Modular

BILLY ZHU
CHRIS LATTNER

OCT 28, 2025
LLVM DEV MEETING

Building Modern Language Frontends with MLIR

Lessons from Mojo's Compile-Time Meta-Programming

Agenda

01 Brief Mojo Primer

02 Meta-Programming in Mojo

03 MLIR Parametric Code Representation

04 Parameter Domain Computations

05 What Does This Mean in Practice?

Brief Mojo Primer

Modular

Why Mojo🔥?

Requirement: Needed a portable systems programming language for heterogeneous compute

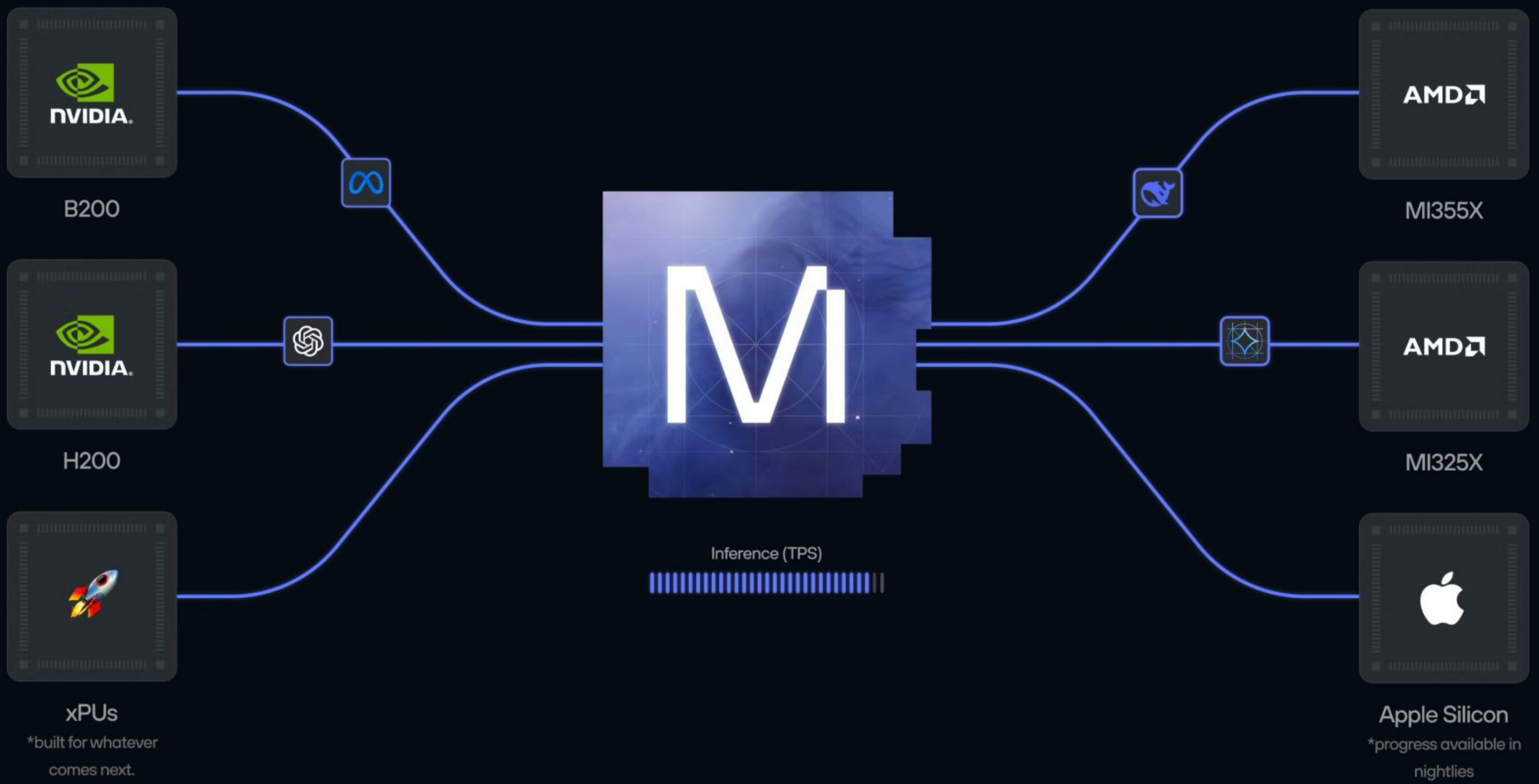
Goal: unlock the full power of the hardware by giving programmers:

- low level control
- zero-cost abstractions
- safety guarantees

Approach: Willing to invest a lot to get the best quality, developer UX, and performance

- Mojo is a real language, not a Python eDSL

```
def mandelbrot_kernel[
   width: Int  # SIMD Width
](c: ComplexSIMD[float, width]) ->
  SIMD[int, width]:

  """A vectorized implementation of
  the inner mandelbrot computation."""

  z = ComplexSIMD[float, width](0, 0)
  iters = SIMD[DType.index, width](0)
  mask = SIMD[DType.bool, width](True)

  for i in range(MAX_ITERS):
    if not any(mask):
      break

    mask = z.squared_norm() <= 4
    iters = mask.select(iters + 1, iters)
    z = z.squared_add(c)

  return iters
```



B200
H200
xPUs
*built for whatever comes next.
Inference (TPS)
MI355X
MI325X
Apple Silicon
*progress available in nightlies

Modular

Blackwell Throughput vs vLLM

Note: We use vllm 0.9.0.1 as a reference in some cases cause vllm 0.10.0 crashes with memory errors due to flash infer.

153.4%

Llama3.1 - 405b arxiv-summarization
NVIDIA - B200

0.718 rps

0.468 rps

*Modular v25.4 vs vLLM v0.10.0

152.7%

Llama3.1 - 405b code_debug
NVIDIA - B200

0.718 rps

0.468 rps

*Modular v25.4 vs vLLM v0.9.0.1

104.3%

Llama3.1 - 405b sonnet-prefill-heavy-prefix-low
NVIDIA - B200

15.437 rps

14.808 rps

*Modular v25.4 vs vLLM v0.9.0.1

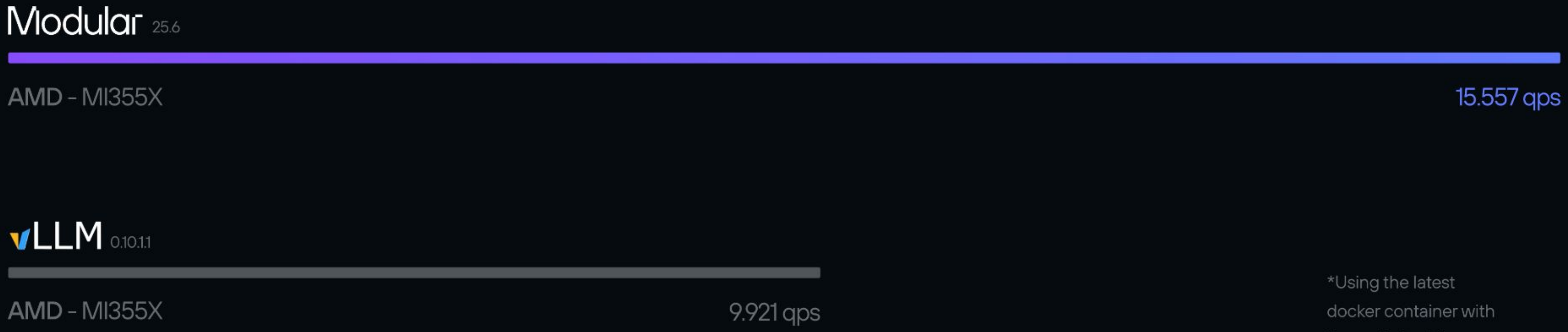
Modular

AMD MI355 Throughput vs vLLM

171%

Improved throughput

Gemma-3-27B
Sonnet Decode Heavy



*Using the latest docker container with the optimized flags

How? Meta-Programming

Mojo is a "predictable" language + compiler:

- Does not embed AI or HW knowledge into the compiler itself
- Eschews "compiler magic"

Metaprogramming enables powerful features

- Specialization, codegen, autotuning, etc
- Empower Mojo programmers with "generalization" power

Mojo is cool for other reasons too:

- Powerful type system
- Fancy MLIR compiler implementation
- See many previous llvmdev talks, e.g. *"Mojo🔥: A system programming language for heterogeneous computing"* at LLVM Dev 2023 [link]

```
fn fill_fib(size: Int) -> List[Int]:
  if size <= 0:
    return []
  if size == 1:
    return [0]
  var fib: List[Int] = [0, 1]
  for idx in range(2, size):
    fib.append(fib[idx-2] + fib[idx-1])
  return fib^


fn main():
  # List computed at compile-time.
  alias a6 = fill_fib(6)

  # Unrolled by iterating over List.
  @parameter
  for elem in a6:
    print(elem)

  # Same code computed at run-time.
  var v6 = fill_fib(6)
  for elem in v6:
    print(elem)
```

Modular

C++ already does this!?

Template Meta-Programming is the existing standard in GPU programming

Terrible error messages due to “duck typing”

- Errors show up as instantiation problems

Different comp-time and runtime languages

- Hard to learn & awkward to use

Limited type system:

- Cannot use computed strings or trees ergonomically

Slow compile times, can’t use debuggers on comptime code, …

```
template <class LayoutA,
          class Offset,
          class LayoutB>
CUTE_HOST_DEVICE constexpr
auto
composition(LayoutA const& layoutA,
            Offset  const& offset,
            LayoutB const& layoutB)
{
  return ComposedLayout<LayoutA, Offset, LayoutB>{layoutA, offset, layoutB};
}

template <class A, class O, class B, class Tiler>
CUTE_HOST_DEVICE constexpr
auto
composition(ComposedLayout<A,O,B> const& a,
            Tiler                 const& b)
{
  return composition(a.layout_a(), a.offset(), composition(a.layout_b(), b));
}

template <class ShapeA, class StrideA,
          class A, class O, class B>
CUTE_HOST_DEVICE constexpr
auto
composition(Layout<ShapeA,StrideA> const& a,
            ComposedLayout<A,O,B>  const& b)
{
  CUTE_STATIC_ASSERT_V(b.offset() == Int<0>{}, "Require offset == 0.");

  return composition(composition(a, b.layout_a()), b.layout_b());
}
```

Meta-Programming in Mojo🔥

Modular

Code vs Meta-Code

Normal functions have "*arguments*":

- These are runtime values that can vary on every invocation

Mojo allows [comptime arguments]:

- Mojo instantiates these - like a template
- We refer to these as *"parameters"*

Parameters get superpowers:

- e.g. can fully unroll a loop

Expressions are written the same way in both roles

```
# Naive power function.
fn pow(base: Int, exp: Int) -> Int:
  res = 1
  for i in range(exp):
    res *= base
  return res

# Easily refactor when `exp` can be known
# at compile time.
fn pow_exp[exp: Int](base: Int) -> Int:
  res = 1
  for i in range(exp):
    res *= base
  return res


# Can also unroll the loop at compile time.
fn pow_exp_fast[exp: Int](base: Int) -> Int:
  res = 1
  @parameter
  for i in range(exp):
    res *= base
  return res
```

Modular

Anything can be parameterized… by anything 🤯

Can parameterize:

- Functions, Types, Closures, etc.
- Even *arbitrary expressions*

Enables powerful user-defined libraries:

- Very important for GPU programming

Simplifies and unifies the language:

- Types are just comptime values
- Removes many special case features in other languages

```
# Complex custom datatype.
struct Layout:
  var shape: List[Int]   # has malloc
  var stride: List[Int]

  @staticmethod
  fn col_major(shape: List[Int]) -> Layout:
    return …


struct LayoutTensor[
  dtype: DType,
  layout: Layout, # Used at comptime
  ...
]:

  # Methods can add more parameters too.
  fn load[width: Int](self, *idx: Int)
     -> SIMD[dtype, width]:
    return …
```

How does this work?

Explicit control over expression evaluation:

- "alias" guarantees comp-time eval
- "var" is dynamic at its execution time

Meta-code is code run at comp-time

- Executed by an IR interpreter
- Supports ~arbitrary logic, including malloc

Can materialize finished values to runtime

- Shifting work to comp-time

Can also *compute types* in the type system!

- Mojo has powerful "dependent types"

```
# Returns a heap-allocated List.
fn fill_fib(size: Int) -> List[Int]:
  if size <= 0:
    return []
  if size == 1:
    return [0]
  var fib: List[Int] = [0, 1]
  for idx in range(2, size):
    fib.append(fib[idx-2] + fib[idx-1])
  return fib^


fn main():
  # List computed at compile-time.
  alias a6 = fill_fib(6)

  # Unrolled by iterating over List.
  @parameter
  for elem in a6:
    print(elem)

  # Another list computed at run-time.
  var v6 = fill_fib(6)
  for elem in v6:
    print(elem)
```

Polymorphic Generic IR:

Type checked *without instantiating*

What we want:

- Maintain target info for split compilation
- Good error messages, not template stack traces
- Fast compiles + don't ship source code

Type check + Generate IR **before instantiating**:

- Type checking is symbolic instead of concrete

How do we represent uninstantiated "templates" polymorphically in IR??

```
fn exp2(x: SIMD[dtype, simd_width]) -> SIMD[dtype, simd_width]:

   @parameter
   if is_nvidia_gpu():
       @parameter
       if dtype is DType.float16:
           @parameter
           if _is_sm_9x_or_newer():
               return _call_ptx_intrinsic[
                   scalar_instruction="ex2.approx.f16",
                   vector2_instruction="ex2.approx.f16x2",
                   scalar_constraints="=h,h",
                   vector_constraints="=r,r",
               ](x)
           else:
               return _call_ptx_intrinsic[
                   instruction="ex2.approx.f16",
                   constraints="=h,h",
               ](x)
       elif dtype is DType.bfloat16 and _is_sm_9x_or_newer():
           return _call_ptx_intrinsic[
               scalar_instruction="ex2.approx.ftz.bf16",
               vector2_instruction="ex2.approx.ftz.bf16x2",
               scalar_constraints="=h,h",
               vector_constraints="=r,r",
           ](x)
       elif dtype is DType.float32:
           return _call_ptx_intrinsic[
               instruction="ex2.approx.ftz.f32",
               constraints="=f,f",
           ](x)
```

Modular

MLIR Parametric Code Representation

Mojo in MLIR

Uses MLIR to express manipulation of MLIR itself.

The parameterized “base IR” forms the basic IR building blocks. It carries application logic:

- Parameterized Types
- Parameterized Ops

The “meta IR” manipulates the “base IR”:

- Parametric Declarations
- Inline Instantiations

Mojo

```
fn simd_sum
  [dtype: DType, size: Int]
  (value: SIMD[dtype, size])
  -> Scalar[dtype]:
    var sum: Scalar[dtype] = 0
    @parameter
    for i in range(size):
        sum += value[i]
    return sum
```

MLIR

```
lit.fn @simd_sum
  <dtype: @DType, size: @Int>
  (%value: @SIMD<dtype, size>)
    -> @SIMD<dtype, 1> {
  ???
}
```

Parameterized Types

Parameterize

- Dig "typed holes" 🕳

Instantiate

- Fill holes with values of the correct type 🎱

Traditional Types

- Expect specific kinds of Attributes (e.g. IntegerAttr)

Parametric Ops

- Expect attributes of a specific Type

Traditional Type

```
def POP_SIMDType {
  let parameters = (ins
    IndexAttr:$size,
    POP_DTypeConstantAttr:$dtype
  )
}
```

E.g.

```
%arr = pop.simd.add(%s0, %s1)
  : !pop.simd<4, f32>
```

Parametric Type

```
def POP_SIMDType {
  let parameters = (ins
    TypedAttr:$size,
    TypedAttr:$dtype
  )
}
```

E.g.

```
%arr = pop.simd.add(%s0, %s1)
  : !pop.simd<🕳 : index, 🕳 : dtype>
```

```
🎱 builtin.integer<2> : index
🎱 kgen.param.ref<"size"> : index
🎱 ...
```

Modular

Parameterized Ops

Op Attribute/Properties can be parameterized too

A key part of programmatic codegen

Traditional Ops

- Expect specific kinds of Attributes (e.g. IntegerAttr)

Parametric Ops

- Expect attributes of a specific Type

Traditional Op

```
let arguments = (ins
  POP_SIMDType:$vector,
  IndexAttr:$idx
);

pop.simd.extract %arr {
  idx = IntegerAttr<3> : index
}
```

Parametric Op

```
let arguments = (ins
  POP_SIMDType:$vector,
  IndexTypedAttr:$idx
);

pop.simd.extract %arr {
  idx =   : index
}
         builtin.integer<2> : index
         kgen.param.ref<"size"> : index
         ...
```

Parametric Declarations

Declarations (e.g. functions, struct types) encapsulate a block of parametric IR over a set of input parameters

- The input parameters are typed and named
- The body of the declaration can refer to the input parameters

Enables reuse of the parametric body with different input parameter values

Parametric Function

```
lit.fn @simd_sum
  <size: Int, dtype: DType>
  (
    %value: !pop.simd<
      #kgen.param.ref<”size”>,
      #kgen.param.ref<”dtype”>
    >
  )
    -> !pop.simd<
         1,
         #kgen.param.ref<”dtype”>
       >
{
  ...
}

lit.fn @main {
  ...
  %s2 = pop.simd.splat %scalar
    : !pop.simd<2, f32>
  %s4 = pop.simd.splat %scalar
    : !pop.simd<4, f32>
  lit.call @simd_sum<2, f32>(%s2)
  lit.call @simd_sum<4, f32>(%s4)
}
```

Inline Instantiations

Instantiate a block of parametric IR inline

Parameter-If

- Select one of the blocks to instantiate based on some parameter value

Parameter-For

- Instantiate a block multiple times
- Declares an input parameter representing the value returned by an iterator
- The body will be instantiated once with each iterated value

Conditional Instantiation

```
example.before
kgen.param.if <condition> {
  example.do_something
} else {
  example.do_something_else
}
example.after
```

E.g. when `condition` is False

```
example.before
example.do_something_else
example.after
```

Repeated Instantiation

```
example.before
kgen.param.for i in iterable ... {
  example.do_something { attr = i }
}
example.after
```

E.g. when `iterable` is range(2)

```
example.before
example.do_something { attr = 0 }
example.do_something { attr = 1 }
example.after
```

Putting them together …

🎱 Parametric IR

```
lit.fn @simd_sum
  <size: @Int, dtype: @DType>
  (%value: @SIMD<size, dtype>)
    -> @SIMD<1, dtype> {
  ...
  kgen.param.for iter in ... {
    ...
    kgen.param.call @other_fn<size>()
    ...
    kgen.param.if iter {
      ...
      my.param.op { dt = dtype }
      ...
    }
    kgen.deferred {
      name = “hw.specific”,
      attrs = {...} }
  }
  %c = kgen.param.constant = <size>
  ...
}
```

Parameterized Ops

```
lit.fn @simd_sum
  <size: @Int, dtype: @DType>
  (%value: @SIMD<    ,      >)
    -> @SIMD< ,       > {
  ...
  kgen.param.for iter in ... {
    ...
    kgen.param.call @other_fn<    >()
    ...
    kgen.param.if      {
      ...
      my.param.op { dt =        }
      ...
    }
    kgen.deferred {
      name = "hw.specific",
      attrs = {   } }
  }
  %c = kgen.param.constant = <    >
  ...
}
```

Parameters

```
size: @Int  dtype: @DType
              size  dtype
          1  dtype

             iter

                          size

              iter

                      dtype

        ...

                          size
```

Modular

MLIR enables a natural interleave of …

Meta IR

Base IR

Parameterized Ops

```
lit.fn @simd_sum
  <size: @Int, dtype: @DType>
  (%value: @SIMD<    ,      >)
    -> @SIMD< ,       > {
  ...
  kgen.param.for iter in ... {
    ...
    kgen.param.call @other_fn<    >()
    ...
    kgen.param.if      {
      ...
      my.param.op { dt =       }
      ...
    }
    kgen.deferred {
      name = "hw.specific",
      attrs = {   } }
  }
  %c = kgen.param.constant = <    >
  ...
}
```

Parameters

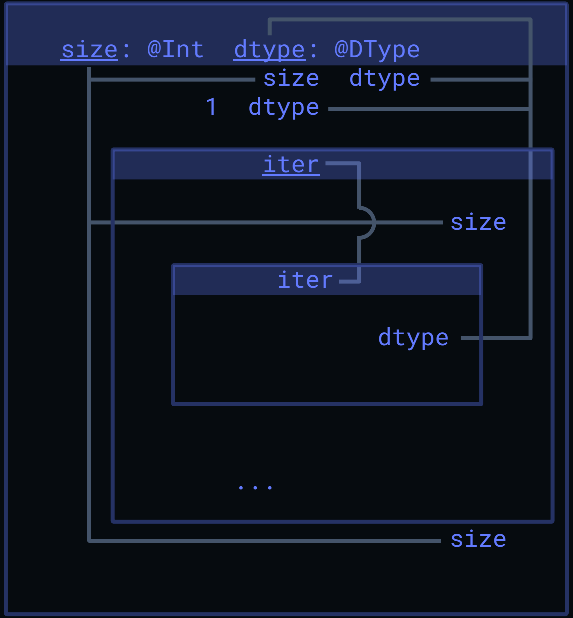
```
size: @Int  dtype: @DType
              size  dtype
          1  dtype

             iter

                          size

              iter

                    dtype



            ...

                          size
```

Meta IR

Op Scopes

```
lit.fn @simd_sum
  <size: @Int, dtype: @DType>
  (%value: @SIMD<     ,       >)
    -> @SIMD< ,       > {
  ...
  kgen.param.for iter in ... {
    ...
    kgen.param.call @other_fn<    >()
    ...
    kgen.param.if       {
      ...
      my.param.op { dt =        }
      ...
    }
    kgen.deferred {
      name = “hw.specific”,
      attrs = {   } }
  }
  %c = kgen.param.constant = <    >
  ...
}
```

Parameter Scopes



Modular

Parameter Domain Computations 🎱

Modular

Consequences

What are the consequences of allowing computation on parameters?

Type-checking requires determining type equality

Recall that types can be parameterized.

- Dependent Types - types depending on non-type values

Type equality requires parameter equality

When are two parameter expressions “equal”?

Different ways to write the same Type:

```
!my_dialect.array<3 + 1>
!my_dialect.array<2 * 2>
!my_dialect.array<pow(2, 2)>
...
```

Modular

Evaluation?

Evaluation is the “ultimate” judge of identity.

- Interpret the parameter expression until an irreducible value is reached

Cons:

- Slow
- Requires fully concrete expressions, which requires instantiation

Different ways to write the same Type:

```
!my_dialect.array<3 + 1>
!my_dialect.array<2 * 2>
!my_dialect.array<pow(2, 2)>
...
```

All Evaluate to:

```
!my_dialect.array<4>
```

Un-evaluatable expressions:

```
!my_dialect.array<x + y>
!my_dialect.array<y + x>
!my_dialect.array<x + 0 + y * 1>
```

Canonicalization

Canonicalize expressions into a “normal” form.

- Goal: All expressions that represent the same computation canonicalize into the same form

Requires a powerful expression canonicalizer

Complement with user-annotation to bridge any gaps

Different ways to write the same Type:

```
!my_dialect.array<x + y>
!my_dialect.array<y + x>
!my_dialect.array<x + 0 + y * 1>
...
```

All Canonicalize to:

```
!my_dialect.array<x + y>
```

Modular

Parameter Representation

Expressions in Mojo have two possible IR representations:

- As an Op:
    - Produces ssa values
- As a Typed Attribute:
    - Produces parameter values

Parameter expressions exist in attribute form

Pros:

- Parameter equality reduces to pointer equality
- Smaller memory footprint & CoW
- Concise inline representation

Source

```
(4 - 2) * 3
```

Op Repr.

```
%0 = index.constant 4 : index
%1 = index.constant 2 : index
%2 = index.constant 3 : index
%3 = index.sub %0, %1 : index
%4 = index.mul %3, %2 : index
```

Attribute Repr.

```
               MulNode
              /      \
         SubNode    ConstNode(3)
         /     \
ConstNode(4)  ConstNode(2)
```

Function References

Attribute expressions can still invoke user-declared functions

This completes the loop:

- Meta-Code is Code
- Code is Meta-Code

Functions can choose to participate in canonicalization, or stay symbolic

Given

```
fn pow(base: Int, exp: Int) -> Int:
  ...
```

Source

```
(4 - 2) * pow(2, 3)
```

Attribute Repr.

```
                 MulNode
                 /     \
          SubNode   CallNode(@pow,2,3)
          /      \
ConstNode(4)  ConstNode(2)
```

What Does This Mean in Practice?

Modular

Unified Libraries

Massively flattens learning curve

Write rich & expressive libraries to express meta-programming logic

Directly test library APIs at runtime:

- Deal with runtime failures instead of compile-time failures
- Debuggers work!

```
# Comptime layout algebra
alias TileType[*tile_sizes: Int] =
  LayoutTensor[
    dtype,
    Self._compute_tile_layout[
      *tile_sizes,
    ]()[0],
    origin,
    address_space=address_space,
    element_layout=element_layout,
    layout_int_type=layout_int_type,
    linear_idx_type=linear_idx_type,
    masked = masked or
      _tile_is_masked[layout, *tile_sizes](),
    alignment=alignment,
  ]

fn tile[
  *tile_sizes: Int
](self, *tile_coords: Int) ->
                 self.TileType[*tile_sizes]:

  alias num_tiles =
    stdlib.builtin.variadic_size(tile_sizes)
  alias _tiled_layout =
    Self._compute_tile_layout[*tile_sizes]()
  var offset = 0
  var runtime_shape =
    tile_type.RuntimeLayoutType.ShapeType()
  var runtime_stride =
    tile_type.RuntimeLayoutType.StrideType()
  ...
```

Opt-In Static Specialization

Choose how much of your data structure is static vs dynamic

- Shift info from dynamic to static for more safety guarantees
- Shift info from static to dynamic for quick prototyping and dynamic behavior

Remember: You can reuse the same libraries as you shift data between the variable-domain and the parameter-domain

More Dynamic

```
# Dynamic size & dynamic elements
struct List:
 var size: Int
 var elems: UnsafePointer[Int]
```

```
# Static size & dynamic elements
struct SizedList[size: Int]:
 var elems: UnsafePointer[Int]
```

```
# Dynamic size & static element
struct SplatList[elem: Int]:
 var size: Int
```

```
# Static size & static elements
struct StaticList[size: Int, *elems: Int]:
  # No fields.
```

More Static

What do you prefer?

Types checked at instantiation time

Building CUDA object
examples/46_depthwise_simt_conv2dfprop/CMakeFiles/46_depthwise_simt_conv2dfprop.dir/depthwise_simt_conv2dfprop.cu.o
/home/clattner/cutlass/include/cutlass/conv/kernel/direct_convolution.h(95): error: incomplete type is not allowed detected during:
instantiation of class
"cutlass::conv::kernel::DirectConvolutionParams<Mma_, Epilogue_, ThreadblockSwizzle_, ConvOperator, Arguments_, ConvOutputIteratorParameter_, ConvProblemSize_, GroupMode_, ThreadBlockOutputShape_> [with
Mma_=cutlass::conv::threadblock::DepthwiseFpropDirectConvMultipleStage<ThreadblockShape,
cutlass::conv::threadblock::DepthwiseFpropActivationDirect2dConvTileAccessIteratorFixedStrideDilation<cutlass::MatrixShape<64, 64>,
ThreadBlockOutputShape, StrideShape, DilationShape,
cutlass::conv::TensorNHWCShape<1, 10, 10, 64>, ElementInputA,
LayoutInputA,
cutlass::transform::PitchLinearStripminedThreadMap<cutlass::PitchLinearShape<64, 100>, 128, 4>, cutlass::AlignedArray<ElementInputA, 4, 16>>,
cutlass::transform::threadblock::RegularTileAccessIteratorDirectConv<cutlass::MatrixShape<100, 64>, ElementInputA, cutlass::layout::RowMajor, 0,
cutlass::transform::PitchLinearStripminedThreadMap<cutlass::PitchLinearShape<64, 100>, 128, 4>, false, 16>, cutlass::arch::CacheOperation::Global,
cutlass::conv::threadblock::DepthwiseFpropFilterDirectConvTileAccessIter

Types checked without instantiating

```
fn process_prime[x: Int where is_prime(x)]():
   ...

fn main():
 alias value = get_int()

 # `value` not always prime.
 process_prime[value]()

 @parameter
 if is_prime(value):
   # This is OK.
   process_prime[value]()
```

```
$ mojo test.mojo
test.mojo:9:25: error: invalid call to 'process_prime': unable to satisfy constraint
    process_prime[value]()
    ~~~~~~~~~~~~~~~~~~~~^~
test.mojo:1:31: note: constraint declared here
fn process_prime[x: Int where is_prime(x)]():
                              ^
```

All made possible by the power of MLIR

Thank You

Modular