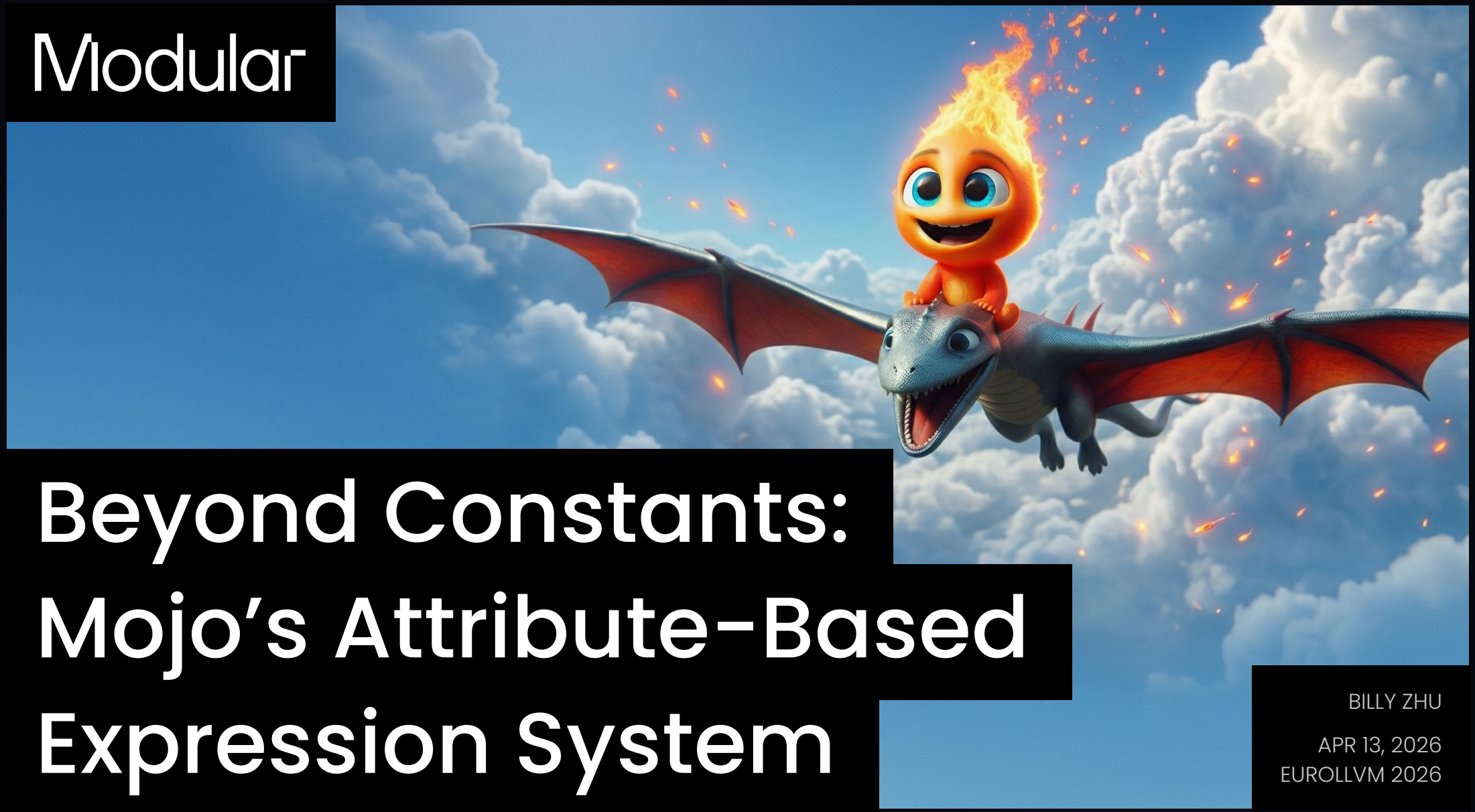
Modular



Beyond Constants: Mojo's Attribute-Based Expression System

BILLY ZHU

APR 13, 2026
EUROLLVM 2026

Agenda

01 Brief Mojo Primer

02 The Attribute Language

03 Expression Evaluation

04 How Well Does it Work?

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

How: Powerful, type-safe **meta-programming**

🔥Mojo

```
def mandelbrot_kernel[
    width: Int  # SIMD Width
](c: ComplexSIMD[float, width]) ->
    SIMD[int, width]:
    z = ComplexSIMD[float, width](0, 0)
    iters = SIMD[DType.index, width](0)
    for i in range(MAX_ITERS):
        mask = z.squared_norm() <= 4
        iters = mask.select(iters + 1, iters)
        z = z.squared_add(c)
    return iters
```

Meta-Programming

Typed-Checked Generics:
- Write once, checked once

Dependent Types: Parameterize anything
- Parameterize functions, types, and even arbitrary expressions
- Parameterize on types and user-defined structures

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

🔥Mojo

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

  @staticmethod
  def col_major(shape: List[Int]) -> Layout:
    return ...

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

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

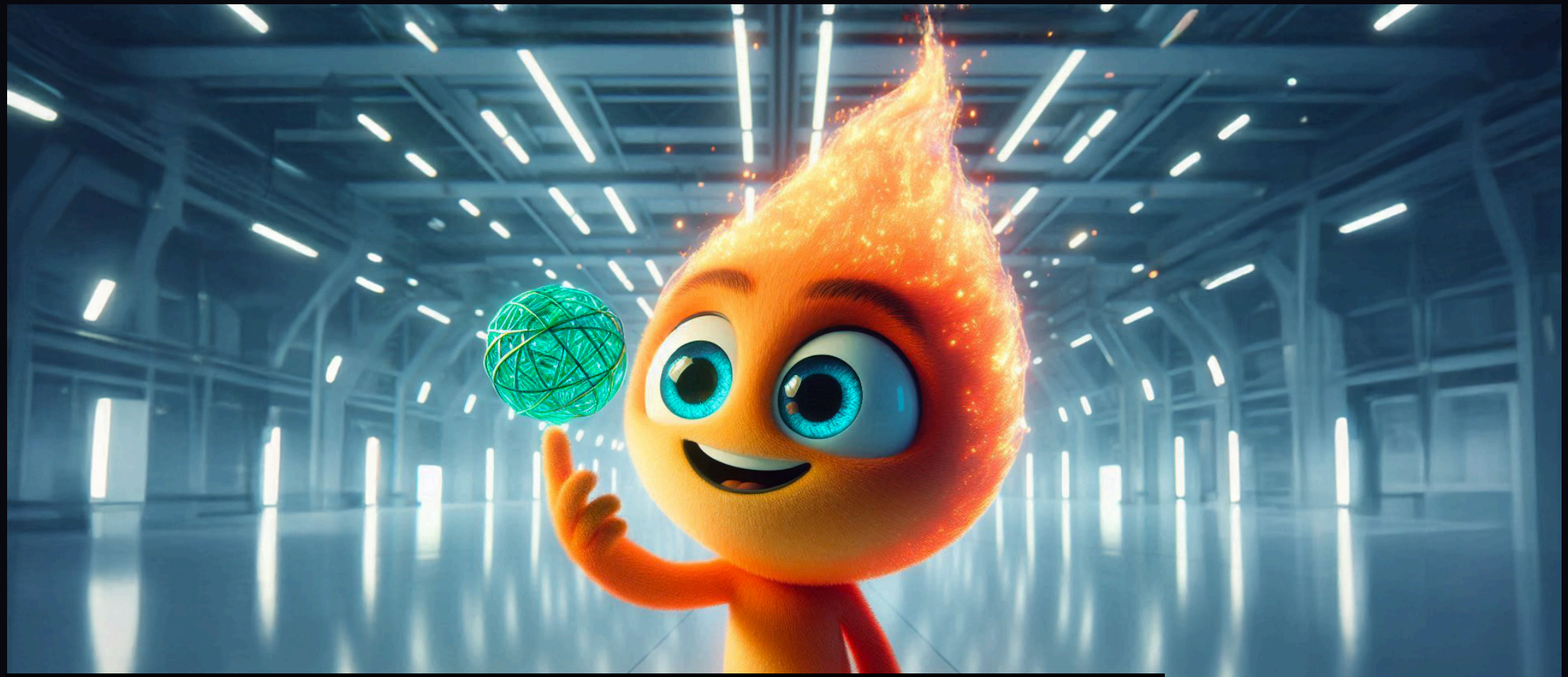
Parametric MLIR

Two-layer Parametric IR in MLIR:

- **Base Layer**: Operations with typed-holes
  - Part of the “execution-time” logic
- **Meta Layer**: Typed-Expressions for filling holes
  - Used to “configure” some base IR operation

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>
  ...
}
```



The Attribute Language

Modular

MLIR Attributes

> "Attributes are the mechanism for specifying **constant data on operations** in places where a variable is never allowed - e.g. the comparison predicate of a cmpi operation."
>
> — MLIR LangRef

MLIR Operations defined as parametric "templates"

- E.g. The `cmpi` op is parameterized by an attribute that represents the predicate to use

Definition

TableGen

```
def Arith_CmpIOp
    : Arith_CompareOp<"cmpi", ...> {
  let summary = "integer comparison";
  ...
  let arguments = (ins
    Arith_CmpIPredicateAttr:$predicate,
    SignlessIntegerOrIndexLike:$lhs,
    SignlessIntegerOrIndexLike:$rhs);
}
```

Example

MLIR

```
%x = arith.cmpi eq, %lhs, %rhs
       : vector<4xi64>
```

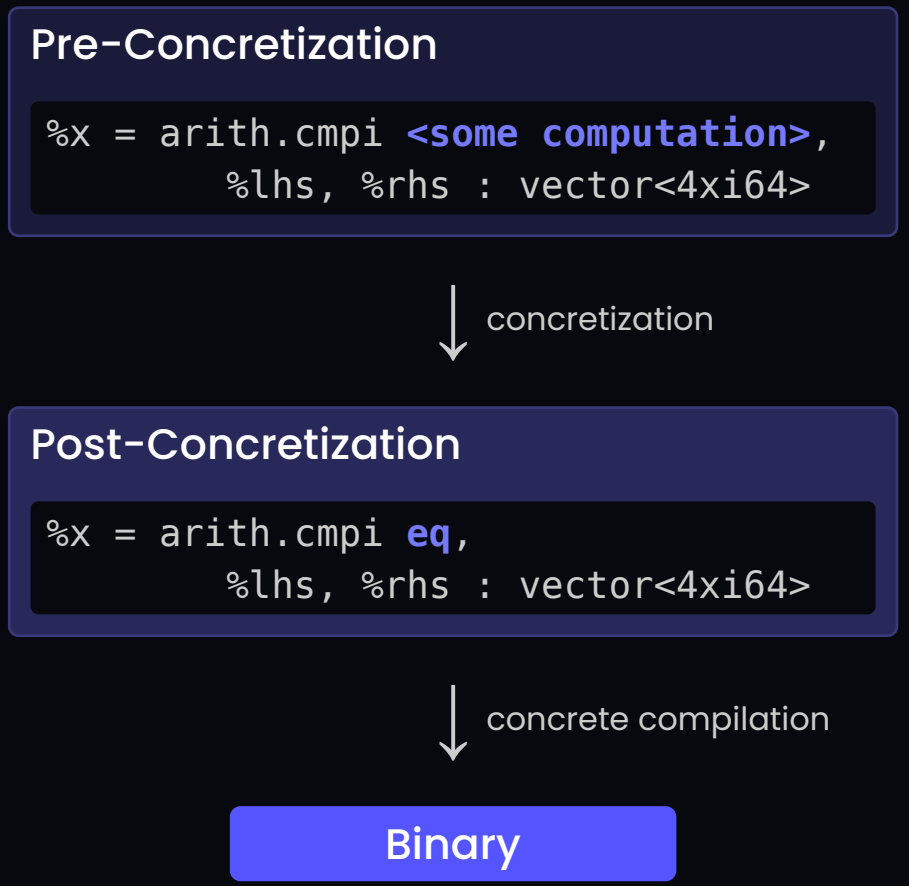
Eventually Constant Attributes

The missing piece:

- Specify the attribute not with a “constant” attribute, but with **some computation** that results in such an attribute

Solution: An expression language that produces attributes for parameterized MLIR Ops

- Not “immediately” constant data, but “eventually” constant data
- Becomes constant over the course of compilation



Mojo's Attribute Language

Attributes can themselves be parameterized by Attributes

Use Attributes to express computation "nodes"

Components:
- Types
- Abstractions
- Extensions

Source

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

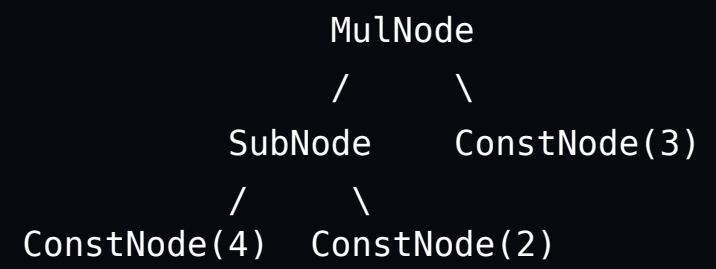
🔥Mojo

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
```

MLIR

Attribute Repr.



Typed Expressions

All nodes are typed:

- `TypedAttr` MLIR interface provides `Type getType()`

Types enforce expression well-formedness:

- A leaf attr defines its type intrinsically
- An operator attr defines:
  - Its operand types
  - Its own type
    - May or may not depend on its operands

Operand types are checked via Attribute verifier on construction

Already powerful enough to define grammar rules for a programming language

TableGen

```
def MyStringAttr
    : MyDialect_Attr<"string"> {
  let parameters = (ins
    StringRefParameter<"">:$value);

  let extraClassDeclaration =
    [{ Type getType() const; }];
}

def MyStringConcatAttr
    : MyDialect_Attr<"string.concat"> {
  let parameters = (ins
    "TypedAttr":$lhs,
    "TypedAttr":$rhs);

  let extraClassDeclaration =
    [{ Type getType() const; }];

  let genVerifyDecl = 1;
}
```

Abstractions

`GeneratorType`

- The type of lambdas $(A \to B)$
- … with:
  - Multiple arguments $(A_0, A_1, A_2 \to B)$
  - Dependent abstractions $(\Pi_{x:A} B(x))$

Specify:

- Input parameter types
  - Later parameter types can depend on earlier parameter values
- Result type
  - Can depend on input parameters

```
comptime incr[x: Int] = x + 1
# type: !gen<[x: Int] Int>


comptime add[x: Int, y: Int] = x + y
# type: !gen<[x: Int, y: Int] Int>


comptime id[T: AnyType, x: T] = x
# type: !gen<[T: AnyType, x: T] T>
```

Binders

Inside a `GeneratorType`, reference arguments via De Bruijn indices

`ParamIndexRefAttr`:

- depth: size_t
- index: size_t
- type: Type

Unique representation: Avoid variable naming problems

```
comptime incr[x: Int] = x + 1
# type: !gen<[Int] Int>


comptime add[x: Int, y: Int] = x + y
# type: !gen<[Int, Int] Int>


comptime id[T: AnyType, x: T] = x
# type: !gen<[AnyType, *(0,0)] *(0,0)>
```

Binder Depths

Inside a `GeneratorType`, reference arguments via De Bruijn indices

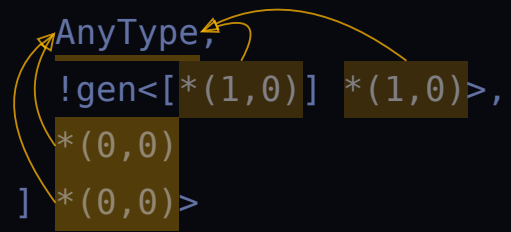
`ParamIndexRefAttr`:

- depth: size_t
- index: size_t
- type: Type

Unique representation: Avoid variable naming problems

```mojo
comptime id[T: AnyType, x: T] = x
# type: !gen<[AnyType, *(0,0)] *(0,0)>

comptime id2[
  T: AnyType,
  f: type_of(id[T]),
  v: T
]: T = f[v]
# type: !gen<[
#         AnyType,
#         !gen<[*(1,0)] *(1,0)>,
#         *(0,0)
#       ] *(0,0)>
```



Abstractions

`GeneratorAttr`

- A constructor of `GeneratorType`

Specify:

- Input parameter types
- Result type
- Body attribute

```
comptime incr[x: Int] = x + 1
# type : !gen<[Int] Int>
# value: #gen<add<*(0,0), 1>>

comptime add[x: Int, y: Int] = x + y
# type : !gen<[Int, Int] Int>
# value: #gen<add<*(0,0), *(0,1)>>

comptime id[T: AnyType, x: T] = x
# type : !gen<[AnyType, *(0,0)] *(0,0)>

# value: #gen<*(0,0)>
```

Leveraging MLIR Extensibility

Expand our core calculus with extensions, organized by MLIR dialects

A dialect can define:
- Custom Types
- Type constructors/eliminators as Attributes

Examples:
- From upstream:
  - `index` Type
  - `IntegerAttr`: constants for `index`
- Locally:
  - `simd` Type
  - `SIMDAttr`: constants for `simd`
  - `SIMDCmpAttr`: `(simd<dtype, size>, simd<dtype, size>) -> simd<bool, size>`

Expression Evaluation

Modular

Evaluation Exercise

Fold:

- `1 + (2 * 3)`
- `add<1, mul<2, 3>>`

Want to reduce it into the constant attribute `7`

Evaluation Exercise

Fold:

- `1 + (2 * 3)`
- `add<1, mul<2, 3>>`

Want to reduce it into the constant attribute `7`

Rewrite rule in the constructors

- If both operands are constants, emit a constant attribute instead of an operator attribute

Evaluation Exercise 2

Given:

- `comptime add_gen[x: index, y: index] = x + y`
- `#add_gen: !gen<<index, index> index> = #gen<add<*(0,0), *(0,1)>>`

Fold:

- `add_gen[2, 3]`
- `#bind_params<#add_gen, 2, 3>`

Evaluation Exercise 2

Given:

- `comptime add_gen[x: index, y: index] = x + y`
- `#add_gen: !gen<<index, index> index> = #gen<add<*(0,0), *(0,1)>>`

Fold:

- `add_gen[2, 3]`
- `#bind_params<#add_gen, 2, 3>`

Reach for the AttrTypeReplacer:

- On `ParamIndexRefAttr`, replace with the corresponding input attribute
- The `add` attribute constructor will automatically kick in

All good?

Depth-Aware

AttrTypeReplacer:
- Powerful: Generic over all attributes
- Efficient: Caches replacements

Problem: Binder replacement is context-dependent

Depth-aware walk:
- Track how many abstraction scopes we have entered
  - None: Replace `*(0, i)`
  - 2: Replace `*(2, i)`

Depth-aware replacement cache:
- Key on walk depth

~30% attr/types hit the cache

Context-Aware

Context external to the attribute expression

Symbol lookup:

- Symbol references are also just attributes

Use cases:

- Invoke a function op
- Lookup witness table for a type's conformance to a trait

MLIR

```
// Symbol op
kgen.generator @ceildiv(
  x: Float32, y: Float32) -> Float32 {
    ...
}

// Attribute Expression
#attr = apply<@ceildiv, 5.0, 2.0>
  : scalar<f32>
```

EvaluationContext

Custom evaluation context for looking up external information

Context-Aware evaluation AttrInterface:

- `evaluateWithContext(EvaluationContext&).`

EvaluationContext interface:

- Lookup functions
- Lookup type definitions

Enables:

- More efficient repr.
- Invoke function ops (base IR) from attributes (meta IR)
  - Wednesday 11am: *Mojo Compile-time Interpreter in MLIR* - Weiwei Chen

MLIR

```
// Symbol op
kgen.generator @ceildiv(
  x: Float32, y: Float32) -> Float32 {
    ...
}

// Attribute Expression
#attr = apply<@ceildiv, 5.0, 2.0>
  : scalar<f32>
```

How Well Does it Work?

Modular

Compile Time

Folding:

- Construction-time folding for non-leaf attrs
- Explicit folding when substituting parameters

mojo build time

Memory Footprint

Immortal Attributes & Types

max RSS


Thank You
Acknowledgement: Mojo Compiler Team @ Modular