`Nelli` - a lightweight, Pythonic, frontend for MLIR

Max Levental, Alok Kamatar, Ryan Chard, Kyle Chard, Ian Foster

(and, recently, Nicolas Vasilache)

THE UNIVERSITY OF CHICAGO | DEPARTMENT OF COMPUTER SCIENCE

globus labs

TL;DR

makslevental/nelli

TL;DR

```python
@mlir_func
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        two = constant(2.0)
        mem = MemRef.alloca([3, 3], F64)
    else:
        six = constant(6.0)
        mem = MemRef.alloca([7, 7], F64)
```

```mlir
func.func @ifs(%arg0: f64, %arg1: f64) {
  %cst = arith.constant 1.000000e+00 : f64
  %0 = arith.cmpf olt, %arg0, %arg1 : f64
  scf.if %0 {
    %cst_0 = arith.constant 2.000000e+00 : f64
    %1 = memref.alloca() : memref<3x3xf64>
  } else {
    %cst_0 = arith.constant 6.000000e+00 : f64
    %1 = memref.alloca() : memref<7x7xf64>
  }
  return
}
```

TL;DR

```python
M, N, K = 4, 16, 8

@mlir_func
def matmul(A: MemRef[(M, N), F32],
           B: MemRef[(N, K), F32],
           C: MemRef[(M, K), F32]):
    for i in range(M):
        for j in range(N):
            for k in range(K):
                a = A[i, j]
                b = B[i, j]
                c = C[i, k]
                d = a * b
                e = c + d
                C[i, k] = e
```

```mlir
func.func @matmul(%arg0: memref<4x16xf32>,
                  %arg1: memref<16x8xf32>,
                  %arg2: memref<4x8xf32>) {
  affine.for %arg3 = 0 to 4 {
    affine.for %arg4 = 0 to 16 {
      affine.for %arg5 = 0 to 8 {
        %0 = memref.load %arg0[%arg3, %arg4]
        %1 = memref.load %arg1[%arg3, %arg4]
        %2 = memref.load %arg2[%arg3, %arg5]
        %3 = arith.mulf %0, %1 : f32
        %4 = arith.addf %2, %3 : f32
        memref.store %4, %arg2[%arg3, %arg5]
      }
    }
  }
  return
}
```

TL;DR

```python
M, N, K = 4, 16, 8
class MyClass1(GPUModule):
    def kernel(self,
               A: MemRef[(M, N), F32],
               B: MemRef[(N, K), F32],
               C: MemRef[(M, K), F32]):
        x = block_id_x()
        y = block_id_y()
        a = A[x, y]
        b = B[x, y]
        C[x, y] = a * b
        return

m = MyClass1()

@mlir_func
def main(A: MemRef[(M, N), F32],
         B: MemRef[(N, K), F32],
         C: MemRef[(M, K), F32]):
    m.kernel(
        A, B, C,
        grid_size=[4, 4, 1],
        block_size=[1, 1, 1]
    )
```

```mlir
module {
  gpu.module @MyClass1 {
    gpu.func @kernel(%arg0: memref<4x16xf32>,
                     %arg1: memref<16x8xf32>,
                     %arg2: memref<4x8xf32>) kernel {
      %0 = gpu.block_id  x
      %1 = gpu.block_id  y
      %2 = memref.load %arg0[%0, %1] : memref<4x16xf32>
      %3 = memref.load %arg1[%0, %1] : memref<16x8xf32>
      %4 = arith.mulf %2, %3 : f32
      memref.store %4, %arg2[%0, %1] : memref<4x8xf32>
      gpu.return
    }
  }
  func.func @main(%arg0: memref<4x16xf32>,
                  %arg1: memref<16x8xf32>,
                  %arg2: memref<4x8xf32>) {
    %c4 = arith.constant 4 : index
    %c1 = arith.constant 1 : index
    %0 = gpu.launch_func async @MyClass1::@kernel
         blocks in (%c4, %c4, %c1)
         threads in (%c1, %c1, %c1)
         args(%arg0 : memref<4x16xf32>,
              %arg1 : memref<16x8xf32>,
              %arg2 : memref<4x8xf32>
         )
    return
  }
}
```

TL;DR

```python
@mlir_func(range_ctor=scf_range)
def loop_unroll_op():
    for i in range(0, 42, 5):
        v = i + i

@sequence
def basic(target, *extra_args):
    m = match(target, ["arith.addi"])
    loop = get_parent_for_loop(m)
    unroll(loop, 4)
```

```
module {
  func.func @loop_unroll_op() {
    %c0 = arith.constant 0 : index
    %c42 = arith.constant 42 : index
    %c5 = arith.constant 5 : index
    scf.for %arg0 = %c0 to %c42 step %c5 {
      %0 = arith.addi %arg0, %arg0 : index
    }
    return
  }
  transform.sequence
  failures(propagate)
  attributes {transform.target_tag = "basic"} {
  ^bb0(%arg0: !pdl.operation):
    %0 = transform.structured.match
         ops{["arith.addi"]} in %arg0
         : (!pdl.operation) -> !pdl.operation
    %1 = transform.loop.get_parent_for %0
         : (!pdl.operation) -> !transform.op<"scf.for">
    transform.loop.unroll %1 {factor = 4 : i64}
    : !transform.op<"scf.for">
  }
}
```

TL;DR

```
run_pipeline(
    module,
    Pipeline()
    .transform_dialect_interpreter()
    .transform_dialect_erase_schedule()
    .materialize(),
)
```

```
func.func @loop_unroll_op() {
  %c0 = arith.constant 0 : index
  %c42 = arith.constant 42 : index
  %c5 = arith.constant 5 : index
  %c40 = arith.constant 40 : index
  %c20 = arith.constant 20 : index
  scf.for %arg0 = %c0 to %c40 step %c20 {
    %1 = arith.addi %arg0, %arg0 : index
    %c1 = arith.constant 1 : index
    %2 = arith.muli %c5, %c1 : index
    %3 = arith.addi %arg0, %2 : index
    %4 = arith.addi %3, %3 : index
    ...
```

TL;DR

```python
M, N, K = 4, 16, 8

@mlir_func
def matmul(
    A: MemRef[(M, N), F64],
    B: MemRef[(N, K), F64],
    C: MemRef[(M, K), F64],
):
    for i in range(0, M):
        for j in range(0, N):
            for k in range(0, K):
                C[i, k] += A[i, j] * B[j, k]

backend = LLVMJITBackend(
    shared_libs=[
        str(c_runner_utils_lib_path),
        str(runner_utils_lib_path)
    ]
)
```

```python
module = backend.compile(
    module,
    kernel_name="matmul",
    pipeline=Pipeline().bufferize().lower_to_llvm(),
)

A = randn(M, N)
B = randn(N, K)
C = zeros((M, K))
self.backend.load(module).matmul(A, B, C)
assert np.allclose(A @ B, C)
```

Outline

1. Prologue
2. Goals
3. Non-goals
4. Four weird tricks (real) language designers hate
    a. `class PYBIND11_EXPORT PyObjectRef`
        i. `def __add__(self): return ArithValue(add(self).result)`
    b. `ast.Expr(ast_call(self.endfor.__name__))`
    c. `POP_JUMP_IF_FALSE`, `POP_JUMP_FORWARD_IF_FALSE`
5. Upstreaming (~IREE)
6. Discussion

“Standing on the shoulders of giants”


stellaraccident


ftynse


joker-eph


rkayaith


River707


nicolasvasilache


jpienaar


teqdruid

Prologue

`loop-fusion?`

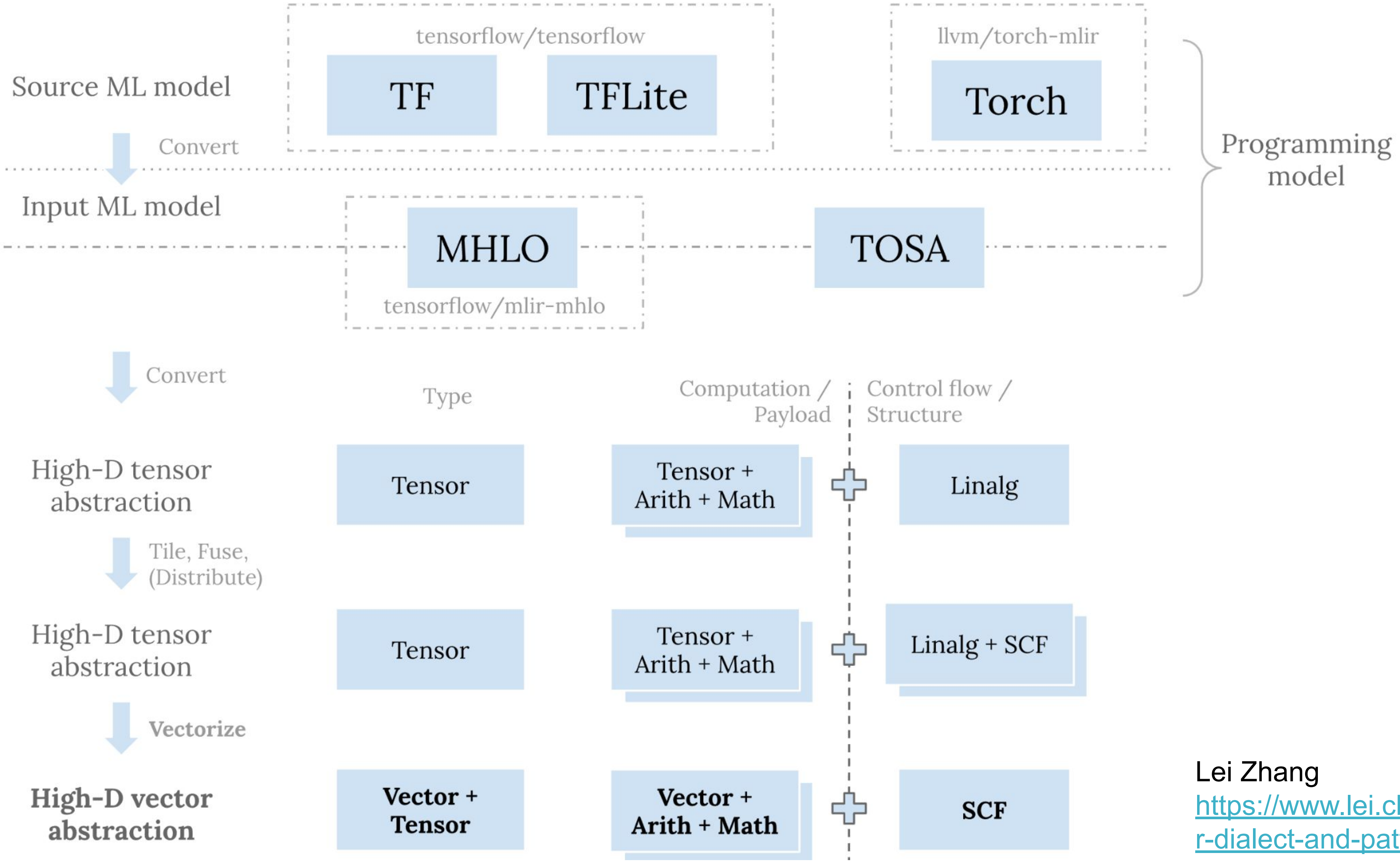
`loop-normalize?`

`loop-tile?`

`loop-unroll-jam?`

Me ~12 months ago

All roads too "royal"



Lei Zhang
https://www.lei.chat/posts/mlir-vector-dialect-and-patterns/

E.g., Torch-MLIR

```python
torch.nn.Conv2d(3, 1, 3)
```

→

```
%alloc = memref.alloc() {alignment = 64 : i64}
scf.for %arg1 = %c0 to %c1 step %c1 {
  scf.for %arg2 = %c0 to %c1 step %c1 {
    scf.for %arg3 = %c0 to %c222 step %c1 {
      scf.for %arg4 = %c0 to %c222 step %c1 {
        memref.store %cst, %alloc[%arg1, %arg2, %arg3, %arg4]

%alloc_0 = memref.alloc() {alignment = 64 : i64}
scf.for %arg1 = %c0 to %c1 step %c1 {
 scf.for %arg2 = %c0 to %c1 step %c1 {
   scf.for %arg3 = %c0 to %c222 step %c1 {
     scf.for %arg4 = %c0 to %c222 step %c1 {
       %1 = memref.load %alloc[%arg1, %arg2, %arg3, %arg4]
       memref.store %1, %alloc_0[%arg1, %arg2, %arg3, %arg4]

memref.dealloc %alloc : memref<1x1x222x222xf32>
scf.for %arg1 = %c0 to %c1 step %c1 {
 scf.for %arg2 = %c0 to %c1 step %c1 {
   scf.for %arg3 = %c0 to %c222 step %c1 {
     scf.for %arg4 = %c0 to %c222 step %c1 {
       scf.for %arg5 = %c0 to %c3 step %c1 {
         scf.for %arg6 = %c0 to %c3 step %c1 {
           scf.for %arg7 = %c0 to %c3 step %c1 {
             %1 = arith.addi %arg3, %arg6 : index
             %2 = arith.addi %arg4, %arg7 : index
             %3 = memref.load %cast[%arg1, %arg5, %1, %2]
             %4 = memref.load %0[%arg2, %arg5, %arg6, %arg7]
             %5 = memref.load %alloc_0[%arg1, %arg2, %arg3, %arg4]
             %6 = arith.mulf %3, %4 : f32
             %7 = arith.addf %5, %6 : f32
             memref.store %7, %alloc_0[%arg1, %arg2, %arg3, %arg4]
```

Disparate flows

```python
implicit_return = results is None
symbol_name = name or f.__name__
function_type = FunctionType.get(
  inputs=inputs,
  results=[] if implicit_return else results
)
func_op = FuncOp(name=symbol_name, type=function_type)
with InsertionPoint(func_op.add_entry_block()):
  func_args = func_op.entry_block.arguments
  func_kwargs = {}
```

Python “flow”

```cpp
template <typename OpType>
class QuantizeSingleOperandFloat : public OpRewritePattern<OpType> {
  using OpRewritePattern<OpType>::OpRewritePattern;

  LogicalResult matchAndRewrite(OpType op,
                                PatternRewriter &rewriter) const override {

    auto operand = op.getOperand();
    auto ctx = rewriter.getContext();
    if (operand.getType().isInteger(16)) {
      auto bitcastRes =
          rewriter
              .create<arith::BitcastOp>(operand.getLoc(), Float16Type::get(ctx),
                                        operand)
              .getResult();
      auto sqrtRes =
          rewriter.create<OpType>(op.getLoc(), bitcastRes).getResult();
```

C++ “flow”

Goals

1. Easy to use (not simple, not safe)
2. Simple to understand (implementation)
3. Easy to get (install, download, etc)
4. As close to MLIR *as possible*

Easy := “just works”
Simple := doable given a little effort (no monads, no λ-calculi, no quantum computers

Non-goals

Markus Böck
zero9178 ·

Ivan Butygin
Hardcode84

Pylir Public

An optimizing ahead-of-time Python Compiler

C++ ☆ 63 2

numba-mlir Public

POC work on MLIR backend

C++ ☆ 4 4 0 4

How?

Attempt 1: Build a whole-ass compiler (~~parsing~~, ~~AST~~, type inference, control-flow, etc.)

- **Pros:**
  - Parsing and AST come free (`import ast`)
  - Own your whole destiny (e.g. custom syntax)
  - Can be faster?
- **Cons**:
  - Immense amount of work
  - **Basically can't reuse existing bindings** because you're analyzing the program, not running it

Attempt 2: Build a Python bytecode interpreter *in Python* that runs the Python program (overriding some of the opcodes, such as `MAKE_FUNCTION`)

- **Pros:**
  - Own most of your destiny
  - **Can reuse existing bindings** (since you're actually running the Python program)
- **Cons**:
  - Immense amount of unnecessary work (handling op codes that you don't care about overriding)

Is there a better way?

Idea 1: Use `sys.settrace` and `f_trace_opcodes` to hook *particular opcodes*

- **Pros:**
  - Most of the benefits of the bytecode interpreter approach with none of the extra gristle
- **Cons**:
  - Can't override the opcodes (only instrument)
  - 3.11 removes `f_valuestack` (which you need for things like replacing induction vars)

Idea 2: Override execution at the AST node level *at runtime*.

- **Pros:**
  - Override only the nodes you care about (e.g. `ast.FunctionDef`)
  - Stable across Python versions
  - **Can reuse existing bindings** (because you are running the Python program)
- **Cons**:
  - Really hard to get right (basically gotta implement macro expansion rules from Lisp)
- **Crisis averted**: Pyccolo (god blessed the full-timers for they are the patron saints of the weekend warriors)

Idea 3

Step 1

mlir/lib/Bindings/Python/IRModule.h

```
class PyObjectRef {
class PyThreadContextEntry {
...
class PyType : public BaseContextObject {
class PyValue {
class PyAffineExpr : public BaseContextObject {
class PyAffineMap : public BaseContextObject {
class PyIntegerSet : public BaseContextObject {
class PySymbolTable {
```

cpp_ext/Pybind.h

```
class PyArithValue : public PyConcreteValue<PyArithValue> {
public:
...
}

class PyMemRefValue : public PyConcreteValue<PyMemRefValue> {
public:
...
}

class PyTensorValue : public PyConcreteValue<PyTensorValue> {
public:
...
}
```

mlir/lib/Bindings/Python/*.cpp

```
py::class_<PyAttribute>(m, "Attribute", py::module_local())
py::class_<PyNamedAttribute>(m, "NamedAttribute", py::module_local())
py::class_<PyType>(m, "Type", py::module_local())
py::class_<PyValue>(m, "Value", py::module_local())
```

nelli/mlir/arith.py

```
from ._mlir._mlir_libs._nelli_mlir import ArithValue

class ArithValue(ArithValue):
    def __add__(self, other):
        return ArithValue(arith.AddIOp(self, other).result)
        ...
```

Step 2

fiddle with **`Parameter.annotation`**

```python
def double_loop(M: Index, N: Index):
    two = constant(1.0)
    mem = AffineMemRef.alloca([10, 10], F64)
    for i in affine_range(1, 10, 1):
        for j in affine_range(1, 10, 1):
            v = mem[i, j]
            w = v * two
            mem[i, j] = w
            affine_endfor()
        affine_endfor()
```

```
func.func @double_loop(%arg0: index, %arg1: index) {
  %cst = arith.constant 1.000000e+00 : f64
  %alloca = memref.alloca() : memref<10x10xf64>
  affine.for %arg2 = 1 to 10 {
    affine.for %arg3 = 1 to 10 {
      %1 = affine.load %0[%arg2, %arg3] : memref<10x10xf64>
      %2 = arith.mulf %1, %cst : f64
      affine.store %2, %0[%arg2, %arg3] : memref<10x10xf64>
    }
  }
  return
}
```

overload **`__getitem__`**

“[where] art thou [block terminator]?”

```
@mlir_func(rewrite_ast_=True)
def double_loop(M: Index, N: Index):
    two = constant(1.0)
    mem = AffineMemRef.alloca([10, 10], F64)
    for i in range(1, 10, 1):
        for j in range(1, 10, 1):
            v = mem[i, j]
            w = v * two
            mem[i, j] = w
```

“[where] art thou [block terminator]?”

nelli/mlir/func.py

```
@mlir_func(rewrite_ast_=True)
def double_loop(M: Index, N: Index):
    two = constant(1.0)
    mem = AffineMemRef.alloca([10, 10], F64)
    for i in range(1, 10, 1):
        for j in range(1, 10, 1):
            v = mem[i, j]
            w = v * two
            mem[i, j] = w
```

```
class InsertEndFors(ast.NodeTransformer):
    def __init__(self, endfor):
        self.endfor = endfor

    def visit_For(self, node):
        for i, b in enumerate(node.body):
            node.body[i] = self.visit(b)
        node.body.append(
            ast.Expr(ast_call(self.endfor.__name__))
        )
        return node
```

Step 3

```python
@mlir_func(rewrite_ast_=False)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if scf_if(M < N):
        one = constant(1.0)
        mem = MemRef.alloca([10, 10], F64)
        scf_endif_branch()
```

→

```python
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        one = constant(1.0)
        mem = MemRef.alloca([10, 10], F64)
```

Step 3

```python
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        two = constant(2.0)
        mem = MemRef.alloca([3, 3], F64)
    else:
        six = constant(6.0)
        mem = MemRef.alloca([7, 7], F64)
```

Recall goal “simple to understand” → no/little source analysis

Non-deterministic TM?

```python
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        two = constant(2.0)
        mem = MemRef.alloca([3, 3], F64)
    else:
        six = constant(6.0)
        mem = MemRef.alloca([7, 7], F64)
```

```
func.func @ifs(%arg0: f64, %arg1: f64) {
  %cst = arith.constant 1.000000e+00 : f64
  %0 = arith.cmpf olt, %arg0, %arg1 : f64
  scf.if %0 {
    %cst_0 = arith.constant 2.000000e+00 : f64
    %1 = memref.alloca() : memref<3x3xf64>
  } else {
    %cst_0 = arith.constant 6.000000e+00 : f64
    %1 = memref.alloca() : memref<7x7xf64>
  }
  return
}
```

No, just haxx

```python
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        two = constant(2.0)
        mem = MemRef.alloca([3, 3], F64)
    else:
        six = constant(6.0)
        mem = MemRef.alloca([7, 7], F64)
```

```
149           0 LOAD_GLOBAL              0 (constant)
              2 LOAD_CONST               1 (1.0)
              4 CALL_FUNCTION            1
...
             22 COMPARE_OP               0 (<)
             24 CALL_FUNCTION            1
             26 POP_JUMP_IF_FALSE       31 (to 62)
... // true branch
150          52 LOAD_GLOBAL              6 (scf_endif_branch)
...
150     >>   62 LOAD_GLOBAL              7 (scf_else)
... // else branch
150          92 LOAD_GLOBAL              6 (scf_endif_branch)
             94 CALL_FUNCTION            0
             96 POP_TOP
             98 LOAD_GLOBAL              8 (scf_endif)
...
            106 RETURN_VALUE
```

In summary

- As little metaprogramming as possible (fat)
- A whole lot of syntactic sugar
- Built on a strong (protein rich) foundation (i.e., the upstream bindings)

In summary

Upstreaming

PSA: Status of end-to-end Structured Codegen on Tensors and Declarative Pipelines

MLIR

nicolasvasilache

Hi everyone,

This is a followup post from:

- transform dialect cheatsheet (link 18)
- our paper from last year on structured codegen (link 13)

→

common extensions and structured op extensions #31

Open **makslevental** wants to merge 3 commits into `main` from `catchup_iree`

Conversation 0 | Commits 3 | Checks 3 | Files changed 40

makslevental commented yesterday • edited — Owner

casualities:

CommonExtensionsOps.td

```
// def IREEEraseHALDescriptorTypeFromMemRefOp : Op<Transform_Dialect,
//     "erase_hal_descriptor_type_from_memref",
//     [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
//      TransformEachOpTrait,
//      TransformOpInterface]> {
```

and

Reviewers
No reviews
Still in progress? C

Assignees
No one—assign yo

Labels
None yet

`Nelli` as a "staging ground" for upstreaming IREE

Discussion

1. If this seems useful to you, can you articulate why?
    a. “micro-kernels”?
2. Re “easy to get” goal: MLIR builder bot?
3. Knowledge transfer for bindings in other languages (Java/FFI)

`Nelli` - a lightweight, Pythonic, frontend for MLIR

Max Levental, Alok Kamatar, Ryan Chard, Kyle Chard, Ian Foster

(and, recently, Nicolas Vasilache)

THE UNIVERSITY OF CHICAGO | DEPARTMENT OF COMPUTER SCIENCE

globus labs