Modular

Mojo Compile-time Interpreter in MLIR 🔥 

Weiwei Chen
weiwei.chen@modular.com

EuroLLVM 2026

Compile Time 🔥 Meta-programming

Normal functions have *"arguments"*:

- runtime values that can vary on every invocation

Mojo allows [comptime arguments]:

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

Simplifies and unifies the language:

- types are just comptime values
- removes many special case features in other languages
- superpowers: e.g. can fully unroll a loop

```
# Naive power function.
def 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.
def 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.
def pow_exp_fast[exp: Int](base: Int) -> Int:
 res = 1
 comptime for i in range(exp):
   res *= base
 return res
```

Building Modern Language Frontends with MLIR by Billy Zhu, Chris Lattner, *LLVM Dev Meeting 2025*

```
# Returns a heap-allocated List.
def 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^

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

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

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

Code vs Meta-code

Explicit control over expression evaluation:

- **"comptime"** guarantees comp-time eval
- **"var"** is dynamic at its execution time
- Expressions are written the same way in both roles - easy to debug

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

Enables powerful user-defined libraries:

- kernel parameterization over vector length, unroll factor, tile factor, …
- very important for GPU programming

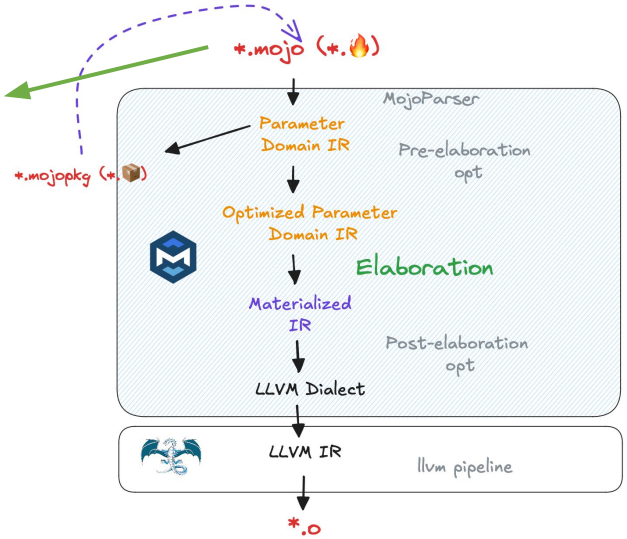
Building Modern Language Frontends with MLIR by Billy Zhu, Chris Lattner, *LLVM Dev Meeting 2025*

Elaboration: Meta to Concrete

```mojo
def add[p: Int](arg: Int) -> Int:
    return p + arg

def f(arg: Int) -> Int:     # arg = 3
    comptime c = add[1](2)  # c = 1 + 2 = 3
    var v1 = add[1](arg)    # v1 = 1 + 3 = 4
    var v2 = add[c](arg)    # v2 = c + 3 = 6
    return v1 + v2          # v1 + v2 = 10

def main():
    comptime v = f(3)       # f(3) = 10
    var a = add[1](1)       # a = 2
    print(v + a)            # 10 + 2 = 12
```



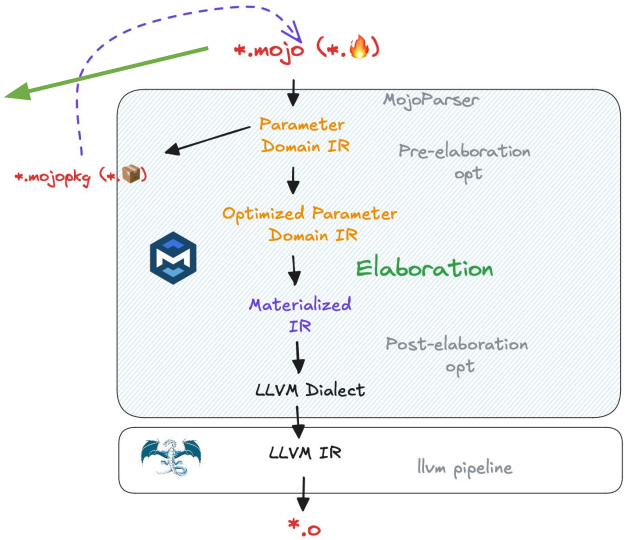
What we learned from building Mojo's optimization pipeline by Weiwei Chen, *LLVM Dev Meeting 2024*

Elaboration: Meta to Concrete

```mojo
def add[p: Int](arg: Int) -> Int:
    return p + arg

def f(arg: Int) -> Int:     # arg = 3
    comptime c = add[1](2)  # c = 1 + 2 = 3
    var v1 = add[1](arg)    # v1 = 1 + 3 = 4
    var v2 = add[c](arg)    # v2 = c + 3 = 6
    return v1 + v2          # v1 + v2 = 10

def main():
    comptime v = f(3)       # f(3) = 10
    var a = add[1](1)       # a = 2
    print(v + a)            # 10 + 2 = 12
```



Elaborator substitutes comp-time *parameters* (symbolic) with constants (concrete) evaluated by the **Interpreter.**

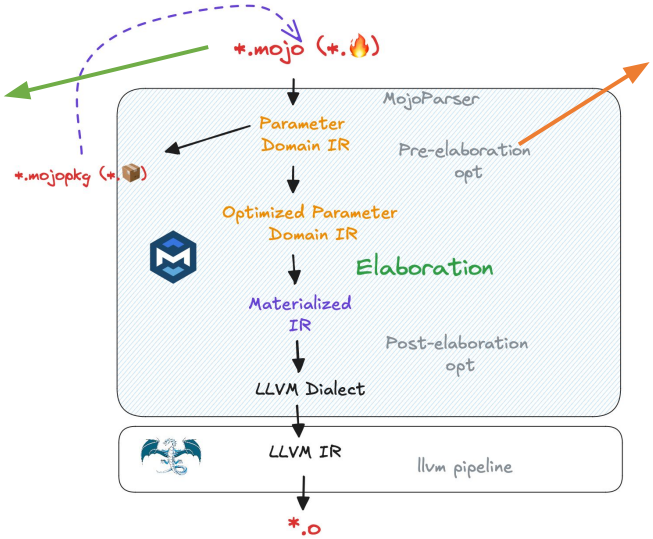
What we learned from building Mojo's optimization pipeline by Weiwei Chen, *LLVM Dev Meeting 2024*

Elaboration: Meta to Concrete

```
def add[p: Int](arg: Int) -> Int:
    return p + arg

def f(arg: Int) -> Int:     # arg = 3
    comptime c = add[1](2)  # c = 1 + 2 = 3
    var v1 = add[1](arg)    # v1 = 1 + 3 = 4
    var v2 = add[c](arg)    # v2 = c + 3 = 6
    return v1 + v2          # v1 + v2 = 10

def main():
    comptime v = f(3)       # f(3) = 10
    var a = add[1](1)       # a = 2
    print(v + a)            # 10 + 2 = 12
```




```
kgen.generator @print(%arg: index){
   kgen.return
}
kgen.generator @add<p>(%arg: index) -> index {
 %0 = kgen.param.constant = <p>
 %1 = index.add %0, %arg
 kgen.return %1: index
}
kgen.generator @f(%arg: index) -> index  {
 kgen.param.apply c = [(index) -> index: @add<1>](2)
 %1 = kgen.call @add<1>(%arg) : (index) -> index
 %2 = kgen.call @add<c>(%arg) : (index) -> index
 %3 = index.add %1, %2
 kgen.return %3: index
}
kgen.generator export @main() {
 kgen.param.apply v = [(index) -> index: @f](3)
 %0 = kgen.param.constant = <v>
 %index1 = kgen.param.constant = <1>
 %1 = kgen.call @add<1>(%index1) : (index) -> index
 %2 = index.add %0, %1
 kgen.call @print(%2): (index) -> ()
 kgen.return
}
```

Elaborator substitutes comp-time *parameters* (symbolic) with constants (concrete) evaluated by the **Interpreter.**

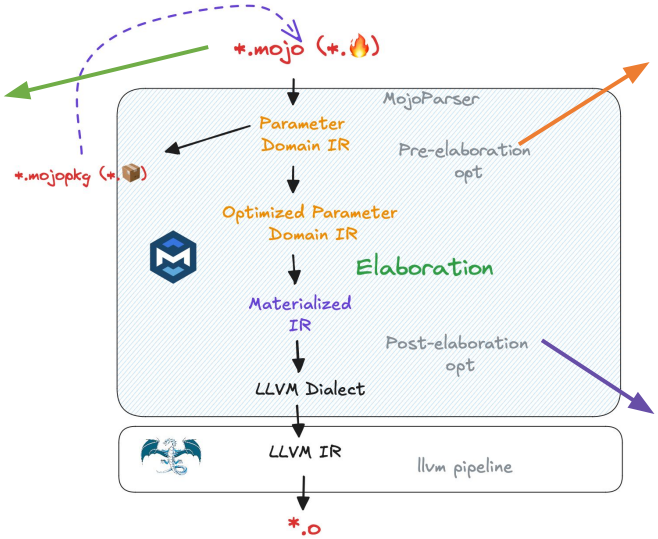
What we learned from building Mojo's optimization pipeline by Weiwei Chen, *LLVM Dev Meeting 2024*

Elaboration: Meta to Concrete

```
def add[p: Int](arg: Int) -> Int:
    return p + arg

def f(arg: Int) -> Int:     # arg = 3
    comptime c = add[1](2)  # c = 1 + 2 = 3
    var v1 = add[1](arg)    # v1 = 1 + 3 = 4
    var v2 = add[c](arg)    # v2 = c + 3 = 6
    return v1 + v2          # v1 + v2 = 10

def main():
    comptime v = f(3)       # f(3) = 10
    var a = add[1](1)       # a = 2
    print(v + a)            # 10 + 2 = 12
```




```
kgen.generator @print(%arg: index){
   kgen.return
}
kgen.generator @add<p>(%arg: index) -> index {
 %0 = kgen.param.constant = <p>
 %1 = index.add %0, %arg
 kgen.return %1: index
}
kgen.generator @f(%arg: index) -> index  {
 kgen.param.apply c = [(index) -> index: @add<1>](2)
 %1 = kgen.call @add<1>(%arg) : (index) -> index
 %2 = kgen.call @add<c>(%arg) : (index) -> index
 %3 = index.add %1, %2
 kgen.return %3: index
}
kgen.generator export @main() {
 kgen.param.apply v = [(index) -> index: @f](3)
 %0 = kgen.param.constant = <v>
 %index1 = kgen.param.constant = <1>
 %1 = kgen.call @add<1>(%index1) : (index) -> index
 %2 = index.add %0, %1
 kgen.call @print(%2): (index) -> ()
 kgen.return
}
```

```
kgen.func @print(%arg0: index) {
 kgen.return
}
kgen.func @"add,p=1"(%arg0: index) -> index {
 %index1 = kgen.param.constant = <1>
 %0 = index.add %index1, %arg0
 kgen.return %0 : index
}
kgen.func @"add,p=3"(%arg0: index) -> index {
 %index3 = kgen.param.constant = <3> // c = 3
 %0 = index.add %index3, %arg0
 kgen.return %0 : index
}
kgen.func @f(%arg0: index) -> index {
 %0 = kgen.call @"add,p=1"(%arg0) : (index) -> index
 %1 = kgen.call @"add,p=3"(%arg0) : (index) -> index // c = 3
 %2 = index.add %0, %1
 kgen.return %2 : index
}
kgen.func export @main() {
 %index10 = kgen.param.constant = <10> // v = 10
 %index1 = kgen.param.constant = <1>
 %0 = kgen.call @"add,p=1"(%index1) : (index) -> index
 %1 = index.add %index10, %0
 kgen.call @print(%1) : (index) -> ()
 kgen.return
}
```

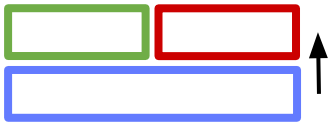
Elaborator substitutes comp-time *parameters* (symbolic) with constants (concrete) evaluated by the **Interpreter.**

What we learned from building Mojo's optimization pipeline by Weiwei Chen, *LLVM Dev Meeting 2024*

Interpreting a function in MLIR

```
kgen.func @"add,p=1"(%arg0: index) -> index {
 %index1 = kgen.param.constant = <1>
 %0 = index.add %index1, %arg0
 kgen.return %0 : index
}
kgen.func @"add,p=3"(%arg0: index) -> index {
 %index3 = kgen.param.constant = <3>
 %0 = index.add %index3, %arg0
 kgen.return %0 : index
}
kgen.func @f(%arg0: index) -> index {
 %0 = kgen.call @"add,p=1"(%arg0) : (index) -> index
 %1 = kgen.call @"add,p=3"(%arg0) : (index) -> index
 %2 = index.add %0, %1
 kgen.return %2 : index
}
kgen.generator export @main() {
 # comptime v = f(3)
 kgen.param.apply v = [(index) -> index: @f](3)
 %0 = kgen.param.constant = <v>
 %index1 = kgen.param.constant = <1>
 %1 = kgen.call @add<1>(%index1) : (index) -> index
 %2 = index.add %0, %1
 kgen.call @print(%2): (index) -> ()
 kgen.return
}
```

- The interpreter executes the function IR (concrete) as a program
- Program counter (pc)
- Evaluate each op with OpType::interpret() or OpType::fold()
- Map: SSA value => interpreted constant value (mlir::TypedAttr)
- Stack frame for function calls:



frame: `kgen.func @"add,p=1"(%arg0: index) -> index`
pc: `kgen.return %0 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	4

frame: `kgen.func @"add,p=3"(%arg0: index) -> index`
pc: `kgen.return %0 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	6

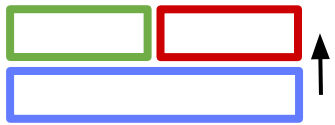
frame: `@f(%arg0: index) -> index`
pc: `kgen.return %2 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	4
%1	6
%2	10

Interpreting a function in MLIR

```
kgen.func @"add,p=1"(%arg0: index) -> index {
 %index1 = kgen.param.constant = <1>
 %0 = index.add %index1, %arg0
 kgen.return %0 : index
}
kgen.func @"add,p=3"(%arg0: index) -> index {
 %index3 = kgen.param.constant = <3>
 %0 = index.add %index3, %arg0
 kgen.return %0 : index
}
kgen.func @f(%arg0: index) -> index {
 %0 = kgen.call @"add,p=1"(%arg0) : (index) -> index
 %1 = kgen.call @"add,p=3"(%arg0) : (index) -> index
 %2 = index.add %0, %1
 kgen.return %2 : index
}
kgen.generator export @main() {
 # comptime v = f(3)
 kgen.param.apply v = [(index) -> index: @f](3)
 %0 = kgen.param.constant = <v>
 %index1 = kgen.param.constant = <1>
 %1 = kgen.call @add<1>(%index1) : (index) -> index
 %2 = index.add %0, %1
 kgen.call @print(%2): (index) -> ()
 kgen.return
}
```

- The interpreter executes the function IR (concrete) as a program
- Program counter (pc)
- Evaluate each op with OpType::interpret() or OpType::fold()
- Map: SSA value => interpreted constant value (mlir::TypedAttr)
- Stack frame for function calls:



MLIR Operations

MLIR Attributes

MLIR interfaces

frame: `kgen.func @"add,p=1"(%arg0: index) -> index`
pc: `kgen.return %0 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	4

frame: `kgen.func @"add,p=3"(%arg0: index) -> index`
pc: `kgen.return %0 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	6

frame: `@f(%arg0: index) -> index`
pc: `kgen.return %2 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	3
%0	4
%1	6
%2	10

Interpreting with memory types

```mojo
def get_ptr() -> UnsafePointer[Int, MutAnyOrigin]:
  var result = alloc[Int](2)
  result[0] = 0x27
  result[1] = 0x42
  return result

def get_value(ptr: UnsafePointer[Int, MutAnyOrigin], i: Int) -> Int:
  return ptr[i]

def main():
  comptime ptr = get_ptr()
  comptime v = get_value(ptr, 1)
  print(ptr[0] + v)
```

allocate memory during compile-time

materializing compile-time memory to runtime

use compile-time memory to evaluate another comptime

Interpreting with memory types

```
def get_ptr() -> UnsafePointer[Int, MutAnyOrigin]:
  var result = alloc[Int](2)
  result[0] = 0x27
  result[1] = 0x42
  return result

def get_value(ptr: UnsafePointer[Int, MutAnyOrigin], i: Int) -> Int:
  return ptr[i]

def main():
  comptime ptr = get_ptr()
  comptime v = get_value(ptr, 1)
  print(ptr[0] + v)
```

allocate memory during compile-time

materializing compile-time memory to runtime

use compile-time memory to evaluate another comptime

Interpreter Memory Model

memory blob per allocation

```
struct MemoryBlob {
  using MemoryT = SmartVariant<void *, MemoryHandleAttr, std::nullptr_t>;
  /// The base address of the blob.
  int64_t baseAddr;
  /// The size of the blob.
  size_t size;
  /// The alignment of the blob.
  size_t align;
  /// The address space the blob lives in.
  unsigned addressSpace;
  /// The actual memory managed by the interpreter.
  MemoryT memory;
  // ...
};
```

physical memory

Interpreting with memory types

```
def get_ptr() -> UnsafePointer[Int, MutAnyOrigin]:
  var result = alloc[Int](2)
  result[0] = 0x27
  result[1] = 0x42
  return result

def get_value(ptr: UnsafePointer[Int, MutAnyOrigin], i: Int) -> Int:
  return ptr[i]

def main():
  comptime ptr = get_ptr()
  comptime v = get_value(ptr, 1)
  print(ptr[0] + v)
```

allocate memory during compile-time

materializing compile-time memory to runtime

use compile-time memory to evaluate another comptime

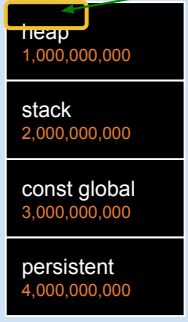
Interpreter Memory Model

memory blob per allocation

```
struct MemoryBlob {
  using MemoryT = SmartVariant<void *, MemoryHandleAttr, std::nullptr_t>;
  /// The base address of the blob.
  int64_t baseAddr;
  /// The size of the blob.
  size_t size;
  /// The alignment of the blob.
  size_t align;
  /// The address space the blob lives in.
  unsigned addressSpace;
  /// The actual memory managed by the interpreter.
  MemoryT memory;
  // ...
};
```

physical memory

@get_ptr() memory space



base addr: 1,000,000,000
size: 16
align: 8

data (hex):
27 0 0 0 42 0 0 0

frame: `@get_ptr() -> !kgen.pointer<index>`
pc: `kgen.return %2 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%size	16
%align	8
%mem	interp.poiner<1,000,000,000>
%i1	1
%p1	interp.poiner<1,000,000,008>

@get_value() memory space

base addr: 1,000,000,000
size: 16
align: 8

data (hex):
27 0 0 0 42 0 0 0

frame: `@get_value(%arg0: !kgen.pointer<index>, %i: index) -> index`
pc: `kgen.return %0: index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	#interp.pointer<1,000,000,000>
%i	1
%ptr	#interp.pointer<1,000,000,008>
%0	0x42

Interpreting with memory types

```
def get_ptr() -> UnsafePointer[Int, MutAnyOrigin]:
  var result = alloc[Int](2)
  result[0] = 0x27
  result[1] = 0x42
  return result

def get_value(ptr: UnsafePointer[Int, MutAnyOrigin], i: Int) -> Int:
  return ptr[i]

def main():
  comptime ptr = get_ptr()
  comptime v = get_value(ptr, 1)
  print(ptr[0] + v)
```

allocate memory during compile-time

materializing compile-time memory to runtime

use compile-time memory to evaluate another comptime

Interpreter Memory Model

memory space per comptime func eval

memory blob per allocation


```
struct MemoryBlob {
  using MemoryT = SmartVariant<void *, MemoryHandleAttr, std::nullptr_t>;
  /// The base address of the blob.
  int64_t baseAddr;
  /// The size of the blob.
  size_t size;
  /// The alignment of the blob.
  size_t align;
  /// The address space the blob lives in.
  unsigned addressSpace;
  /// The actual memory managed by the interpreter.
  MemoryT memory;
  // ...
};
```

physical memory

@get_ptr() memory space

base addr: 1,000,000,000
size: 16
align: 8

data (hex):
27 0 0 0 42 0 0 0

externalize

```
#memory_handle = #interp.memory_handle<8, "0x2700000000000000420000000000000">
#interp.memref<{[(#memory_handle, heap, [], [])], [], 0, 0>
```

MLIR attribute

frame: `@get_ptr() -> !kgen.pointer<index>`
pc: `kgen.return %2 : index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%size	16
%align	8
%mem	interp.poiner<1,000,000,000>
%i1	1
%p1	interp.poiner<1,000,000,008>

@get_value() memory space

internalize

base addr: 1,000,000,000
size: 16
align: 8

data (hex):
27 0 0 0 42 0 0 0

frame: `@get_value(%arg0: !kgen.pointer<index>, %i: index) -> index`
pc: `kgen.return %0: index`
value map:

IR (mlir::Value)	Const (mlir:TypedAttr)
%arg0	#interp.pointer<1,000,000,000>
%i	1
%ptr	#interp.pointer<1,000,000,008>
%0	0x42

Interpreting with memory types

Interpreter memory to IR representation

```
#memory handle = #interp.memory handle<8, "0x270000000000000042000000000000000">
kgen.func @get_ptr() -> !kgen.pointer<index> {
 %idx16 = index.constant 16
 %idx8 = index.constant 8
 %0 = pop.aligned alloc %idx8, %idx16 : <index>
 %index1 = kgen.param.constant = <1>
 %index39 = kgen.param.constant = <39>
 %index66 = kgen.param.constant = <66>
 pop.store %index39, %0 : !kgen.pointer<index>
 %1 = pop.offset %0[%index1] : !kgen.pointer<index>
 pop.store %index66, %1 : !kgen.pointer<index>
 kgen.return %0 : !kgen.pointer<index>
}
kgen.func @get_value(%arg0: !kgen.pointer<index>, %arg1: index) -> index {
 %0 = pop.offset %arg0[%arg1] : !kgen.pointer<index>
 %1 = pop.load %0 : !kgen.pointer<index>
 kgen.return %1 : index
}
kgen.func export @main() {
 %pointer = kgen.param.constant: pointer<index> = <#interp.memref<{[(#memory_handle, heap, [], [])], []}, 0, 0>>
 %index66 = kgen.param.constant = <66>
 %0 = pop.load %pointer : !kgen.pointer<index>
 %1 = index.add %index66, %0
 kgen.call @print(%1) : (index) -> ()
 kgen.return
}
```

MLIR Attribute

```
def get_ptr() -
  var result =
  result[0] = (
  result[1] = (
  return result

def get_value(p
  return ptr[i]

def main():
  comptime ptr
  comptime v =
  print(ptr[0]
```

materializin
memory

@get_p

heap
1,000,000,(

stack
2,000,000,(

const glo
3,000,000,(

persisten
4,000,000,(

%p1	interp.poiner<1,000,000,008>

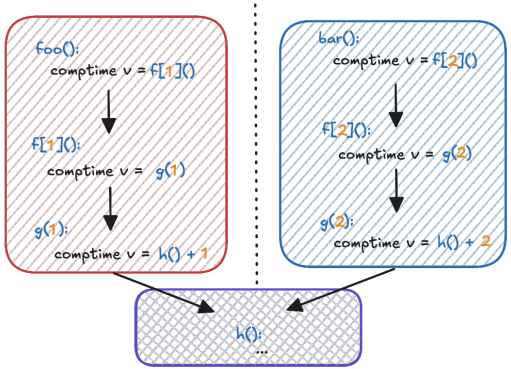
bal

t

Interpreter Performance

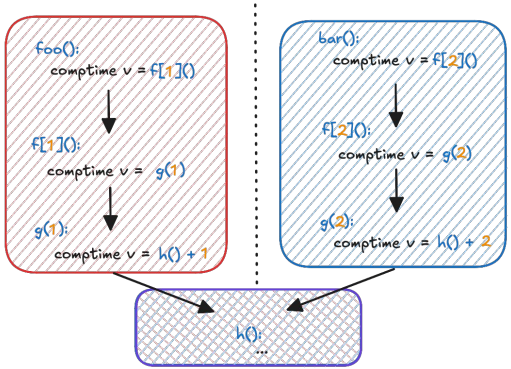
Interpreter Performance

Parallelizing the interpreter



Interpreter Performance

Parallelizing the interpreter



Caching and reusing interpreted results

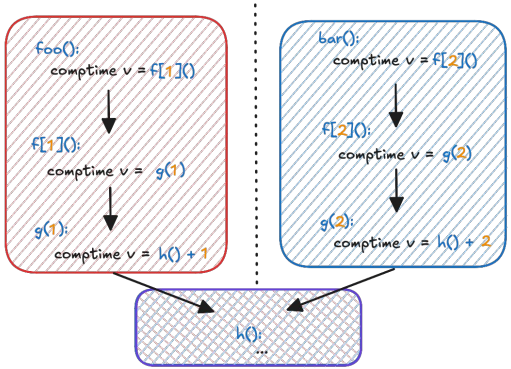
```
def foo():
    comptime v = f[1]("it's fine") # cache

def bar():
    comptime v = f[1]("it's fine") # reuse
```

- cache and reuse results of function invocations with **same** parameters and arguments
- two-level caching (parallelization): thread-local, global
- 100x speedup - with vs without caching

Interpreter Performance

Parallelizing the interpreter



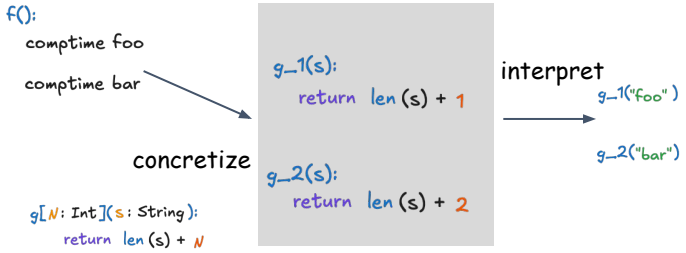
Caching and reusing interpreted results

```
def foo():
    comptime v = f[1]("it's fine") # cache

def bar():
    comptime v = f[1]("it's fine") # reuse
```

- cache and reuse results of function invocations with **same** parameters and arguments
- two-level caching (parallelization): thread-local, global
- 100x speedup - with vs without caching

Interpreting Concrete IR (now)



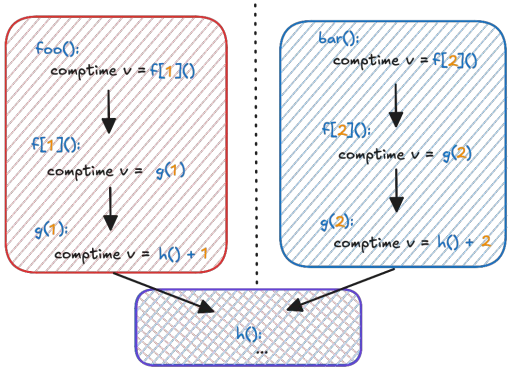

```
def g[N: Int](s: String) -> Int:
    return len(s) + N

def f():
    comptime foo = g[1]("foo")
    comptime bar = g[2]("bar")
    ...
```

- simple interpreting
- concretizing IR overhead
  (70% IR only generated for interpreting test_matmul)

Interpreter Performance

Parallelizing the interpreter



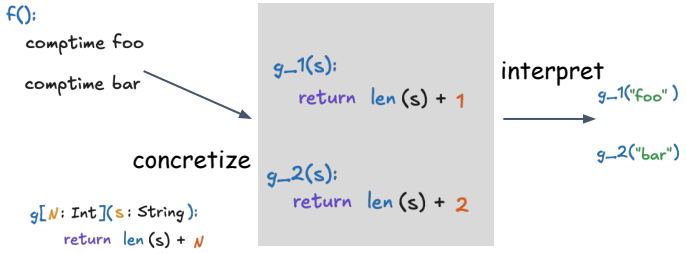
Caching and reusing interpreted results

```
def foo():
    comptime v = f[1]("it's fine") # cache

def bar():
    comptime v = f[1]("it's fine") # reuse
```

- cache and reuse results of function invocations with **same** parameters and arguments
- two-level caching (parallelization): thread-local, global
- 100x speedup - with vs without caching

Interpreting Concrete IR (now)

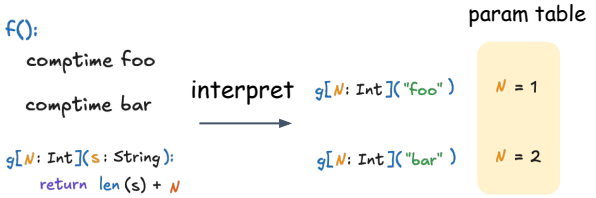



```
def g[N: Int](s: String) -> Int:
    return len(s) + N

def f():
    comptime foo = g[1]("foo")
    comptime bar = g[2]("bar")
    ...
```

- simple interpreting
- concretizing IR overhead
  (70% IR only generated for interpreting test_matmul)

Interpreting Parametric IR (new)

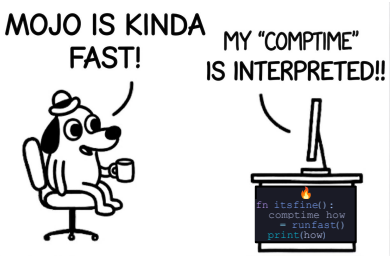


- no need to concretize IR and simple parallelization dependencies
- complex interpreting

Conclusions

Powerful Mojo
Meta-programming 🔥

MLIR based Interpreter



Fast Compilation ⚡

Thank You!

Acknowledgement:
The Mojo Language Team at Modular