Parameterization: compile-time metaprogramming

Many languages have facilities for *metaprogramming*: that is, for writing code that generates or modifies code. Python has facilities for dynamic metaprogramming: features like decorators, metaclasses, and many more. These features make Python very flexible and productive, but since they're dynamic, they come with runtime overhead. Other languages have static or compile-time metaprogramming features, like C preprocessor macros and C++ templates. These can be limiting and hard to use.

To support Modular's work in AI, Mojo aims to provide powerful, easy-to-use metaprogramming with zero runtime cost. This compile-time metaprogramming uses the same language as runtime programs, so you don't have to learn a new language—just a few new features.

The main new feature is *parameters*. You can think of a parameter as a compile-time variable that becomes a runtime constant. This usage of "parameter" is probably different from what you're used to from other languages, where "parameter" and "argument" are often used interchangeably. In Mojo, "parameter" and "parameter expression" refer to compile-time values, and "argument" and "expression" refer to runtime values.

In Mojo, you can add parameters to a struct or function. You can also define named parameter expressions—aliases—that you can use as runtime constants.

Parameterized functions

To define a *parameterized function*, add parameters in square brackets ahead of the argument list. Each parameter is formatted just like an argument: a parameter name, followed by a colon and a type (which is required). In the following example, the function has a single parameter, `count` of type `Int`.

```
fn repeat[count: Int](msg: String):
    @parameter
    for i in range(count):
        print(msg)
```

The `@parameter` directive shown here causes the `for` loop to be evaluated at compile time. The directive only works if the loop limits are compile-time constants. Since `count` is a parameter, `range(count)` can be calculated at compile time.

Calling a parameterized function, you provide values for the parameters, just like function arguments:

```
repeat[3]("Hello")
```

```
Hello
Hello
Hello
```

The compiler resolves the parameter values during compilation, and creates a concrete version of the `repeat[]()` function for each unique parameter value. After resolving the parameter values and unrolling the loop, the `repeat[3]()` function would be roughly equivalent to this:

```
fn repeat_3(msg: String):
    print(msg)
    print(msg)
    print(msg)
```

Note

This doesn't represent actual code generated by the compiler. By the time parameters are resolved, Mojo code has already been transformed to an intermediate representation in MLIR.

If the compiler can't resolve all parameter values to constant values, compilation fails.

Parameters and generics

"Generics" refers to functions that can act on multiple types of values, or containers that can hold multiple types of values. For example, `List`, can hold different types of values, so you can have a list of `Int` values, or a list of `String` values).

In Mojo, generics use parameters to specify types. For example, `List` takes a type parameter, so a vector of integers is written `List[Int]`. So all generics use parameters, but **not** everything that uses parameters is a generic.

For example, the `repeat[]()` function in the previous section includes parameter of type `Int`, and an argument of type `String`. It's parameterized, but not generic. A generic function or struct is parameterized on *type*. For example, we could rewrite `repeat[]()` to take any type of argument that conforms to the `Stringable` trait:

```
fn repeat[MsgType: Stringable, count: Int](msg: MsgType):
    @parameter
    for i in range(count):
        print(str(msg))
```

```
# Must use keyword parameter for `count`
repeat[count=2](42)
```

```
42
42
```

This updated function takes any `Stringable` type, so you can pass it an `Int`, `String`, or `Bool` value.

You can't pass the `count` as a positional keyword without also specifying `MsgType`. You can put `//` after `MsgType` to specify that it's always inferred by the argument. Now you can pass the following parameter `count` positionally:

```
fn repeat[MsgType: Stringable, //, count: Int](msg: MsgType):
    @parameter
    for i in range(count):
        print(str(msg))

# MsgType is always inferred, so first positional keyword `2` is passed to `count`
repeat[2](42)
```

```
42
42
```

Mojo's support for generics is still early. You can write generic functions like this using traits and parameters. You can also write generic collections like `List` and `Dict`. If you're interested in learning how these types work, you can find the source code for the standard library collection types on GitHub.

Parameterized structs

You can also add parameters to structs. You can use parameterized structs to build generic collections. For example, a generic array type might include code like this:

```
from memory.unsafe_pointer import UnsafePointer

struct GenericArray[ElementType: CollectionElement]:
    var data: UnsafePointer[ElementType]
    var size: Int

    fn __init__(inout self, *elements: ElementType):
        self.size = len(elements)
        self.data = UnsafePointer[ElementType].alloc(self.size)
        for i in range(self.size):
            (self.data + i).init_pointee_move(elements[i])
    fn __del__(owned self):
        for i in range(self.size):
            (self.data + i).destroy_pointee()
        self.data.free()

    fn __getitem__(self, i: Int) raises -> ref [__lifetime_of(self)] ElementType:
        if (i < self.size):
            return self.data[i]
        else:
            raise Error("Out of bounds")
```

This struct has a single parameter, `ElementType`, which is a placeholder for the data type you want to store in the array, sometimes called a *type parameter*. `ElementType` is typed as `CollectionElement`, which is a trait representing any type that can be copied and moved.

As with parameterized functions, you need to pass in parameter values when you use a parameterized struct. In this case, when you create an instance of `GenericArray`, you need to specify the type you want to store, like `Int`, or `Float64`. (This is a little confusing, because the *parameter value* you're passing in this case is a *type*. That's OK: a Mojo type is a valid compile-time value.)

You'll see that `ElementType` is used throughout the struct where you'd usually see a type name. For example, as the formal type for the `elements` in the constructor, and the return type of the `__getitem__()` method.

Here's an example of using `GenericArray`:

```
var array = GenericArray[Int](1, 2, 3, 4)
for i in range(array.size):
    print(array[i], end=" ")
```

```
1 2 3 4
```

A parameterized struct can use the `Self` type to represent a concrete instance of the struct (that is, with all its parameters specified). For example, you could add a static factory method to `GenericArray` with the following signature:

```
struct GenericArray[ElementType: CollectionElement]:
    ...

    @staticmethod
    fn splat(count: Int, value: ElementType) -> Self:
        # Create a new array with count instances of the given value
```

Here, `Self` is equivalent to writing `GenericArray[ElementType]`. That is, you can call the `splat()` method like this:

```
GenericArray[Float64].splat(8, 0)
```

The method returns an instance of `GenericArray[Float64]`.

Conditional conformance

When creating a generic struct, you might want to define some methods that require extra features. For example, consider a collection like `GenericArray` that holds instances of `CollectionElement`. The `CollectionElement` trait only requires that the stored data type be copyable and movable. This imposes a lot of limitations: you can't implement a `sort()` method because you can't guarantee that the stored type supports the comparison operators; you can't write a useful `__str__()` or `__repr__()` dunder method because you can't guarantee that the stored type supports conversion to a string.

The answer to these issues is *conditional conformance*, which lets you define a method that requires additional features. You do this by defining the `self` value that has a more specific bound on one or more of its parameters.

For example, the following code defines a `Container` type that holds an instance of `CollectionElement`. It also defines a `__str__()` method that can only be called if the stored `ElementType` conforms to `StringableCollectionElement`:

```
@value
struct Container[ElementType: CollectionElement]:
    var element: ElementType

    def __str__[StrElementType: StringableCollectionElement, //](
            self: Container[StrElementType]) -> String:
        return str(self.element)

def use_container():
    float_container = Container(5)
    string_container = Container("Hello")
    print(float_container.__str__())
    print(string_container.__str__())

use_container()
```

```
5
Hello
```

Note the signature of the `__str__()` method, which declares the `self` argument with a more specific type. Specifically, it declares that it takes a `Container` with an `ElementType` that conforms to the `StringableCollectionElement` trait.

```
def __str__[StrElementType: StringableCollectionElement, //](
        self: Container[StrElementType]) -> String:
```

This trait must be a superset of `ElementType`'s original trait: for example, `StringableCollectionElement` inherits from `CollectionElement`, so it includes all of requirements of the original trait.

Note that the `use_container()` function calls the `__str__()` method directly, rather than calling `str(float_container)`. One current limitation of conditional conformance is that Mojo can't recognize the struct `Container[Int]` as conforming to `Stringable`, even though the `__str__()` method is implemented for any `ElementType` that's also `Stringable`.

Case study: the SIMD type

For a real-world example of a parameterized type, let's look at the `SIMD` type from Mojo's standard library.

Single instruction, multiple data (SIMD) is a parallel processing technology built into many modern CPUs, GPUs, and custom accelerators. SIMD allows you to perform a single operation on multiple pieces of data at once. For example, if you want to take the square root of each element in an array, you can use SIMD to parallelize the work.

Processors implement SIMD using low-level vector registers in hardware that hold multiple instances of a scalar data type. In order to use the SIMD instructions on these processors, the data must be shaped into the proper SIMD width (data type) and length (vector size). Processors may support 512-bit or longer SIMD vectors, and support many data types from 8-bit integers to 64-bit floating point numbers, so it's not practical to define all of the possible SIMD variations.

Mojo's `SIMD` type (defined as a struct) exposes the common SIMD operations through its methods, and makes the SIMD data type and size values parametric. This allows you to directly map your data to the SIMD vectors on any hardware.

Here's a cut-down (non-functional) version of Mojo's `SIMD` type definition:

```
struct SIMD[type: DType, size: Int]:
    var value: … # Some low-level MLIR stuff here

    # Create a new SIMD from a number of scalars
    fn __init__(inout self, *elems: SIMD[type, 1]):  ...

    # Fill a SIMD with a duplicated scalar value.
    @staticmethod
    fn splat(x: SIMD[type, 1]) -> SIMD[type, size]: ...

    # Cast the elements of the SIMD to a different elt type.
    fn cast[target: DType](self) -> SIMD[target, size]: ...
    # Many standard operators are supported.
    fn __add__(self, rhs: Self) -> Self: ...
```

So you can create and use a SIMD vector like this:

```
var vector = SIMD[DType.int16, 4](1, 2, 3, 4)
vector = vector * vector
for i in range(4):
    print(vector[i], end=" ")
```

```
1 4 9 16
```

As you can see, a simple arithmetic operator like `*` applied to a pair of `SIMD` vector operates on the corresponding elements in each vector.

Defining each SIMD variant with parameters is great for code reuse because the `SIMD` type can express all the different vector variants statically, instead of requiring the language to pre-define every variant.

Because `SIMD` is a parameterized type, the `self` argument in its functions carries those parameters—the full type name is `SIMD[type, size]`. Although it's valid to write this out (as shown in the return type of `splat()`), this can be verbose, so we recommend using the `Self` type (from PEP673) like the `__add__` example does.

Overloading on parameters

Functions and methods can be overloaded on their parameter signatures. The overload resolution logic filters for candidates according to the following rules, in order of precedence:

1. Candidates with the minimal number of implicit conversions (in both arguments and parameters).
2. Candidates without variadic arguments.
3. Candidates without variadic parameters.
4. Candidates with the shortest parameter signature.
5. Non-`@staticmethod` candidates (over `@staticmethod` ones, if available).

If there is more than one candidate after applying these rules, the overload resolution fails. For example:

```
@register_passable("trivial")
struct MyInt:
    """A type that is implicitly convertible to `Int`."""
    var value: Int

    @always_inline("nodebug")
    fn __init__(inout self, _a: Int):
        self.value = _a

fn foo[x: MyInt, a: Int]():
    print("foo[x: MyInt, a: Int]()")

fn foo[x: MyInt, y: MyInt]():
    print("foo[x: MyInt, y: MyInt]()")

fn bar[a: Int](b: Int):
    print("bar[a: Int](b: Int)")

fn bar[a: Int](*b: Int):
    print("bar[a: Int](*b: Int)")

fn bar[*a: Int](b: Int):
    print("bar[*a: Int](b: Int)")

fn parameter_overloads[a: Int, b: Int, x: MyInt]():
    # `foo[x: MyInt, a: Int]()` is called because it requires no implicit
    # conversions, whereas `foo[x: MyInt, y: MyInt]()` requires one.
    foo[x, a]()

    # `bar[a: Int](b: Int)` is called because it does not have variadic
    # arguments or parameters.
    bar[a](b)

    # `bar[*a: Int](b: Int)` is called because it has variadic parameters.
    bar[a, a, a](b)

parameter_overloads[1, 2, MyInt(3)]()

struct MyStruct:
    fn __init__(inout self):
        pass

    fn foo(inout self):
        print("calling instance method")

    @staticmethod
    fn foo():
        print("calling static method")

fn test_static_overload():
    var a = MyStruct()
    # `foo(inout self)` takes precedence over a static method.
    a.foo()
```

```
foo[x: MyInt, a: Int]()
bar[a: Int](b: Int)
bar[*a: Int](b: Int)
```

Using parameterized types and functions

You can use parametric types and functions by passing values to the parameters in square brackets. For example, for the `SIMD` type above, `type` specifies the data type and `size` specifies the length of the SIMD vector (it must be a power of 2):

```
# Make a vector of 4 floats.
var small_vec = SIMD[DType.float32, 4](1.0, 2.0, 3.0, 4.0)

# Make a big vector containing 1.0 in float16 format.
var big_vec = SIMD[DType.float16, 32](1.0)

# Do some math and convert the elements to float32.
var bigger_vec = (big_vec+big_vec).cast[DType.float32]()

# You can write types out explicitly if you want of course.
var bigger_vec2 : SIMD[DType.float32, 32] = bigger_vec

print('small_vec type:', small_vec.element_type, 'length:', len(small_vec))
print('bigger_vec2 type:', bigger_vec2.element_type, 'length:', len(bigger_vec2))
```

```
small_vec type: float32 length: 4
bigger_vec2 type: float32 length: 32
```

Note that the `cast()` method also needs a parameter to specify the type you want from the cast (the method definition above expects a `target` parametric value). Thus, just as the `SIMD` struct is a generic type definition, the `cast()` method is a generic method definition. At compile time, the compiler creates a concrete version of the `cast()` method with the target parameter bound to `DType.float32`.

The code above shows the use of concrete types (that is, the parameters are all bound to known values). But the major power of parameters comes from the ability to define parametric algorithms and types (code that uses the parameter values). For example, here's how to define a parametric algorithm with `SIMD` that is type- and width-agnostic:

```
from math import sqrt

fn rsqrt[dt: DType, width: Int](x: SIMD[dt, width]) -> SIMD[dt, width]:
    return 1 / sqrt(x)

var v = SIMD[DType.float16, 4](42)
print(rsqrt(v))
```

```
[0.154296875, 0.154296875, 0.154296875, 0.154296875]
```

Notice that the `x` argument is actually a `SIMD` type based on the function parameters. The runtime program can use the value of the parameters, because the parameters are resolved at compile-time before they are needed by the runtime program (but compile-time parameter expressions cannot use runtime values).

Parameter inference

The Mojo compiler can often *infer* parameter values, so you don't always have to specify them. For example, you can call the `rsqrt()` function defined above without any parameters:

```
var v = SIMD[DType.float16, 4](33)
print(rsqrt(v))
```

```
[0.174072265625, 0.174072265625, 0.174072265625, 0.174072265625]
```

The compiler infers its parameters based on the parametric `v` value passed into it, as if you wrote `rsqrt[DType.float16, 4](v)` explicitly.

Mojo can also infer the values of struct parameters from the arguments passed to a constructor or static method.

For example, consider the following struct:

```
@value
struct One[Type: StringableCollectionElement]:
    var value: Type

    fn __init__(inout self, value: Type):
        self.value = value

def use_one():
    s1 = One(123)
    s2 = One("Hello")
```

Note that you can create an instance of `One` without specifying the `Type` parameter—Mojo can infer it from the `value` argument.

You can also infer parameters from a parameterized type passed to a constructor or static method:

```
struct Two[Type: StringableCollectionElement]:
    var val1: Type
    var val2: Type

    fn __init__(inout self, one: One[Type], another: One[Type]):
        self.val1 = one.value
        self.val2 = another.value
        print(str(self.val1), str(self.val2))

    @staticmethod
    fn fire(thing1: One[Type], thing2: One[Type]):
        print("🔥", str(thing1.value), str(thing2.value))

def use_two():
    s3 = Two(One("infer"), One("me"))
    Two.fire(One(1), One(2))

use_two()
```

```
infer me
🔥 1 2
```

`Two` takes a `Type` parameter, and its constructor takes values of type `One[Type]`. When constructing an instance of `Two`, you don't need to specify the `Type` parameter, since it can be inferred from the arguments.

Similarly, the static `fire()` method takes values of type `One[Type]`, so Mojo can infer the `Type` value at compile time.

Note

If you're familiar with C++, you may recognize this as similar to Class Template Argument Deduction (CTAD).

Optional parameters and keyword parameters

Just as you can specify optional arguments in function signatures, you can also define an optional *parameter* by giving it a default value.

You can also pass parameters by keyword, just like you can use keyword arguments. For a function or struct with multiple optional parameters, using keywords allows you to pass only the parameters you want to specify, regardless of their position in the function signature.

For example, here's a function with two parameters, each with a default value:

```
fn speak[a: Int = 3, msg: StringLiteral = "woof"]():
    print(msg, a)

fn use_defaults() raises:
    speak()             # prints 'woof 3'
    speak[5]()          # prints 'woof 5'
    speak[7, "meow"]()  # prints 'meow 7'
    speak[msg="baaa"]() # prints 'baaa 3'
```

Recall that when a parametric function is called, Mojo can infer the parameter values. That is, it can use the parameter values attached to an argument value (see the `sqrt[]()` example above). If the parametric function also has a default value defined, then the inferred parameter type takes precedence.

For example, in the following code, we update the parametric `speak[]()` function to take an argument with a parametric type. Although the function has a default parameter value for `a`, Mojo instead uses the inferred `a` parameter value from the `bar` argument (as written, the default `a` value can never be used, but this is just for demonstration purposes):

```
@value
struct Bar[v: Int]:
    pass

fn speak[a: Int = 3, msg: StringLiteral = "woof"](bar: Bar[a]):
    print(msg, a)

fn use_inferred():
    speak(Bar[9]())  # prints 'woof 9'
```

As mentioned above, you can also use optional parameters and keyword parameters in a struct:

```
struct KwParamStruct[greeting: String = "Hello", name: String = "🔥mojo🔥"]:
    fn __init__(inout self):
        print(greeting, name)

fn use_kw_params():
    var a = KwParamStruct[]()                 # prints 'Hello 🔥mojo🔥'
    var b = KwParamStruct[name="World"]()     # prints 'Hello World'
    var c = KwParamStruct[greeting="Hola"]()  # prints 'Hola 🔥mojo🔥'
```

Note

Mojo supports positional-only and keyword-only parameters, following the same rules as positional-only and keyword-only arguments.

Infer-only parameters

Sometimes you need to declare functions where parameters depend on other parameters. Because the signature is processed left to right, a parameter can only *depend* on a parameter earlier in the parameter list. For example:

```
fn dependent_type[dtype: DType, value: Scalar[dtype]]():
    print("Value: ", value)
    print("Value is floating-point: ", dtype.is_floating_point())

dependent_type[DType.float64, Float64(2.2)]()
```

```
Value:  2.2000000000000002
Value is floating-point:  True
```

You can't reverse the position of the `dtype` and `value` parameters, because `value` depends on `dtype`. However, because `dtype` is a required parameter, you can't leave it out of the parameter list and let Mojo infer it from `value`:

```
dependent_type[Float64(2.2)]() # Error!
```

Infer-only parameters are a special class of parameters that are **always** inferred from context. Infer-only parameters are placed at the **beginning** of the parameter list, set off from other parameters by the `//` sigil:

```
fn example[type: CollectionElement, //, list: List[type]]()
```

Transforming `dtype` into an infer-only parameter solves this problem:

```
fn dependent_type[dtype: DType, //, value: Scalar[dtype]]():
    print("Value: ", value)
    print("Value is floating-point: ", dtype.is_floating_point())

dependent_type[Float64(2.2)]()
```

```
Value:  2.2000000000000002
Value is floating-point:  True
```

Because infer-only parameters are declared at the beginning of the parameter list, other parameters can depend on them, and the compiler will always attempt to infer the infer-only values from bound parameters or arguments.

If the compiler can't infer the value of an infer-only parameter, compilation fails.

Variadic parameters

Mojo also supports variadic parameters, similar to Variadic arguments:

```
struct MyTensor[*dimensions: Int]:
    pass
```

Variadic parameters currently have some limitations that variadic arguments don't have:

- Variadic parameters must be homogeneous—that is, all the values must be the same type.
- The parameter type must be register-passable.
- The parameter values aren't automatically projected into a `VariadicList`, so you need to construct the list explicitly:

```
fn sum_params[*values: Int]() -> Int:
    alias list = VariadicList(values)
    var sum = 0
    for v in list:
        sum += v
    return sum
```

Variadic keyword parameters (for example, `**kwparams`) are not supported yet.

Parameter expressions are just Mojo code

A parameter expression is any code expression (such as `a+b`) that occurs where a parameter is expected. Parameter expressions support operators and function calls, just like runtime code, and all parameter types use the same type system as the runtime program (such as `Int` and `DType`).

Because parameter expressions use the same grammar and types as runtime Mojo code, you can use many "dependent type" features. For example, you might want to define a helper function to concatenate two SIMD vectors:

```
fn concat[ty: DType, len1: Int, len2: Int](
        lhs: SIMD[ty, len1], rhs: SIMD[ty, len2]) -> SIMD[ty, len1+len2]:

    var result = SIMD[ty, len1 + len2]()
    for i in range(len1):
        result[i] = SIMD[ty, 1](lhs[i])
    for j in range(len2):
        result[len1 + j] = SIMD[ty, 1](rhs[j])
    return result

var a = SIMD[DType.float32, 2](1, 2)
var x = concat(a, a)

print('result type:', x.element_type, 'length:', len(x))
```

```
result type: float32 length: 4
```

Note that the resulting length is the sum of the input vector lengths, and this is expressed with a simple `+` operation.

Powerful compile-time programming

While simple expressions are useful, sometimes you want to write imperative compile-time logic with control flow. You can even do compile-time recursion. For instance, here is an example "tree reduction" algorithm that sums all elements of a vector recursively into a scalar:

```
fn slice[ty: DType, new_size: Int, size: Int](
        x: SIMD[ty, size], offset: Int) -> SIMD[ty, new_size]:
    var result = SIMD[ty, new_size]()
    for i in range(new_size):
        result[i] = SIMD[ty, 1](x[i + offset])
    return result

fn reduce_add[ty: DType, size: Int](x: SIMD[ty, size]) -> Int:
    @parameter
    if size == 1:
        return int(x[0])
    elif size == 2:
        return int(x[0]) + int(x[1])

    # Extract the top/bottom halves, add them, sum the elements.
    alias half_size = size // 2
    var lhs = slice[ty, half_size, size](x, 0)
    var rhs = slice[ty, half_size, size](x, half_size)
    return reduce_add[ty, half_size](lhs + rhs)

var x = SIMD[DType.index, 4](1, 2, 3, 4)
print(x)
print("Elements sum:", reduce_add(x))
```

```
[1, 2, 3, 4]
Elements sum: 10
```

This makes use of the `@parameter` decorator to create a parametric if condition, which is an `if` statement that runs at compile-time. It requires that its condition be a valid parameter expression, and ensures that only the live branch of the `if` statement is compiled into the program. (This is similar to use of the `@parameter` decorator with a `for` loop shown earlier.)

Mojo types are just parameter expressions

While we've shown how you can use parameter expressions within types, type annotations can themselves be arbitrary expressions (just like in Python). Types in Mojo have a special metatype type, allowing type-parametric algorithms and functions to be defined.

For example, we can create a simplified `Array` that supports arbitrary types of elements (via the `AnyTrivialRegType` parameter):

```
from memory import UnsafePointer

struct Array[T: AnyTrivialRegType]:
    var data: UnsafePointer[T]
    var size: Int

    fn __init__(inout self, size: Int, value: T):
        self.size = size
        self.data = UnsafePointer[T].alloc(self.size)
        for i in range(self.size):
            (self.data + i).init_pointee_copy(value)

    fn __getitem__(self, i: Int) -> T:
        return self.data[i]

    fn __del__(owned self):
        for i in range(self.size):
            (self.data + i).destroy_pointee()
        self.data.free()

var v = Array[Float32](4, 3.14)
print(v[0], v[1], v[2], v[3])
```

Notice that the `T` parameter is being used as the formal type for the `value` arguments and the return type of the `__getitem__()` function. Parameters allow the `Array` type to provide different APIs based on the different use-cases.

There are many other cases that benefit from more advanced use of parameters. For example, you can execute a closure N times in parallel, feeding in a value from the context, like this:

```
fn parallelize[func: fn (Int) -> None](num_work_items: Int):
    # Not actually parallel: see the 'algorithm' module for real implementation.
    for i in range(num_work_items):
        func(i)
```

Another example where this is important is with variadic generics, where an algorithm or data structure may need to be defined over a list of heterogeneous types such as for a tuple. Right now, this is not fully supported in Mojo and requires writing some MLIR by hand. In the future, this will be possible in pure Mojo.

`alias`: named parameter expressions

It is very common to want to *name* compile-time values. Whereas `var` defines a runtime value, we need a way to define a compile-time temporary value. For this, Mojo uses an `alias` declaration.

For example, the `DType` struct implements a simple enum using aliases for the enumerators like this (the actual `DType` implementation details vary a bit):

```
struct DType:
    var value : UI8
    alias invalid = DType(0)
    alias bool = DType(1)
    alias int8 = DType(2)
    alias uint8 = DType(3)
    alias int16 = DType(4)
    alias int16 = DType(5)
    ...
    alias float32 = DType(15)
```

This allows clients to use `DType.float32` as a parameter expression (which also works as a runtime value) naturally. Note that this is invoking the runtime constructor for `DType` at compile-time.

Types are another common use for aliases. Because types are compile-time expressions, it is handy to be able to do things like this:

```
alias Float16 = SIMD[DType.float16, 1]
alias UInt8 = SIMD[DType.uint8, 1]

var x: Float16 = 0  # Float16 works like a "typedef"
```

Like `var` variables, aliases obey scope, and you can use local aliases within functions as you'd expect.

Fully-bound, partially-bound, and unbound types

A parametric type with its parameters specified is said to be *fully-bound*. That is, all of its parameters are bound to values. As mentioned before, you can only instantiate a fully-bound type (sometimes called a *concrete type*).

However, parametric types can be *unbound* or *partially bound* in some contexts. For example, you can alias a partially-bound type to create a new type that requires fewer parameters:

```
from collections import Dict

alias StringKeyDict = Dict[String, _]
var b = StringKeyDict[UInt8]()
b["answer"] = 42
```

Here, `StringKeyDict` is a type alias for a `Dict` that takes `String` keys. The underscore `_` in the parameter list indicates that the second parameter, `V` (the value type), is unbound. You specify the `V` parameter later, when you use `StringKeyDict`.

For example, given the following type:

```
struct MyType[s: String, i: Int, i2: Int, b: Bool = True]:
    pass
```

It can appear in code in the following forms:

- *Fully bound*, with all of its parameters specified:

  ```
  MyType["Hello", 3, 4, True]
  ```

- *Partially bound*, with *some but not all* of its parameters specified:

  ```
  MyType["Hola", _, _, True]
  ```

- *Unbound*, with no parameters specified:

  ```
  MyType[_, _, _, _]
  ```

You can also use the star-underscore expression `*_` to unbind an arbitrary number of positional parameters at the end of a parameter list.

```
# These two types are equivalent
MyType["Hello", *_]
MyType["Hello", _, _, _]
```

When a parameter is explicitly unbound with the `_` or `*_` expression, you **must** specify a value for that parameter to use the type. Any default value from the original type declaration is ignored.

Partially-bound and unbound parametric types can be used in some contexts where the missing (unbound) parameters will be supplied later—such as in aliases and automatically parameterized functions.

Omitted parameters

Mojo also supports an alternate format for unbound parameter where the parameter is simply omitted from the expression:

```
# Partially bound
MyType["Hi there"]
# Unbound
MyType
```

This format differs from the explicit unbinding syntax described above in that the default values for omitted parameters are bound immediately. For example, the following expressions are equivalent:

```
MyType["Hi there"]
# equivalent to
MyType["Hi there", _, _, True] # Uses the default value for `b`
```

Note

This format is currently supported for backwards compatibility. We intend to deprecate this format in the future in favor of the explicit unbinding syntax.

Automatic parameterization of functions

Mojo supports "automatic" parameterization of functions. If a function argument type is a partially-bound or unbound type, the unbound parameters are automatically added as input parameters on the function. This is easier to understand with an example:

```
fn print_params(vec: SIMD[*_]):
    print(vec.type)
    print(vec.size)

var v = SIMD[DType.float64, 4](1.0, 2.0, 3.0, 4.0)
print_params(v)
```

```
float64
4
```

In the above example, the `print_params` function is automatically parameterized. The `vec` argument takes an argument of type `SIMD[*_]`. This is an unbound parameterized type—that is, it doesn't specify any parameter values for the type. Mojo treats the unbound parameters on `vec` as implicit parameters on the function. This is roughly equivalent to the following code, which includes *explicit* input parameters:

```
fn print_params[t: DType, s: Int](vec: SIMD[t, s]):
    print(vec.type)
    print(vec.size)
```

When you call `print_params()` you must pass it a concrete instance of the `SIMD` type—that is, one with all of its parameters specified, like `SIMD[DType.float64, 4]`. The Mojo compiler *infers* the parameter values from the input argument.

With a manually parameterized function, you can access the input parameters by name (for example, `t` and `s` in the previous example). For an automatically parameterized function, you can access the parameters as attributes on the argument (for example, `vec.type`).

This ability to access a type's input parameters is not specific to automatically parameterized functions, you can use it anywhere. You can access the input parameters of a parameterized type as attributes on the type itself:

```
fn on_type():
    print(SIMD[DType.float32, 2].size) # prints 2
```

Or as attributes on an *instance* of the type:

```
fn on_instance():
    var x = SIMD[DType.int32, 2](4, 8)
    print(x.type) # prints int32
```

You can even use this syntax in the function's signature to define a function's arguments and return type based on an argument's parameters. For example, if you want your function to take two SIMD vectors with the same type and size, you can write code like this:

```
fn interleave(v1: SIMD, v2: __type_of(v1)) -> SIMD[v1.type, v1.size*2]:
    var result = SIMD[v1.type, v1.size*2]()
    for i in range(v1.size):
        result[i*2] = SIMD[v1.type, 1](v1[i])
        result[i*2+1] = SIMD[v1.type, 1](v2[i])
    return result
```

```
var a = SIMD[DType.int16, 4](1, 2, 3, 4)
var b = SIMD[DType.int16, 4](0, 0, 0, 0)
var c = interleave(a, b)
print(c)
```

```
[1, 0, 2, 0, 3, 0, 4, 0]
```

As shown in the example, you can use the magic `__type_of(x)` call if you just want to match the type of an argument. In this case, it's more convenient and compact that writing the equivalent `SIMD[v1.type, v1.size]`.

Automatic parameterization with partially-bound types

Mojo also supports automatic parameterization: with partially-bound parameterized types (that is, types with some but not all of the parameters specified).

For example, suppose we have a `Fudge` struct with three parameters:

```
@value
struct Fudge[sugar: Int, cream: Int, chocolate: Int = 7](Stringable):
    fn __str__(self) -> String:
        return str("Fudge (") + str(sugar) + "," +
            str(cream) + "," + str(chocolate) + ")"
```

We can write a function that takes a `Fudge` argument with just one bound parameter (it's *partially bound*):

```
fn eat(f: Fudge[5, *_]):
    print("Ate " + str(f))
```

The `eat()` function takes a `Fudge` struct with the first parameter (`sugar`) bound to the value 5. The second and third parameters, `cream` and `chocolate` are unbound.

The unbound `cream` and `chocolate` parameters become implicit input parameters on the `eat` function. In practice, this is roughly equivalent to writing:

```
fn eat[cr: Int, ch: Int](f: Fudge[5, cr, ch]):
    print("Ate", str(f))
```

In both cases, we can call the function by passing in an instance with the `cream` and `chocolate` parameters bound:

```
eat(Fudge[5, 5, 7]())
eat(Fudge[5, 8, 9]())
```

```
Ate Fudge (5,5,7)
Ate Fudge (5,8,9)
```

If you try to pass in an argument with a `sugar` value other than 5, compilation fails, because it doesn't match the argument type:

```
eat(Fudge[12, 5, 7]())
# ERROR: invalid call to 'eat': argument #0 cannot be converted from 'Fudge[12, 5, 7]' to
'Fudge[5, 5, 7]'
```

You can also explicitly unbind individual parameters. This gives you more freedom in specifying unbound parameters.

For example, you might want to let the user specify values for `sugar` and `chocolate`, and leave `cream` constant. To do this, replace each unbound parameter value with a single underscore (`_`):

```
fn devour(f: Fudge[_, 6, _]):
    print("Devoured",  str(f))
```

Again, the unbound parameters (`sugar` and `chocolate`) are added as implicit input parameters on the function. This version is roughly equivalent to the following code, where these two values are explicitly bound to the input parameters, `su` and `ch`:

```
fn devour[su: Int, ch: Int](f: Fudge[su, 6, ch]):
    print("Devoured", str(f))
```

You can also specify parameters by keyword, or mix positional and keyword parameters, so the following function is roughly equivalent to the previous one: the first parameter, `sugar` is explicitly unbound with the underscore character. The `chocolate` parameter is unbound using the keyword syntax, `chocolate=_`. And `cream` is explicitly bound to the value 6:

```
fn devour(f: Fudge[_, chocolate=_, cream=6]):
    print("Devoured",  str(f))
```

All three versions of the `devour()` function work with the following calls:

```
devour(Fudge[3, 6, 9]())
devour(Fudge[4, 6, 8]())
```

```
Devoured Fudge (3,6,9)
Devoured Fudge (4,6,8)
```

Legacy syntax (omitted parameters)

You can also specify an unbound or partially-bound type by omitting parameters: for example:

```
fn nibble(f: Fudge[5]):
    print("Ate", str(f))
```

Here, `Fudge[5]` works like `Fudge[5, *_]` **except** in the handling of parameters with default values. Instead of discarding the default value of `chocolate`, `Fudge[5]` binds the default value immediately, making it equivalent to: `Fudge[5, _, 7]`.

This means that the following code won't compile with the previous definition for `nibble()` function, since it doesn't use the default value for `chocolate`:

```
nibble(Fudge[5, 5, 9]())
# ERROR: invalid call to 'eat': argument #0 cannot be converted from 'Fudge[5, 5, 9]' to
'Fudge[5, 5, 7]'
```

TODO

Support for omitting unbound parameters will eventually be deprecated in favor of explicitly unbound parameters using `_` and `*_`.