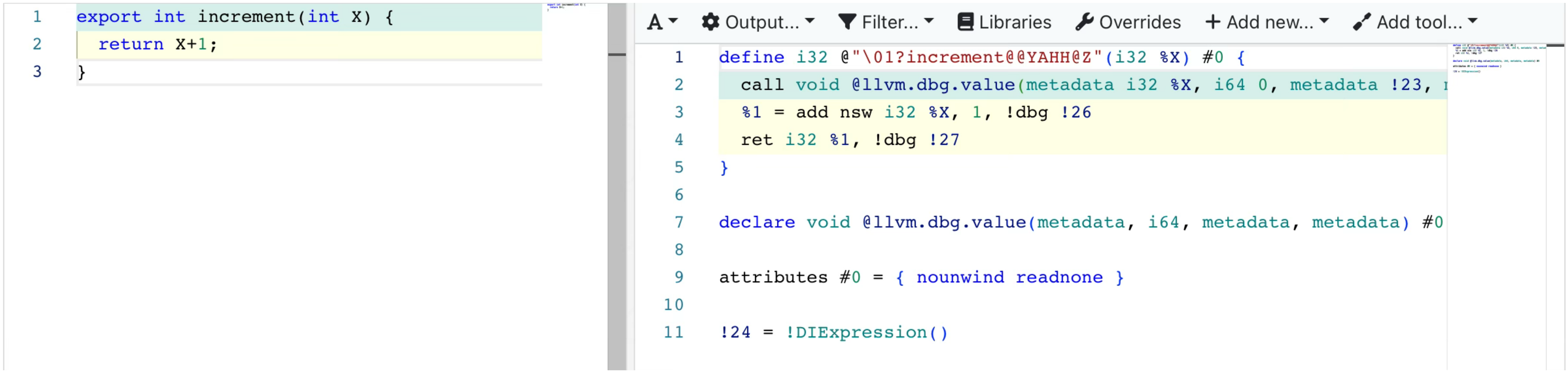
- Early GPUs were very limited
  - No direct memory access
  - No control flow
  - Limited (and strange) data types
- GPU architectures are wildly varied
- GPUs are fundamentally parallel


I don't know why HLSL isn't just C++
and at this point I'm afraid to ask

Similarity: Basic Functions

Reason: HLSL has always been C-like

- HLSL's C/C++ based syntax for functions looks very familiar
- Basic functions just kinda work how you expect
  - HLSL has argument-dependent lookup
- Until they don't…



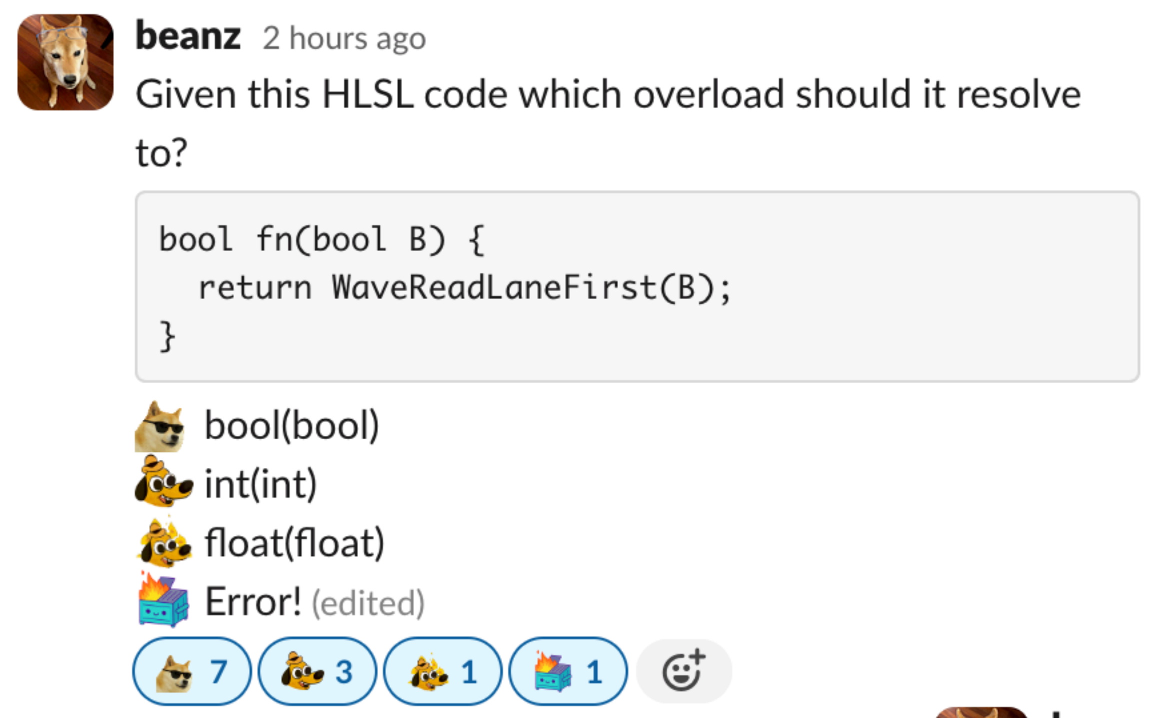
```
export int increment(int X) {
  return X+1;
}
```

```
define i32 @"\01?increment@@YAHH@Z"(i32 %X) #0 {
  call void @llvm.dbg.value(metadata i32 %X, i64 0, metadata !23,
  %1 = add nsw i32 %X, 1, !dbg !26
  ret i32 %1, !dbg !27
}

declare void @llvm.dbg.value(metadata, i64, metadata, metadata) #0

attributes #0 = { nounwind readnone }

!24 = !DIExpression()
```

Difference: Implementation Inconsistency

Reason: Bugs

- DXC is sometimes inconsistent about overload resolution
- built-in functionality has unintended behavior differences

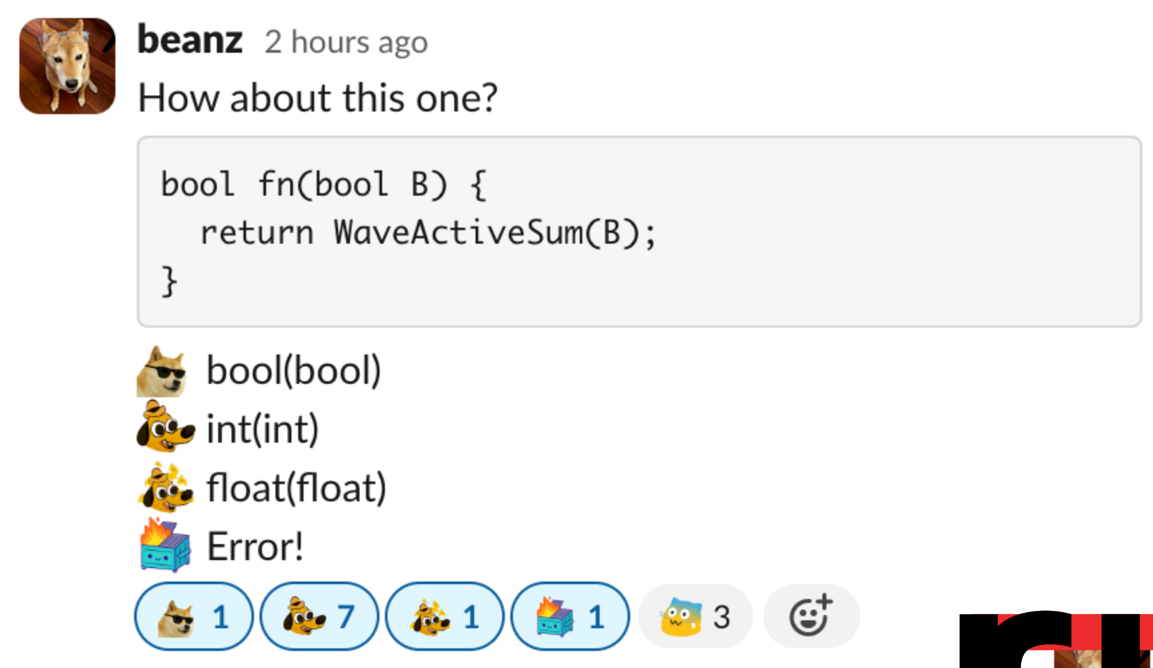


beanz 2 hours ago
Given this HLSL code which overload should it resolve to?

```
bool fn(bool B) {
  return WaveReadLaneFirst(B);
}
```

bool(bool)
int(int)
float(float)
Error! (edited)

7 3 1 1

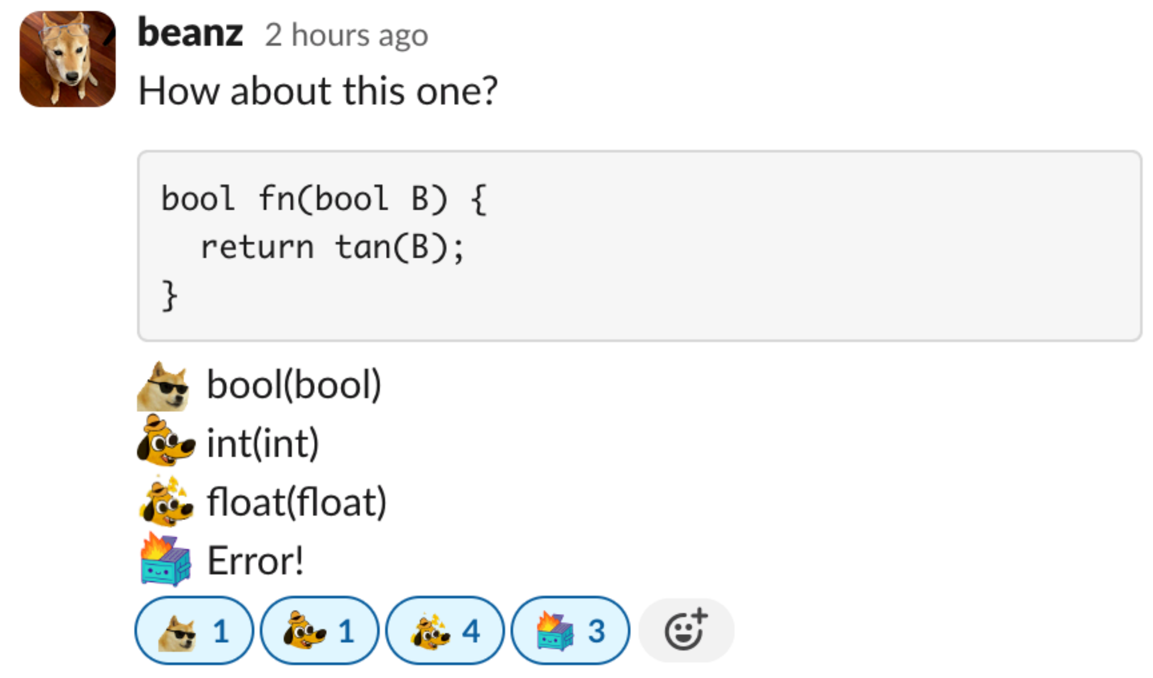


beanz 2 hours ago
How about this one?

```
bool fn(bool B) {
  return WaveActiveSum(B);
}
```

bool(bool)
int(int)
float(float)
Error!

1 7 1 1 3

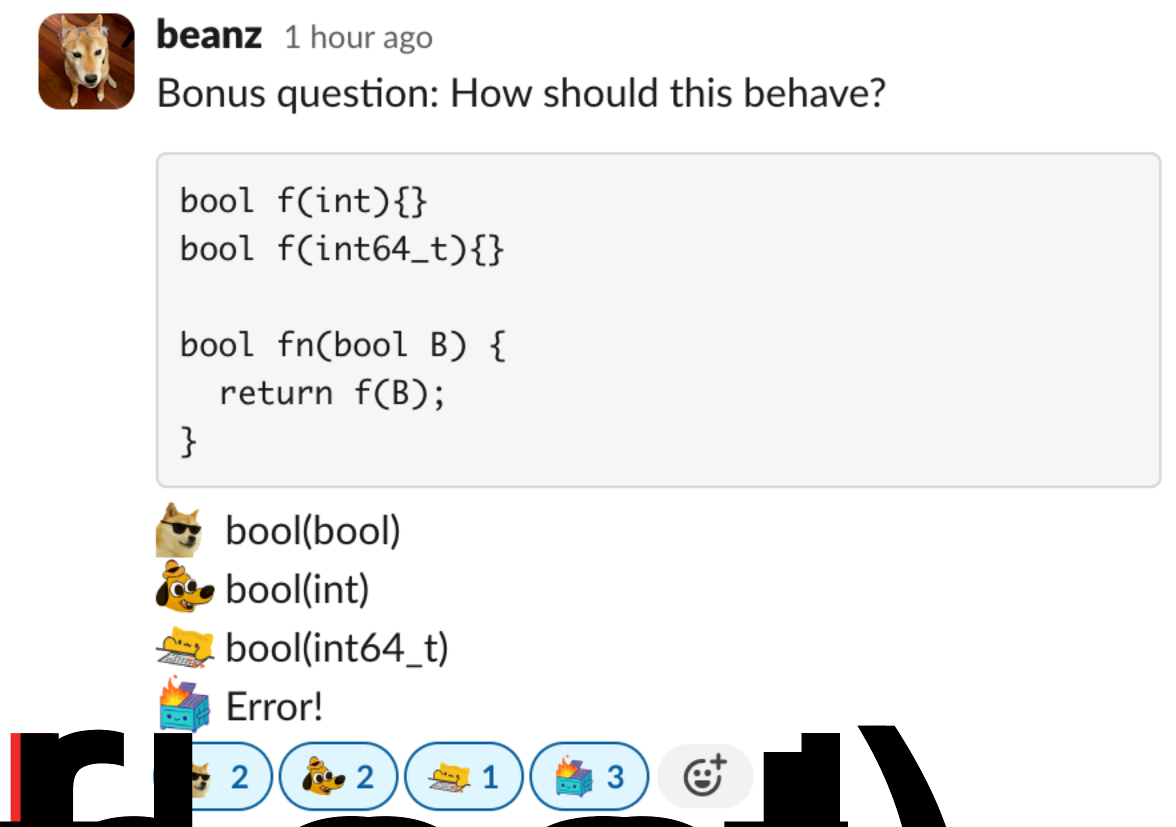


beanz 2 hours ago
How about this one?

```
bool fn(bool B) {
  return tan(B);
}
```

bool(bool)
int(int)
float(float)
Error!

1 1 4 3

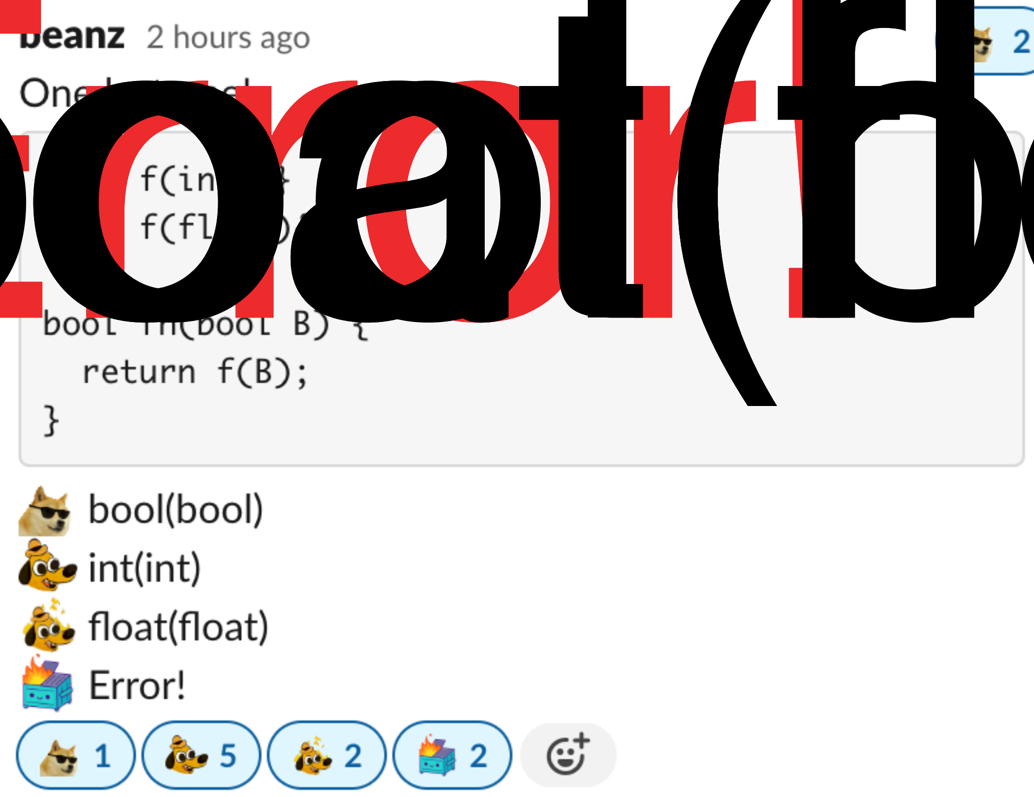


beanz 1 hour ago
Bonus question: How should this behave?

```
bool f(int){}
bool f(int64_t){}

bool fn(bool B) {
  return f(B);
}
```

bool(bool)
bool(int)
bool(int64_t)
Error!

2 2 1 3



beanz 2 hours ago
One

```
f(in
f(fl

bool fn(bool B) {
  return f(B);
}
```

bool(bool)
int(int)
float(float)
Error!

1 5 2 2

float(float)

bool(bool)

Error!

Solution: Implicit hlsl.h

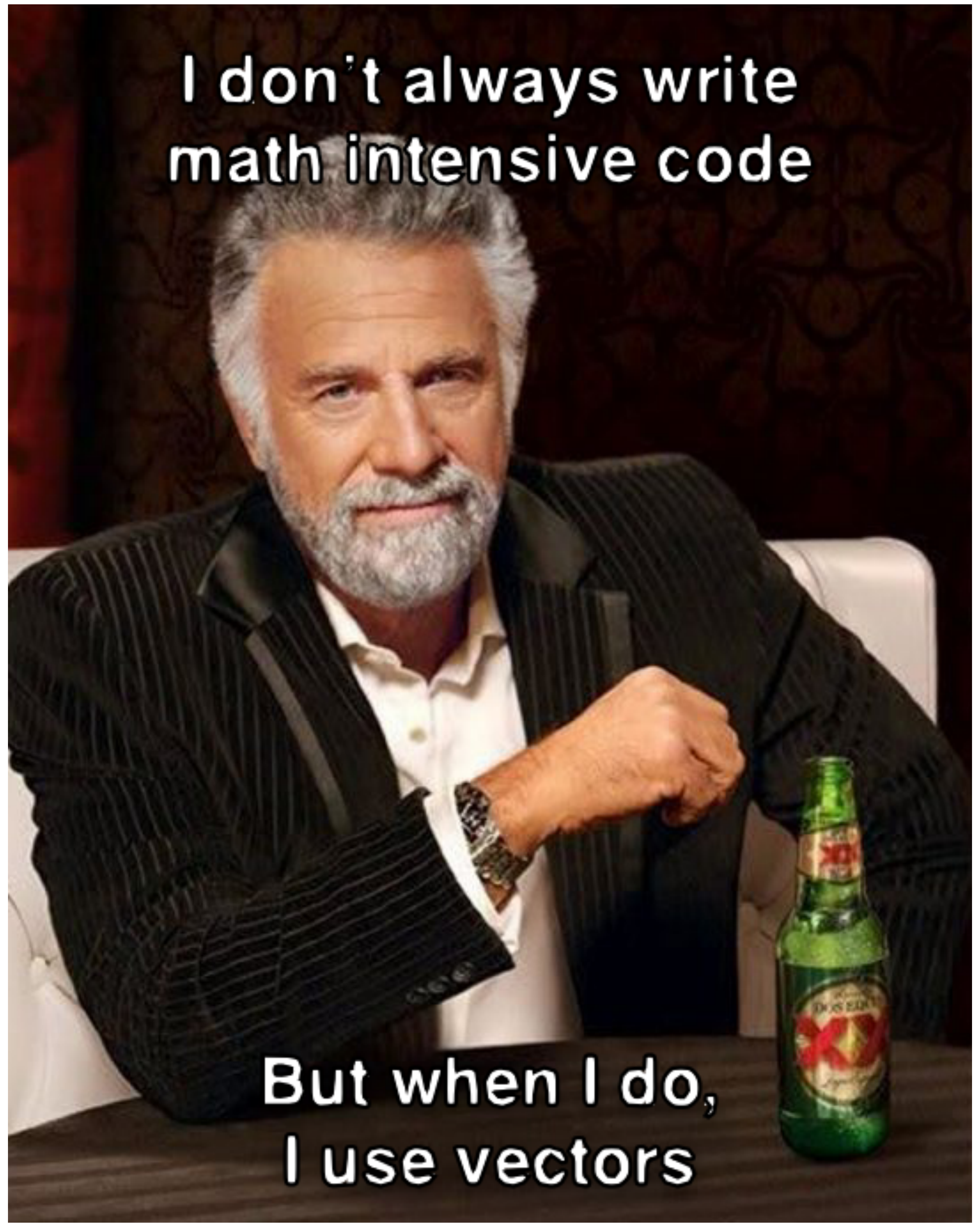
Functions are functions

- Alias function declarations to builtins
- Go-to-definition works for library functions
- Consistent overload resolution behavior
- This will break existing code
    - Socializing this with users

Difference: Vectors all the way down!

Reason: colors, vertices, matrices…

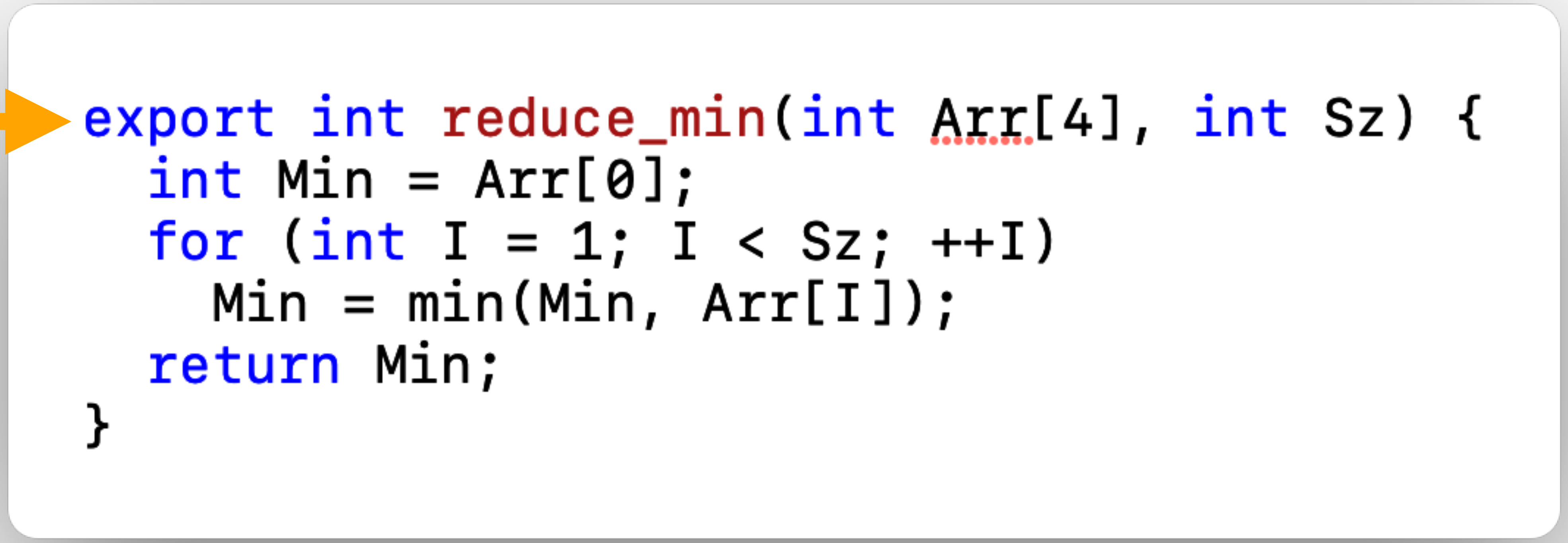
- First class vector types are a requirement
  - Vector types need to work with built-in operators
  - Defined conversion sequences
- Constrain conversions of scalar types… because they're also implicit vectors


I don't always write math intensive code
But when I do, I use vectors

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1		yes
1	{5, 6, 2, 3}	4		yes
2	{9, 6, 0, 0}	2		yes
3	{2, 6, 1, 0}	3		yes



```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	yes
1	{5, 6, 2, 3}	4	5	yes
2	{9, 6, 0, 0}	2	9	yes
3	{2, 6, 1, 0}	3	2	yes

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	5	yes
2	{9, 6, 0, 0}	2	9	yes
3	{2, 6, 1, 0}	3	2	yes

I
1

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	5	yes
2	{9, 6, 0, 0}	2	6	yes
3	{2, 6, 1, 0}	3	2	yes

I
1

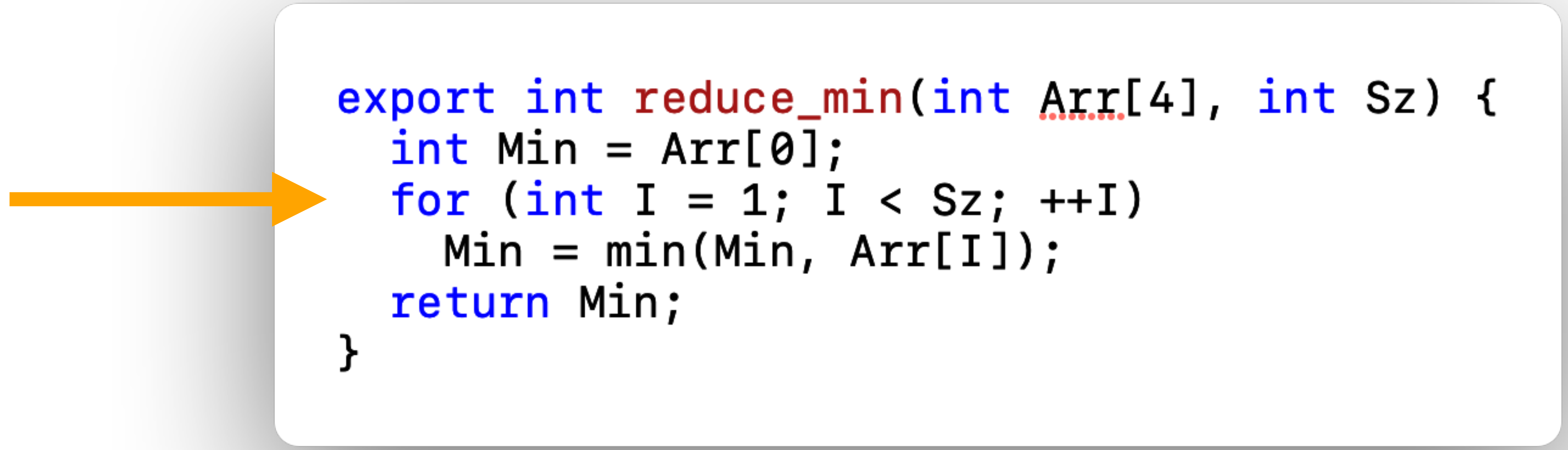
```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	5	yes
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	2	yes

I
2



```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	2	yes
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	1	yes

I
2

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	2	yes
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	1	no

I
3

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	2	yes
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	1	no

I
3

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	2	no
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	1	no

I
4

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

SIMT Execution

Single Instruction Multiple Threads

	Arr	Sz	Min	Executing?
0	{5, 0, 0, 0}	1	5	no
1	{5, 6, 2, 3}	4	2	no
2	{9, 6, 0, 0}	2	6	no
3	{2, 6, 1, 0}	3	1	no

I
4

```
export int reduce_min(int Arr[4], int Sz) {
  int Min = Arr[0];
  for (int I = 1; I < Sz; ++I)
    Min = min(Min, Arr[I]);
  return Min;
}
```

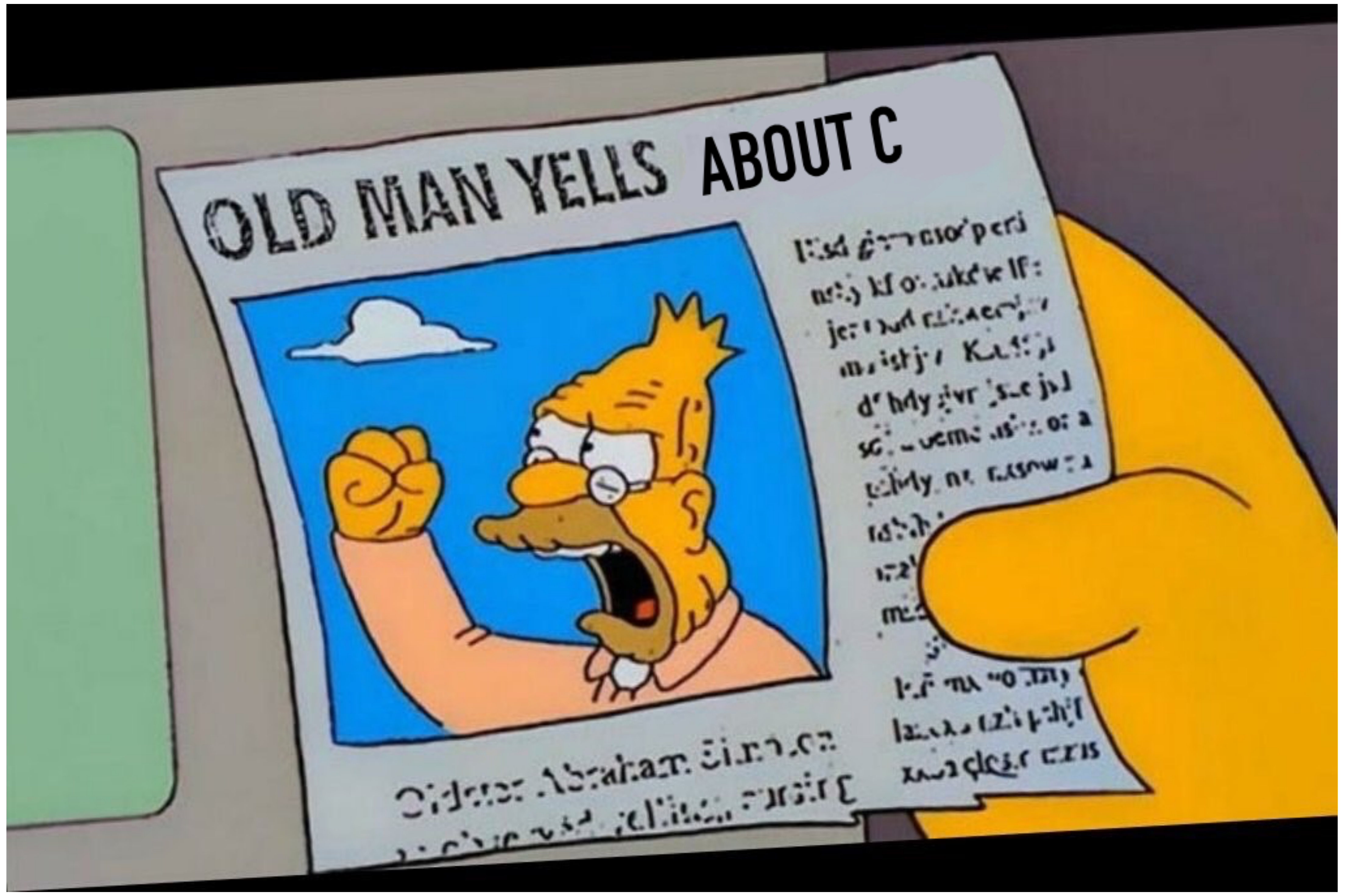
Problem: Hidden Costs

- Each operation happens *n* times
- Each variable is actually *n* variables
- Dynamic control flow is always the worst of threads

Usual Arithmetic Conversions

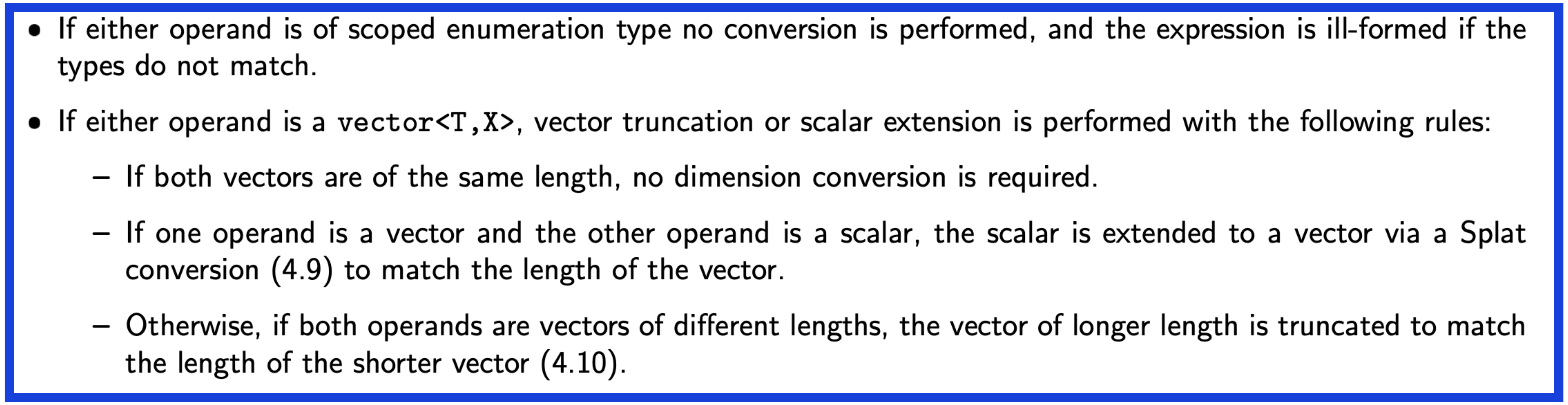
Most hated of C features, worse on vectors

- C implicitly promotes to word-sized values
  - This is extremely expensive on a GPU!
- HLSL also has vectors of arithmetic types
  - We need vector element and dimension conversions!



Solution: Constrain and Extend UAC

- Remove “promotable” sub-word types
- Add vector element and dimension conversions
- Aligns with C language, but considers HLSL’s characteristics



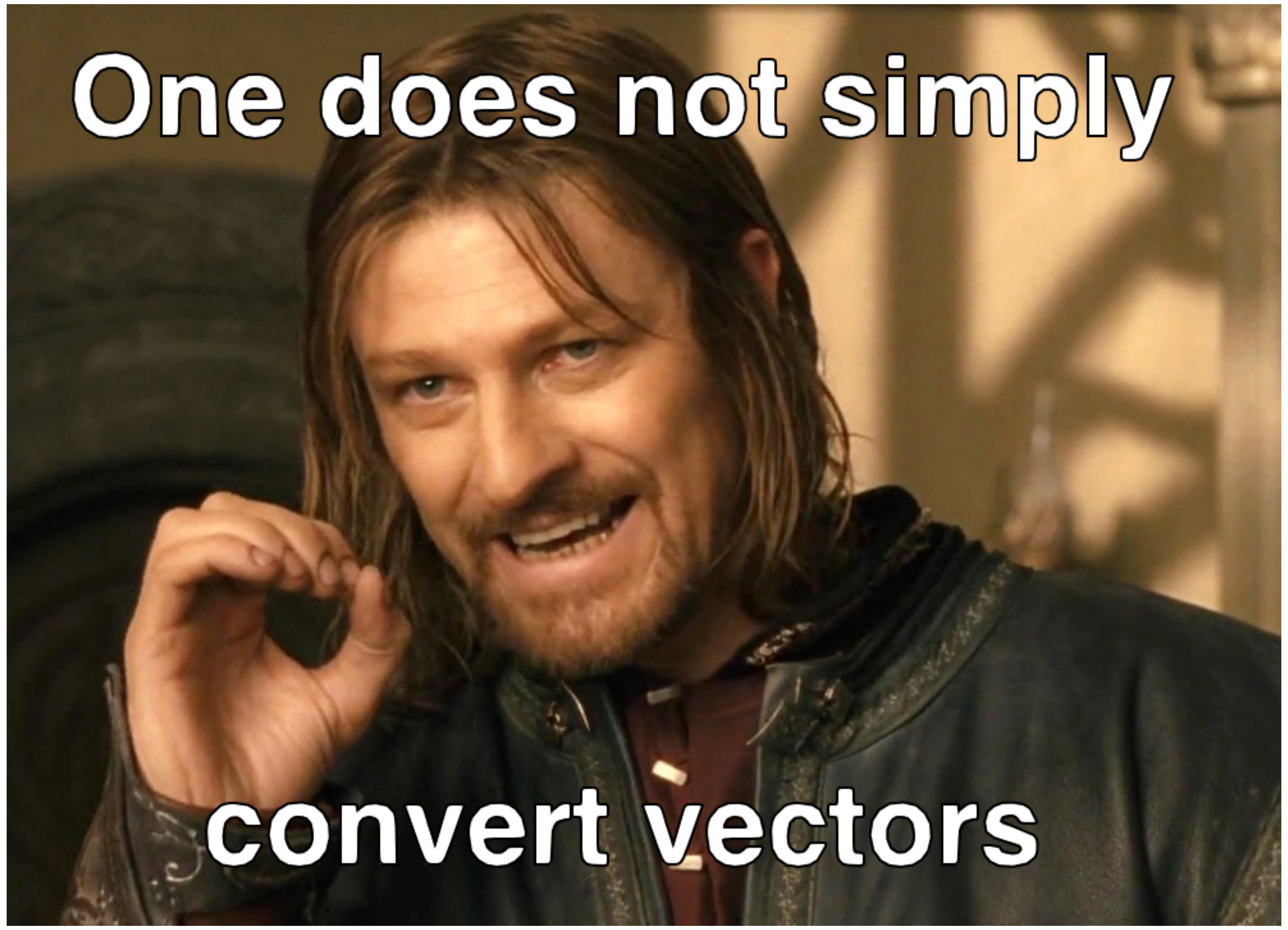
- If either operand is of scoped enumeration type no conversion is performed, and the expression is ill-formed if the types do not match.
- If either operand is a `vector<T,X>`, vector truncation or scalar extension is performed with the following rules:
  - If both vectors are of the same length, no dimension conversion is required.
  - If one operand is a vector and the other operand is a scalar, the scalar is extended to a vector via a Splat conversion (4.9) to match the length of the vector.
  - Otherwise, if both operands are vectors of different lengths, the vector of longer length is truncated to match the length of the shorter vector (4.10).

Wall of Text
Summon Wall
Any poor soul caught within the shadow of the Wall of Text shall never be able to unsee what has been seen.
Illus. Pete Venters
0/7

Difference: Implicit Dimension Conversion

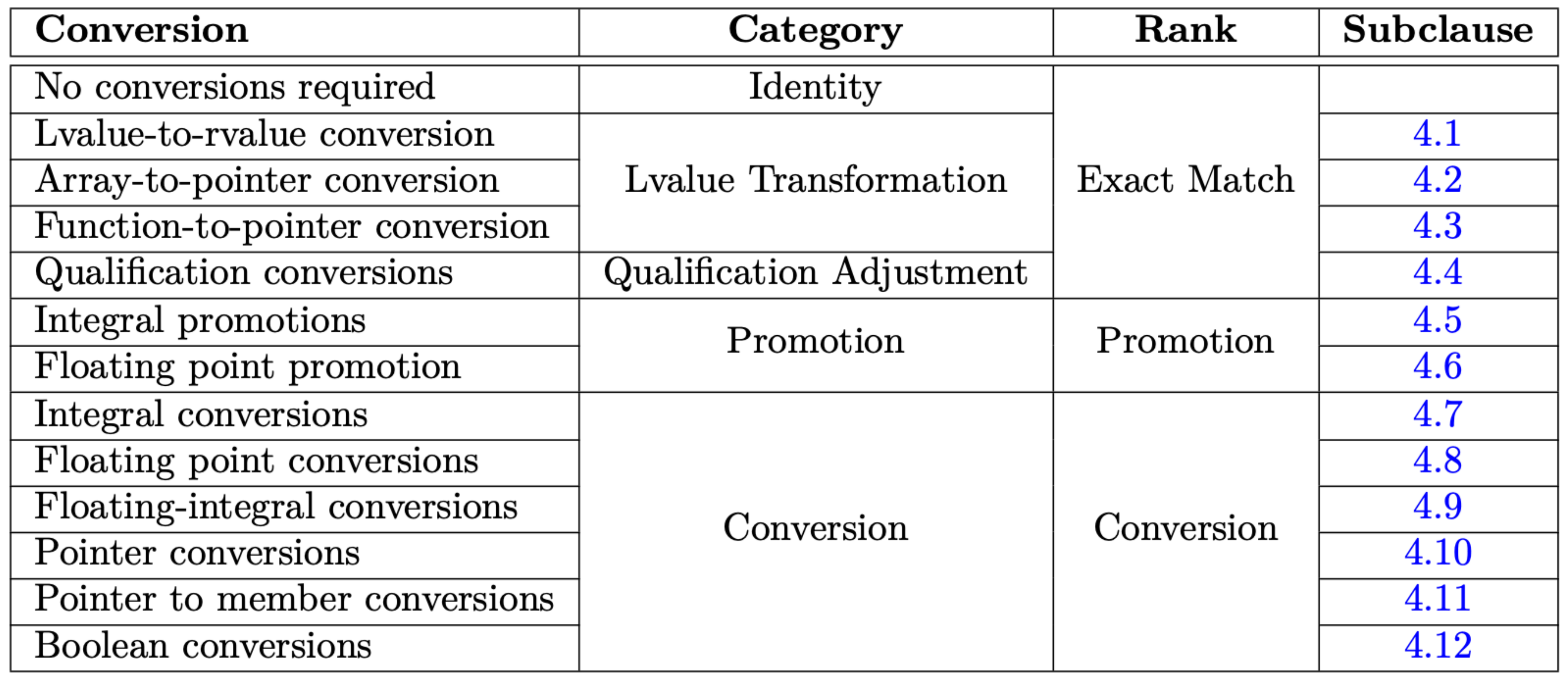
Reason: Lots of vector types!

- C++ Implicit conversion sequences have 3 components
  - Lvalue Transformation
  - Promotion/Conversion
  - Qualification Adjustment
- HLSL adds 4th component for dimension adjustment
  - Extends or truncates dimensionality for scalar, vector or matrix types

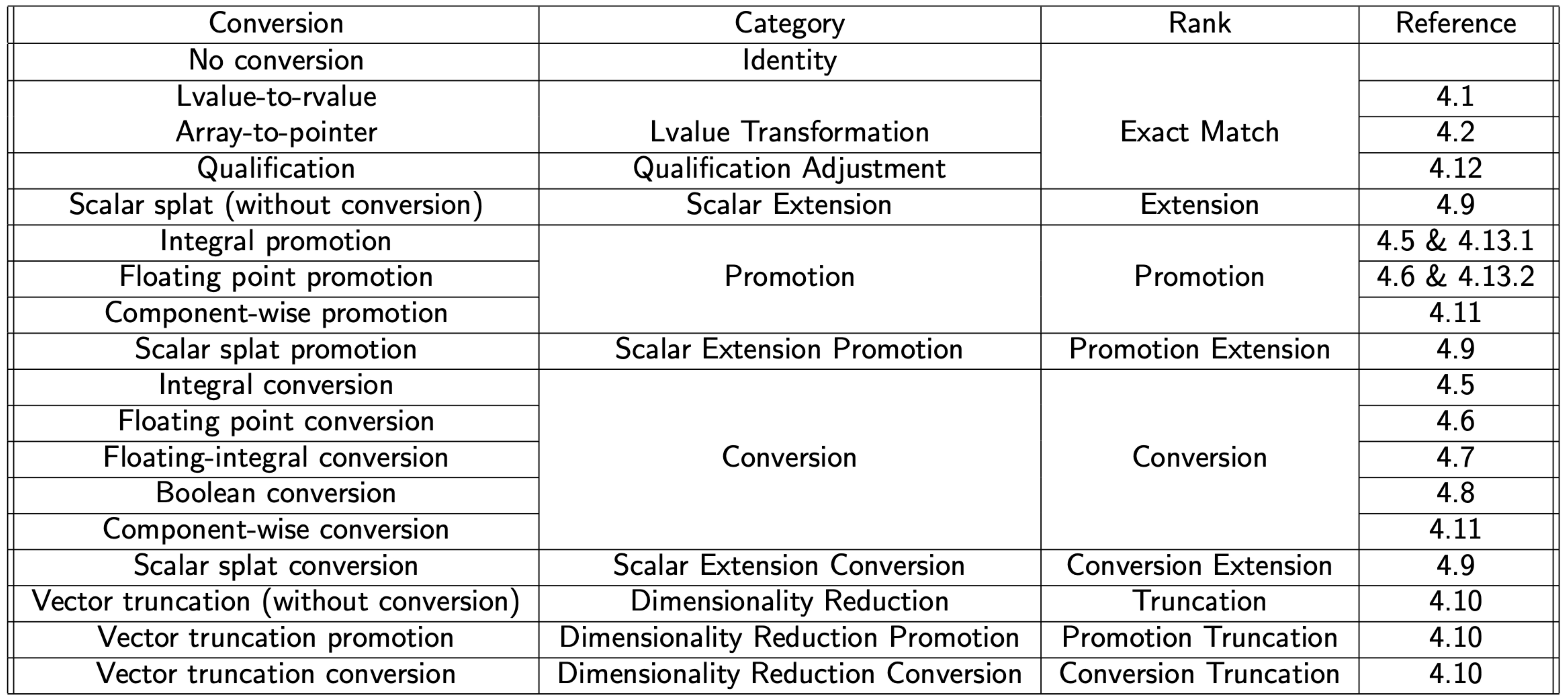


New Problem: Moar Conversions!

- Immediately complicates overload scoring…
  - 3 ranks turns to 9



Conversion	**Category**	**Rank**	**Subclause**
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Boolean conversions			4.12

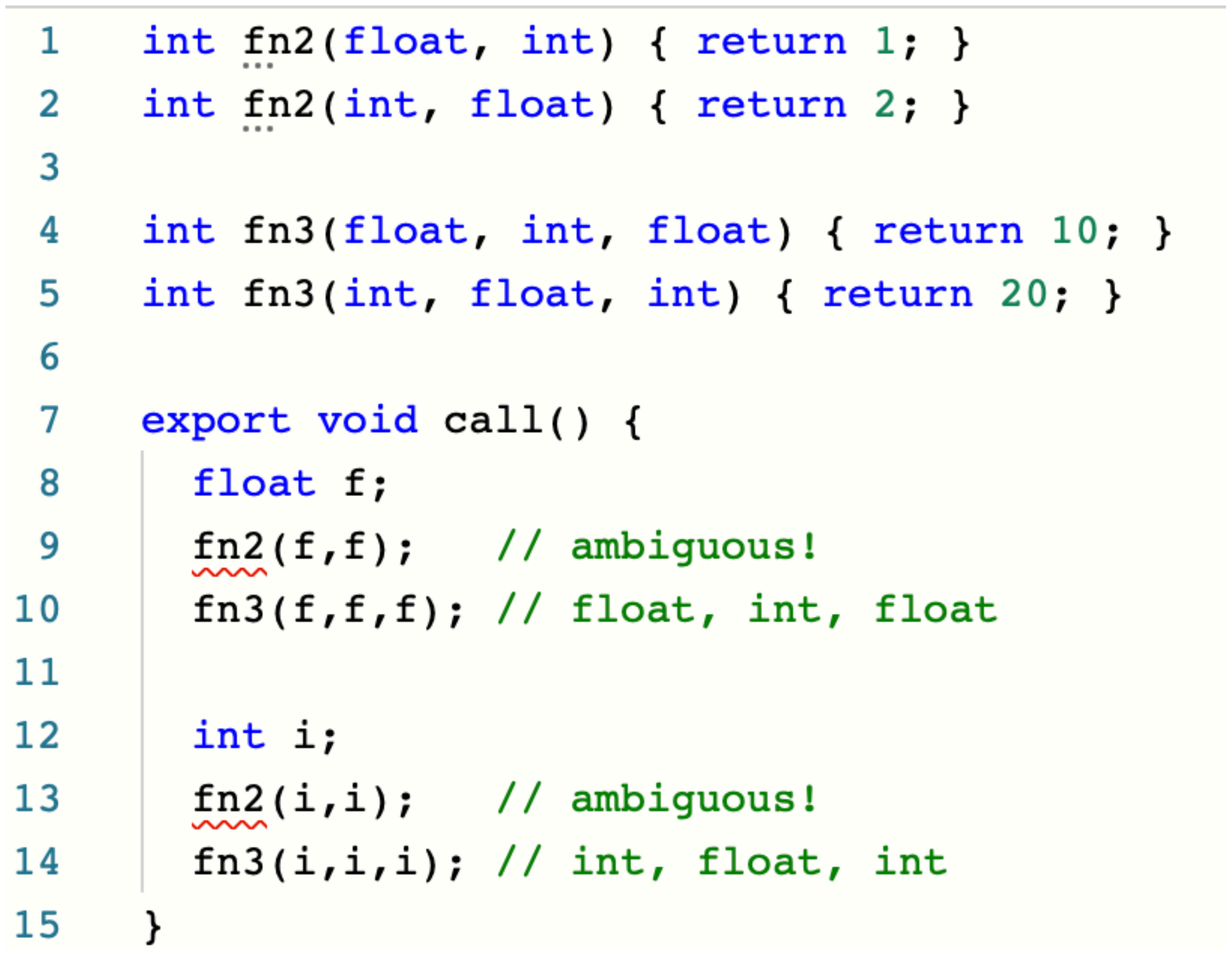


Conversion	Category	Rank	Reference
No conversion	Identity	Exact Match	
Lvalue-to-rvalue	Lvalue Transformation		4.1
Array-to-pointer			4.2
Qualification	Qualification Adjustment		4.12
Scalar splat (without conversion)	Scalar Extension	Extension	4.9
Integral promotion	Promotion	Promotion	4.5 & 4.13.1
Floating point promotion			4.6 & 4.13.2
Component-wise promotion			4.11
Scalar splat promotion	Scalar Extension Promotion	Promotion Extension	4.9
Integral conversion	Conversion	Conversion	4.5
Floating point conversion			4.6
Floating-integral conversion			4.7
Boolean conversion			4.8
Component-wise conversion			4.11
Scalar splat conversion	Scalar Extension Conversion	Conversion Extension	4.9
Vector truncation (without conversion)	Dimensionality Reduction	Truncation	4.10
Vector truncation promotion	Dimensionality Reduction Promotion	Promotion Truncation	4.10
Vector truncation conversion	Dimensionality Reduction Conversion	Conversion Truncation	4.10

Difference: Best Match Resolution

Reason: Lineage from Cg

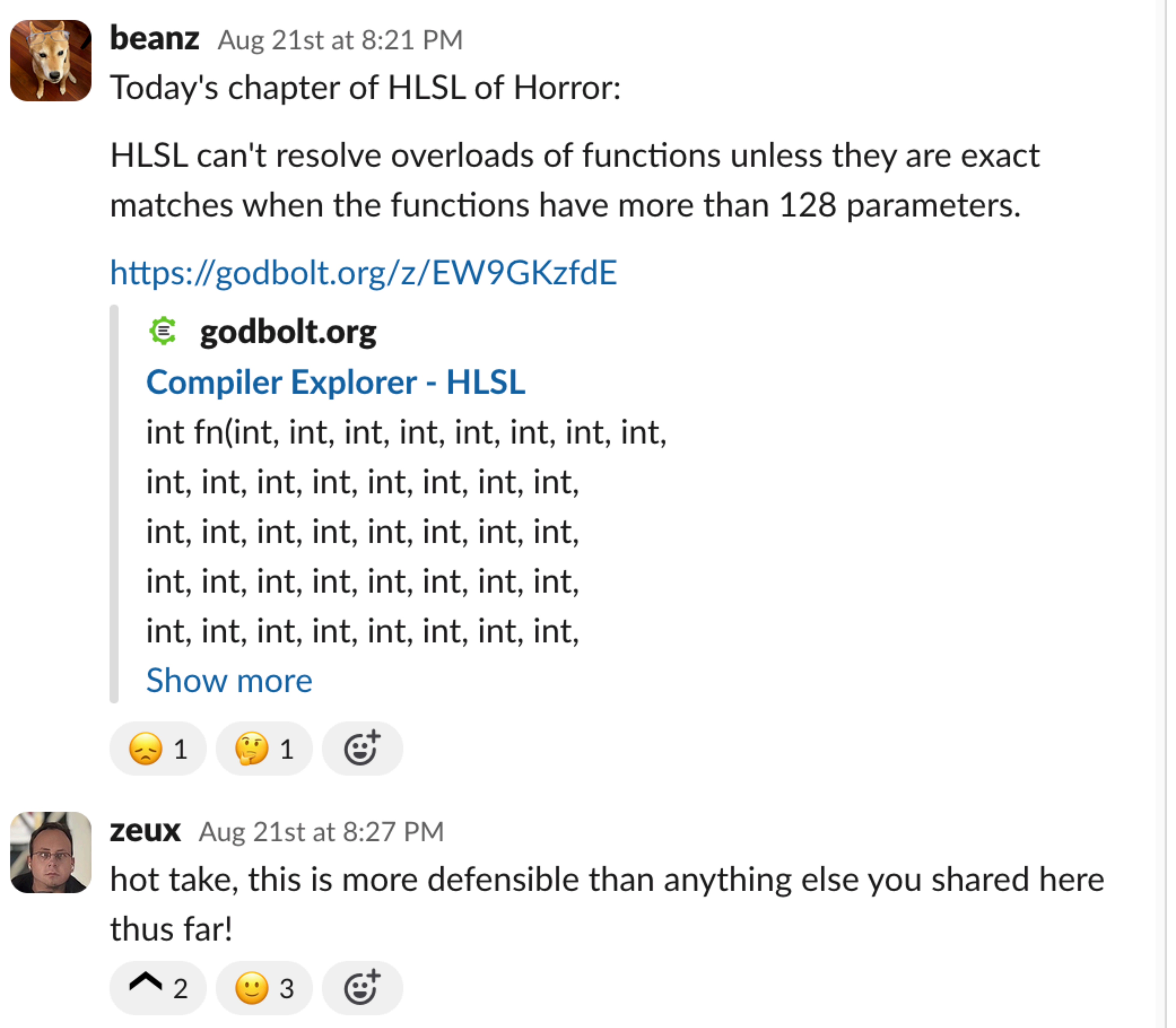
- In C++ all these cases would be ambiguous!
- DXC uses a scoring system
  - Disambiguates cases where “worse” conversions are present
  - Produces unintuitive results
  - Scores have a limit



```
int fn2(float, int) { return 1; }
int fn2(int, float) { return 2; }

int fn3(float, int, float) { return 10; }
int fn3(int, float, int) { return 20; }

export void call() {
  float f;
  fn2(f,f);    // ambiguous!
  fn3(f,f,f); // float, int, float

  int i;
  fn2(i,i);    // ambiguous!
  fn3(i,i,i); // int, float, int
}
```

Solution: Define ICS Rules

- Try and align with C++ as much as possible
  - Use C++’s better/worse/indistinguishable
- New behavior _does not_ match DXC
  - DXC used a number-based scoring system
- New behavior has more ambiguous cases, but it also handles more cases (operator overloads & const-ness)
- Behavior changes that break compiling are preferred!



beanz Aug 21st at 8:21 PM
Today's chapter of HLSL of Horror:

HLSL can't resolve overloads of functions unless they are exact matches when the functions have more than 128 parameters.

https://godbolt.org/z/EW9GKzfdE

godbolt.org
Compiler Explorer - HLSL
int fn(int, int, int, int, int, int, int, int,
int, int, int, int, int, int, int, int,
int, int, int, int, int, int, int, int,
int, int, int, int, int, int, int, int,
int, int, int, int, int, int, int, int,
Show more

😞 1 🤔 1

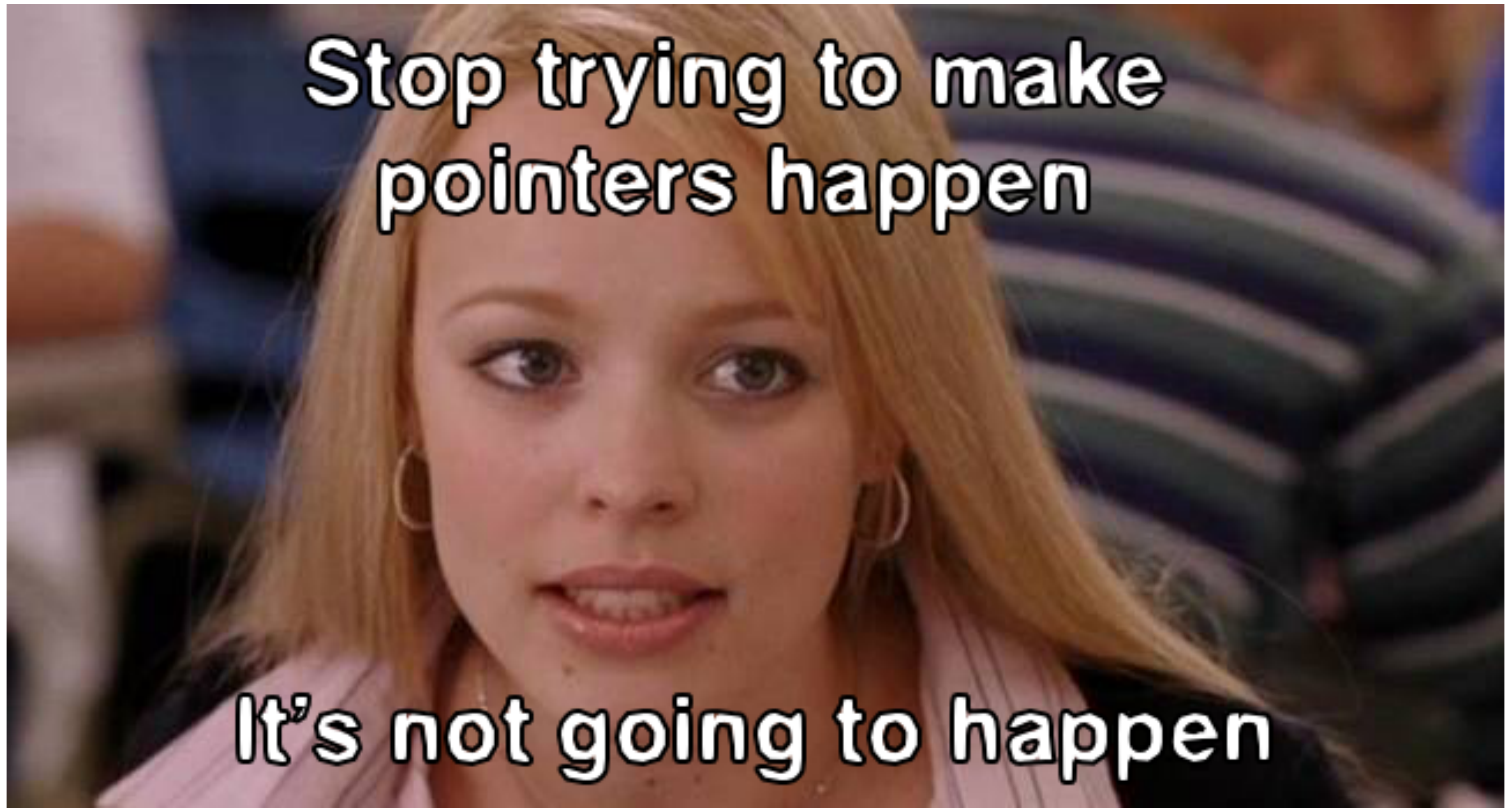
zeux Aug 21st at 8:27 PM
hot take, this is more defensible than anything else you shared here thus far!

^ 2 🙂 3

Difference: No pointers!

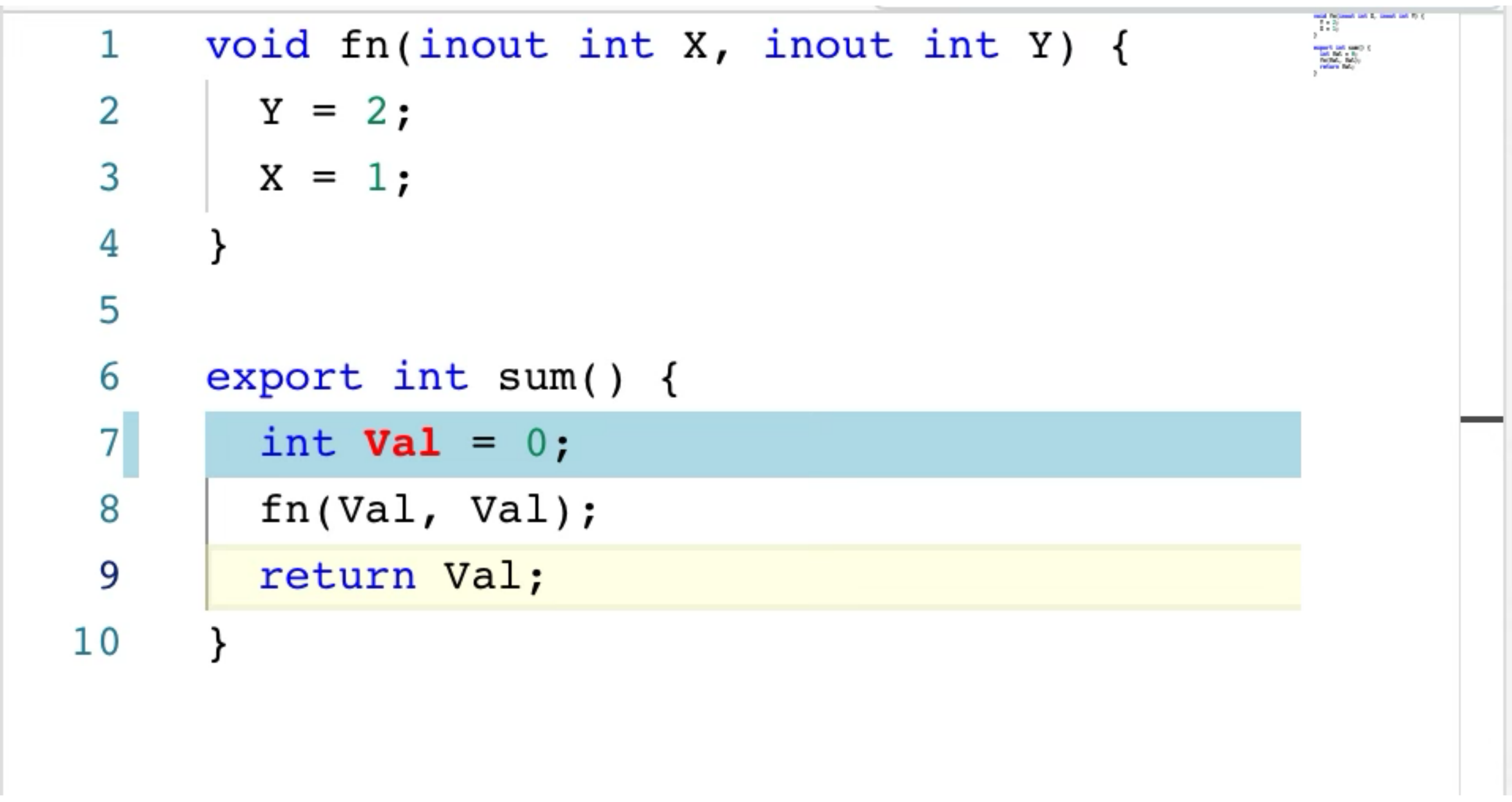
Reason: part history, part complexity

- Early GPUs didn't really have direct memory access
- Modern GPU memory has fragmented memory spaces
- Pointers and unified memory are becoming more common

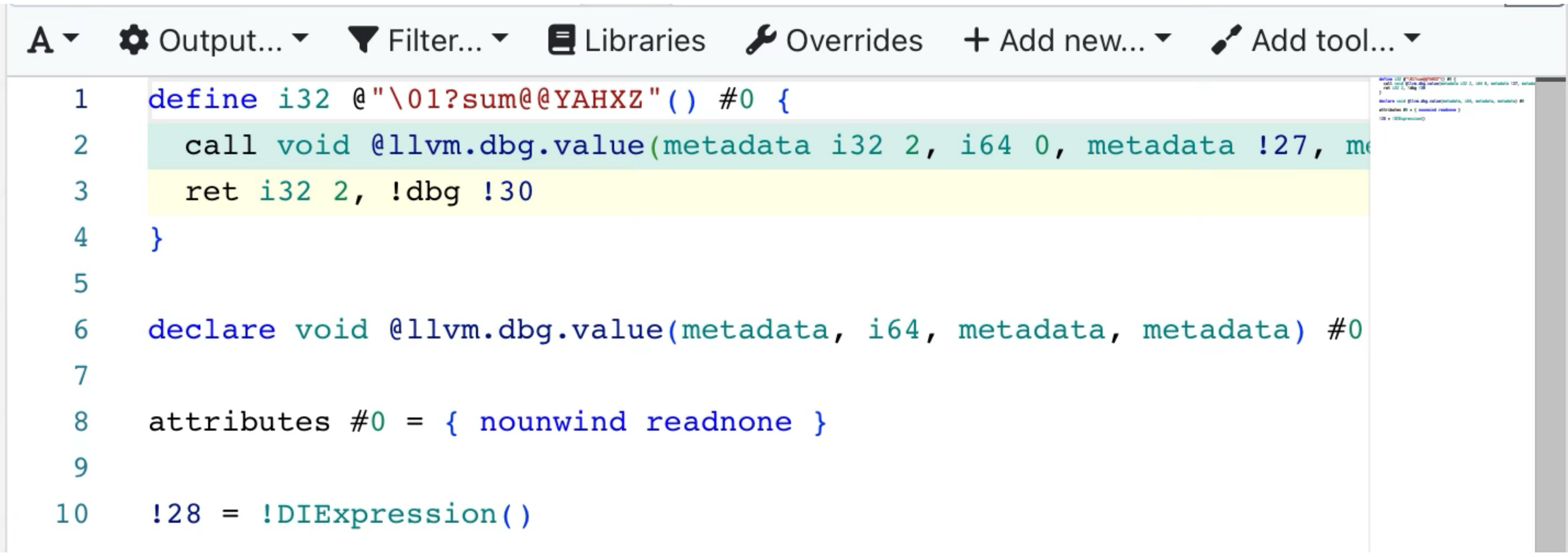


Problem: Multiple Function Outputs

- HLSL's inout and out parameter keywords are still passed by value
- This normalizes addresses for parameters to thread-local memory



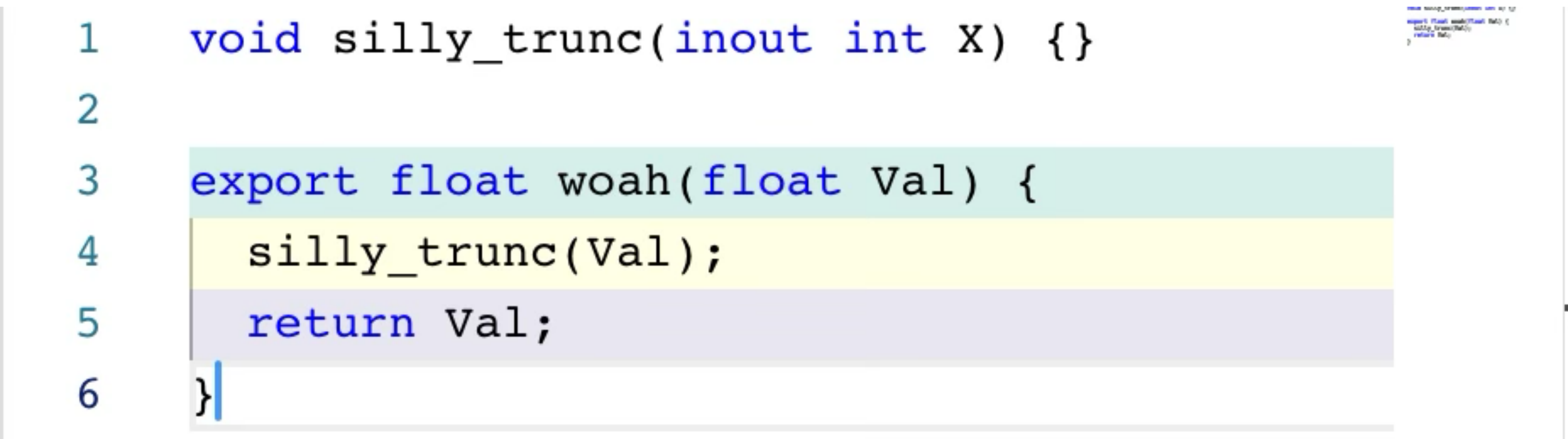
```
void fn(inout int X, inout int Y) {
  Y = 2;
  X = 1;
}

export int sum() {
  int Val = 0;
  fn(Val, Val);
  return Val;
}
```



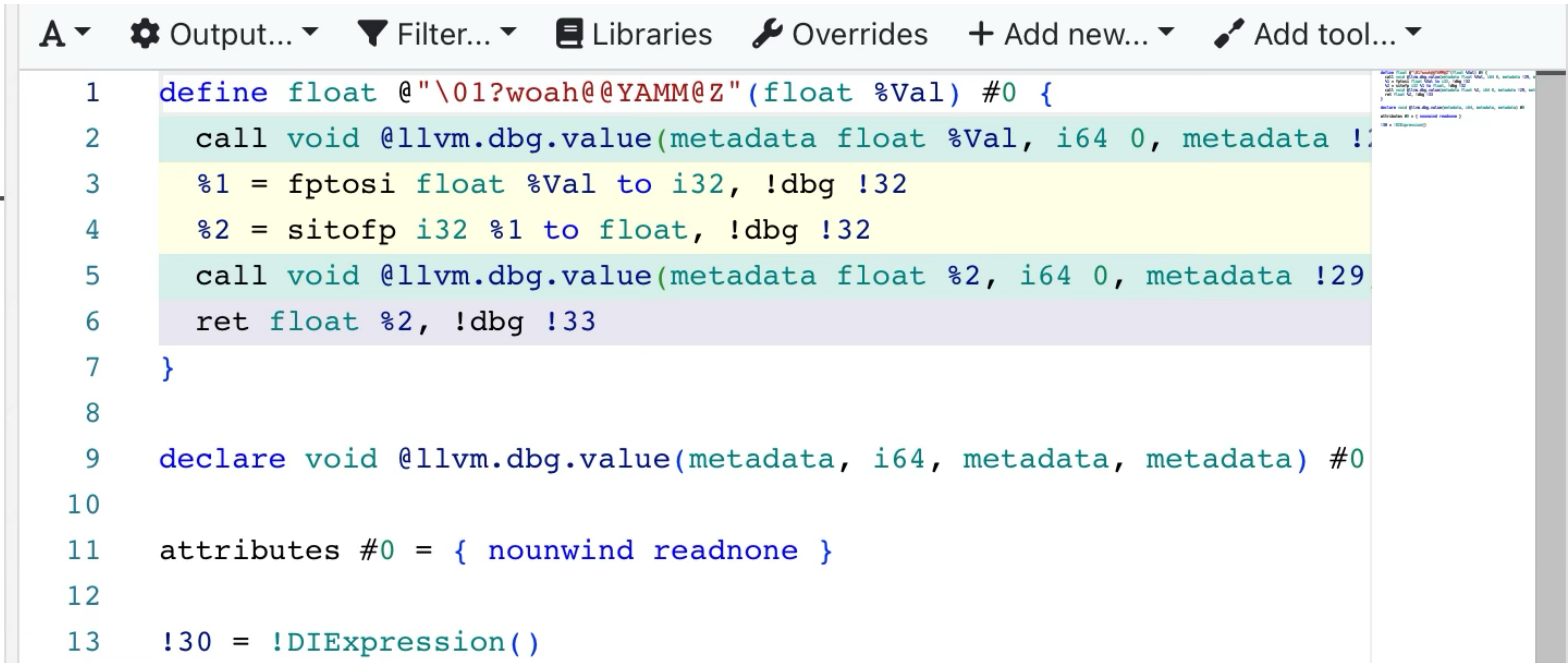
```
define i32 @"\01?sum@@YAHXZ"() #0 {
  call void @llvm.dbg.value(metadata i32 2, i64 0, metadata !27, me
  ret i32 2, !dbg !30
}

declare void @llvm.dbg.value(metadata, i64, metadata, metadata) #0

attributes #0 = { nounwind readnone }

!28 = !DIExpression()
```

Complication: Outputs Can Cast?!?!

- HLSL's output parameters can also have cast



```
void silly_trunc(inout int X) {}

export float woah(float Val) {
  silly_trunc(Val);
  return Val;
}
```



```
define float @"\01?woah@@YAMM@Z"(float %Val) #0 {
  call void @llvm.dbg.value(metadata float %Val, i64 0, metadata !
  %1 = fptosi float %Val to i32, !dbg !32
  %2 = sitofp i32 %1 to float, !dbg !32
  call void @llvm.dbg.value(metadata float %2, i64 0, metadata !29
  ret float %2, !dbg !33
}

declare void @llvm.dbg.value(metadata, i64, metadata, metadata) #0

attributes #0 = { nounwind readnone }

!30 = !DIExpression()
```

Casting Expiring Values

- cx-values are a spec-level construct only
- Vocabulary for casted temporary values
- x-value with a bound l-value
- may be initialized from an l-value or it may be uninitialized
- on expiration it “writes back” to the bound l-value
- it can optionally cast on both initialization and write back

Solution: HLSLOutArgExpr

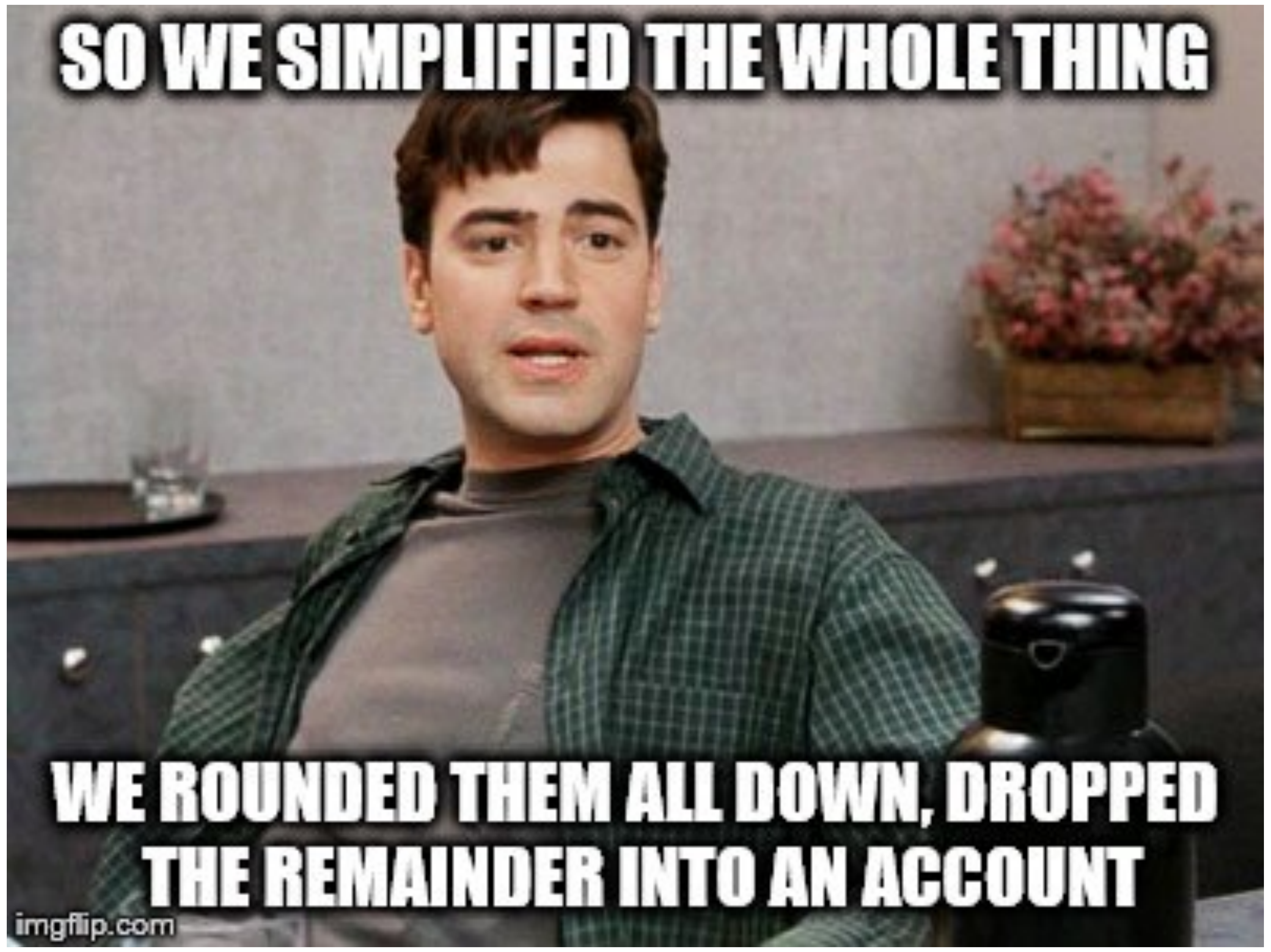
- New AST node to represent the cx-value
- Use Parameter ABI type information added for Swift
- Leverage call argument write back support added for Objective-C
- All casts are represented in the AST!

```
CallExpr 'void'
|-ImplicitCastExpr 'void (*)(int &)' <FunctionToPointerDecay>
| `-DeclRefExpr 'void (int &)' lvalue Function  'Init' 'void (int &)'
`-HLSLOutArgExpr <col:10> 'int' lvalue inout
  |-OpaqueValueExpr 0xSOURCE <col:10> 'int' lvalue
  | `-DeclRefExpr <col:10> 'int' lvalue Var 'V' 'int'
  |-OpaqueValueExpr 0xTEMPORARY <col:10> 'int' lvalue
  | `-ImplicitCastExpr <col:10> 'int' <LValueToRValue>
  |   `-OpaqueValueExpr 0xSOURCE <col:10> 'int' lvalue
  |     `-DeclRefExpr <col:10> 'int' lvalue Var 'V' 'int'
  `-BinaryOperator <col:10> 'int' lvalue '='
    |-OpaqueValueExpr 0xSOURCE <col:10> 'int' lvalue
    | `-DeclRefExpr <col:10> 'int' lvalue Var 'V' 'int'
    `-ImplicitCastExpr <col:10> 'int' <LValueToRValue>
      `-OpaqueValueExpr 0xTEMPORARY <col:10> 'int' lvalue
        `-ImplicitCastExpr <col:10> 'int' <LValueToRValue>
          `-OpaqueValueExpr 0xSOURCE <col:10> 'int' lvalue
            `-DeclRefExpr <col:10> 'int' lvalue Var 'V' 'int'
```

Difference: literal types

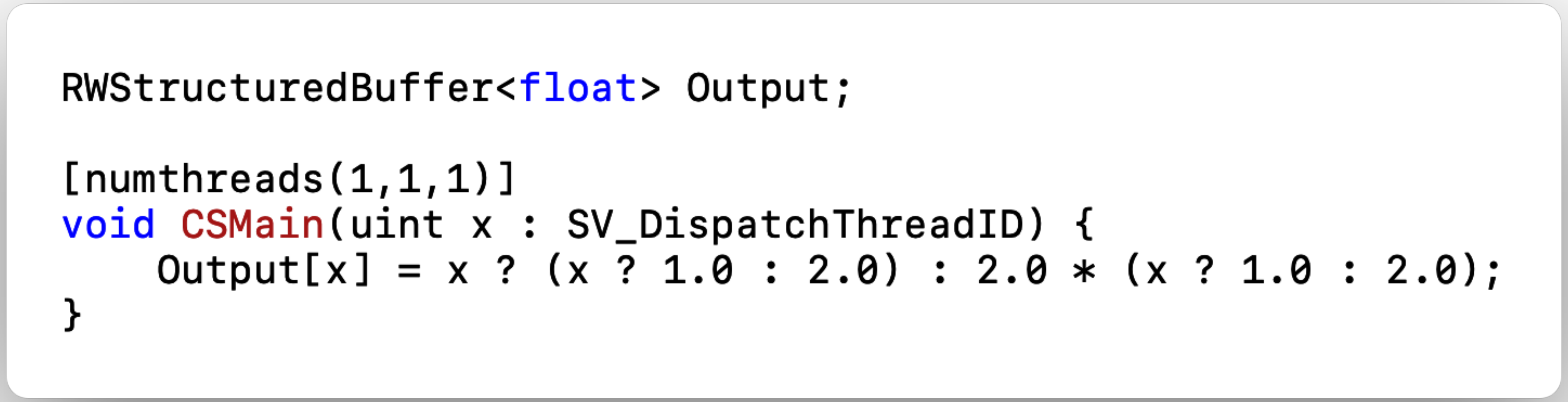
Reason: Low-precision math is fast

- Not unique to HLSL!
  - WGSL calls these abstract numeric types
- Allow compile-time expressions to use higher precision
- AST-level literals are treated as 64-bit for constant evaluation
- Implicitly truncated to a target type… usually

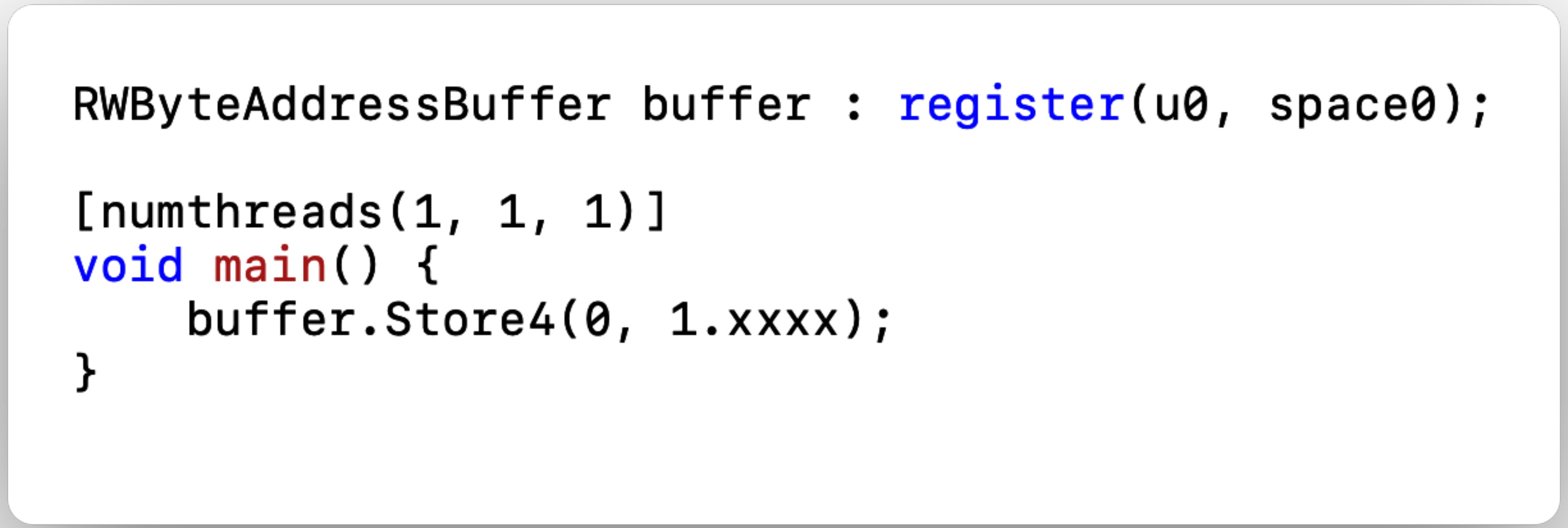

SO WE SIMPLIFIED THE WHOLE THING
WE ROUNDED THEM ALL DOWN, DROPPED THE REMAINDER INTO AN ACCOUNT
imgflip.com

Problem: Unresolved Types

- DXC’s literal handling is extremely complicated!
    - Implementation had strange bugs
- C++ isn’t designed for multi-expression type inference



```
RWStructuredBuffer<float> Output;

[numthreads(1,1,1)]
void CSMain(uint x : SV_DispatchThreadID) {
    Output[x] = x ? (x ? 1.0 : 2.0) : 2.0 * (x ? 1.0 : 2.0);
}
```



```
RWByteAddressBuffer buffer : register(u0, space0);

[numthreads(1, 1, 1)]
void main() {
    buffer.Store4(0, 1.xxxx);
}
```

Solution: HLSL 202x

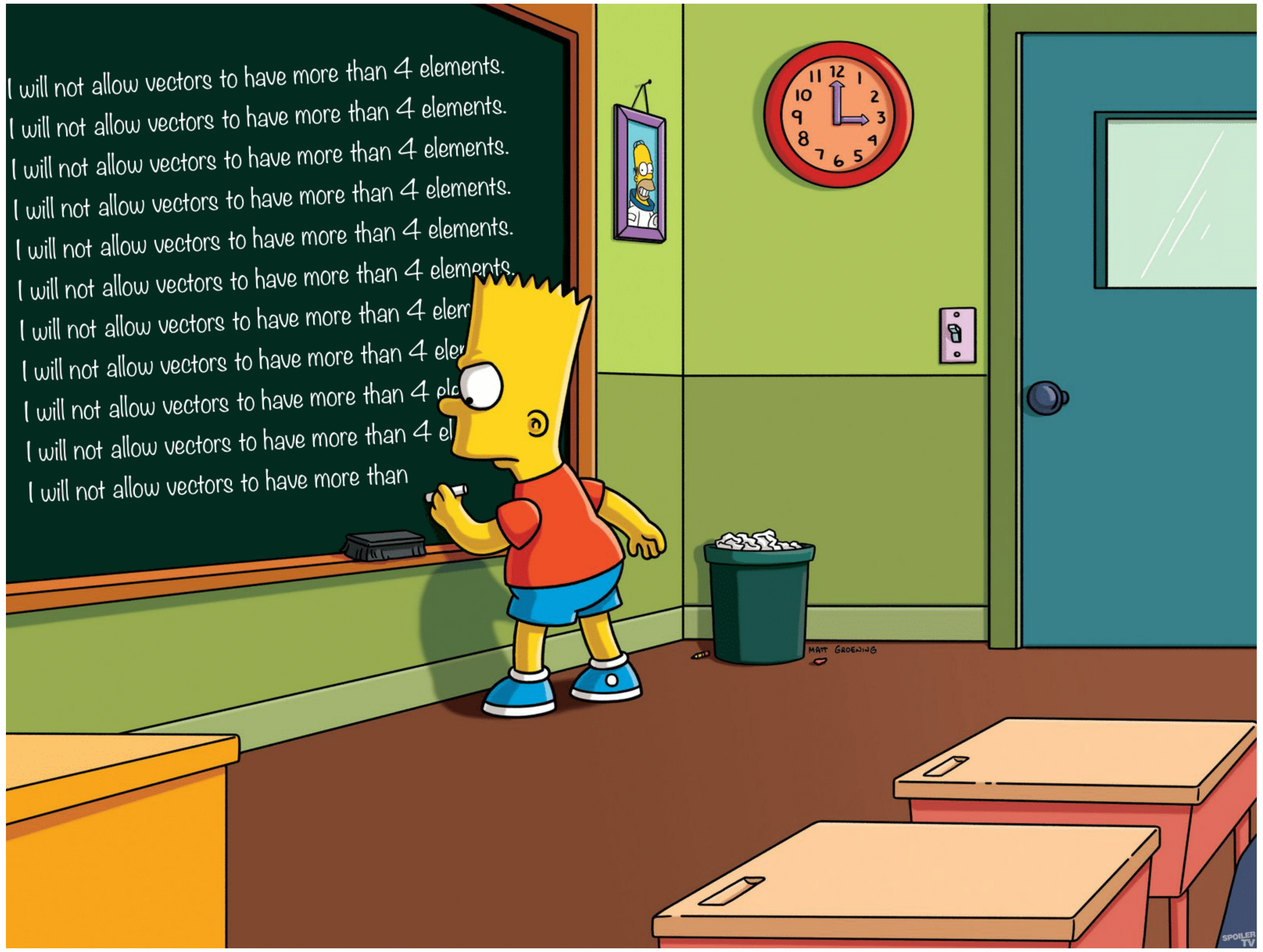
Bridge to Clang

- Without a specification we can't make Clang match
- When possible we're implementing breaking changes in DXC
- Fix bugs in DXC and not introduce complexity or debt to Clang

Problem: Constraints on Templates

Reason: History

- HLSL for years has had template-like syntax for types
- Some built-in types have constraints on valid template arguments
  - vectors must have arithmetic or boolean type
  - vector size must be <= 4
  - Resource types have rules for different types


I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than 4 elements.
I will not allow vectors to have more than

Solution: A Concept of a Plan

- Built-in types are injected into the AST
- Inject a concept declaration & concept specialization
- Avoids needing to add custom checking for types in template instantiation
- Concepts are _super_ useful for GPUs!

Path Forward

- Diligence in writing a spec and docs
- Creative about how we achieve source compatibility
- Considerate about the future of the language
- Open and engaging with our users

Questions?