PyDSL

A Python subset for a better MLIR programming experience (Part II)

Kevin Lee (k323lee@uwaterloo.ca),

Kai-Ting Wang* (kai.ting.wang@huawei.com)

```
def transform_seq_fuse_tile(targ: AnyOp):
    fuse(match(targ, "fuse_1"), match(targ, "fuse_2"), 3)
    tile(match(targ, "tile"), [8, 8, 32, 32], 8)


def heat_fuse_tile(tsteps: Index, n: Index, A: MemF32, B: MemF32):
    a: F32 = 2.0
    b: F32 = 0.125
    """@tag("tile")"""
    for _ in arange(tsteps):
        """@tag("fuse_1")"""
        for i in arange(1, n-1):
            for j in arange(1, n-1):
                for k in arange(1, n-1):
                    B[i,j,k] = A[i,j,k] + b * (
                        A[i+1,j,k] - a * A[i,j,k] + A[i-1,j,k] + \
                        A[i,j+1,k] - a * A[i,j,k] + A[i,j-1,k] + \
                        A[i,j,k+1] - a * A[i,j,k] + A[i,j,k-1]
                    )
        """@tag("fuse_2")"""
        for i in arange(1, n-1):
            for j in arange(1, n-1):
                for k in arange(1, n-1):
                    A[i,j,k] = B[i,j,k] + b * (
                        B[i+1,j,k] - a * B[i,j,k] + B[i-1,j,k] + \
                        B[i,j+1,k] - a * B[i,j,k] + B[i,j-1,k] + \
                        B[i,j,k+1] - a * B[i,j,k] + B[i,j,k-1]
                    )
```

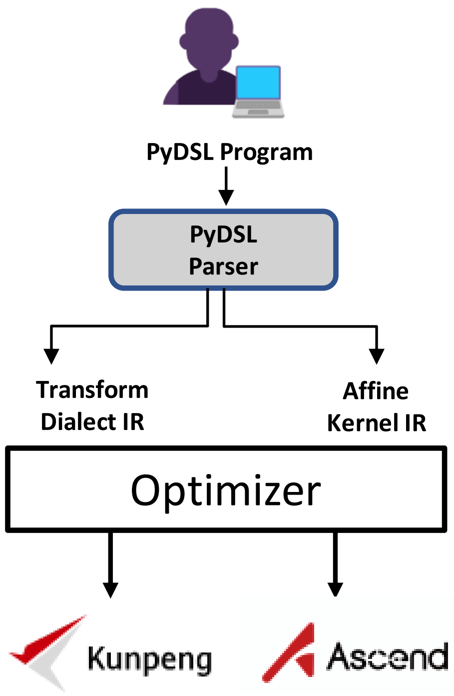
Motivation and principle

Motivation

- MLIR and MLIR Python binding is verbose. It is a compiler frontend’s output. Language users shouldn’t need to use it.
- Python is popular in the AI, scientific community.

Requirements

- Be directly usable from Python.
- Adhere to default Python syntax as much as possible.
  - Ideally user can take an existing Python code and make minimal changes to run on PyDSL compiler.
- Facilitates heterogeneous code generation



Part I of the talk given by Kevin Lee in Open MLIR Meeting 12-21-2023:

Video: https://youtu.be/nmtHeRkl850
Slides: https://mlir.llvm.org/OpenMeetings/2023-12-21-PyDSL.pdf

```mlir
#map = affine_map<()[s0] -> (s0 - 1)>
module attributes {transform.with_named_sequence} {
  func.func public @heat(%arg0: index, %arg1: index, %arg2: memref<?x?x?xf32>, %arg3: memref<?x?x?xf32>) {
    %cst = arith.constant 2.000000e+00 : f32
    %cst_0 = arith.constant 1.250000e-01 : f32
    affine.for %arg4 = 0 to %arg0 {
      affine.for %arg5 = 1 to #map()[%arg1] {
        affine.for %arg6 = 1 to #map()[%arg1] {
          affine.for %arg7 = 1 to #map()[%arg1] {
            %0 = affine.load %arg2[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %1 = affine.load %arg2[%arg5 + 1, %arg6, %arg7] : memref<?x?x?xf32>
            %2 = affine.load %arg2[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %3 = arith.mulf %cst, %2 : f32
            %4 = arith.subf %1, %3 : f32
            %5 = affine.load %arg2[%arg5 - 1, %arg6, %arg7] : memref<?x?x?xf32>
            %6 = arith.addf %4, %5 : f32
            %7 = affine.load %arg2[%arg5, %arg6 + 1, %arg7] : memref<?x?x?xf32>
            %8 = arith.addf %6, %7 : f32
            %9 = affine.load %arg2[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %10 = arith.mulf %cst, %9 : f32
            %11 = arith.subf %8, %10 : f32
            %12 = affine.load %arg2[%arg5, %arg6 - 1, %arg7] : memref<?x?x?xf32>
            %13 = arith.addf %11, %12 : f32
            %14 = affine.load %arg2[%arg5, %arg6, %arg7 + 1] : memref<?x?x?xf32>
            %15 = arith.addf %13, %14 : f32
            %16 = affine.load %arg2[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %17 = arith.mulf %cst, %16 : f32
            %18 = arith.subf %15, %17 : f32
            %19 = affine.load %arg2[%arg5, %arg6, %arg7 - 1] : memref<?x?x?xf32>
            %20 = arith.addf %18, %19 : f32
            %21 = arith.mulf %cst_0, %20 : f32
            %22 = arith.addf %0, %21 : f32
            affine.store %22, %arg3[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
          }
        }
      } {fuse_1}
      affine.for %arg5 = 1 to #map()[%arg1] {
        affine.for %arg6 = 1 to #map()[%arg1] {
          affine.for %arg7 = 1 to #map()[%arg1] {
            %0 = affine.load %arg3[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %1 = affine.load %arg3[%arg5 + 1, %arg6, %arg7] : memref<?x?x?xf32>
            %2 = affine.load %arg3[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %3 = arith.mulf %cst, %2 : f32
            %4 = arith.subf %1, %3 : f32
            %5 = affine.load %arg3[%arg5 - 1, %arg6, %arg7] : memref<?x?x?xf32>
            %6 = arith.addf %4, %5 : f32
            %7 = affine.load %arg3[%arg5, %arg6 + 1, %arg7] : memref<?x?x?xf32>
            %8 = arith.addf %6, %7 : f32
            %9 = affine.load %arg3[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %10 = arith.mulf %cst, %9 : f32
            %11 = arith.subf %8, %10 : f32
            %12 = affine.load %arg3[%arg5, %arg6 - 1, %arg7] : memref<?x?x?xf32>
            %13 = arith.addf %11, %12 : f32
            %14 = affine.load %arg3[%arg5, %arg6, %arg7 + 1] : memref<?x?x?xf32>
            %15 = arith.addf %13, %14 : f32
            %16 = affine.load %arg3[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
            %17 = arith.mulf %cst, %16 : f32
            %18 = arith.subf %15, %17 : f32
            %19 = affine.load %arg3[%arg5, %arg6, %arg7 - 1] : memref<?x?x?xf32>
            %20 = arith.addf %18, %19 : f32
            %21 = arith.mulf %cst_0, %20 : f32
            %22 = arith.addf %0, %21 : f32
            affine.store %22, %arg2[%arg5, %arg6, %arg7] : memref<?x?x?xf32>
          }
        }
      } {fuse_2}
    } {tile}
    return
  transform.named_sequence @__transform_main(%arg0: !transform.any_op) {
    %0 = transform.structured.match attributes {fuse_1} in %arg0 : (!transform.any_op) ->!transform.any_op
    %1 = transform.structured.match attributes {fuse_2} in %arg0 : (!transform.any_op) ->!transform.any_op
    %2 = transform.validator.fuse %0 with %1 at 3 :
      (!transform.any_op, !transform.any_op) -> !transform.any_op
    %3 = transform.structured.match attributes {tile} in %arg0 : (!transform.any_op) -> !transform.any_op
    %4:8 = transform.validator.tile %3 {tile_sizes = [8, 8, 32, 32]} : (!transform.any_op) ->
      (!transform.any_op, !transform.any_op, !transform.any_op, !transform.any_op, !transform.any_op,
       !transform.any_op, !transform.any_op, !transform.any_op)
    transform.yield
  }
}
```

Verbosity comparison between MLIR and PyDSL

PyDSL

~3x Productivity Boost 🚀

Existing work and comparison

*Our goal is to provide a complete Python-to-MLIR **compiler** that is extensible to new dialects and is as Pythonic as possible without sacrificing descriptiveness.*

Below projects work in similar fields and languages but providing a complete compiler for a new language are not their goals.

Name	Features	Author/Organization
mlir-python-extras	Improve usability and reduce boilerplates in upstream MLIR Python binding API. Does not perform compilation. Allows running the emitted program directly in Python.	Maksim Levental
xDSL	Offers Python interface to define and transform new and existing MLIR dialects. Encourages multi-level compiler design. Aims to be a compiler toolkit for facilitating DSL development.	ExCALIBUR
PyMLIR	Offers Python interface to manipulate MLIR. Supports basic dialects but can be extended to custom dialects. Can be thought of as an alternative parser and emitter library besides MLIR Python binding.	Scalable Parallel Computing Laboratory

Overview of PyDSL

- **Supports multiple MLIR dialects**: `arith`, `scf`, `func`, `memref`, `affine`, `transform`
  - Mapped Pythonically to the language. E.g. `arith.AddOp(a, b)` becomes `a + b`
- **Support for compiling and calling the function directly from Python**
  - MemRef currently passed through Numpy arrays
- **High-level static typing support** with **type inference** in some cases
  - Can infer `symbols`, `dimensions`, `affine_map`, `affine_set` from affine dialect
- **Macro system** for extending the compiler with more dialects

Example of numpy array interaction

```python
import numpy as np
from pydsl.frontend import compile
from pydsl.affine import affine_range as arange
from pydsl.memref import DYNAMIC, MemRefFactory
from pydsl.type import Index, UInt64

MemRef64 = MemRefFactory((DYNAMIC, DYNAMIC), UInt64)


@compile()
def hello_memref(size: Index, m: MemRef64) -> MemRef64:
    o = size // 2

    for i in arange(size):
        m[1, i] = o
        m[i, i] = i + o

    return m


arr = np.zeros((8, 8), dtype=np.uint64)

print(hello_memref(8, arr))
```

Type inference

- **Static-typing**: require explicit definition of type in function arguments and return type.
- **Preliminary type inference**:
  - Types of operation outputs are inferred.
  - Types of non-annotated constants uses a polymorphic **Number** type which are lazily lowered. Lowering happens when a Number is used by MLIR
  - **Number** is not concrete: it does not show up in resulting IR if not used.

E.g.

```
def chained_imp(a: UInt64) -> UInt64:
        return a + (6 * 2) // (12 - a)
```

Generic Number type with no MLIR typing information

```
module {
  func.func public @chained_imp(%arg0: i64) -> i64 {
    %c12_i64 = arith.constant 12 : i64
    %0 = arith.subi %c12_i64, %arg0 : i64
    %c12_i64_0 = arith.constant 12 : i64
    %1 = arith.divui %c12_i64_0, %0 : i64
    %2 = arith.addi %arg0, %1 : i64
    return %2 : i64
  }
}
```

6*2 got precomputed

12 inferred to be i64 when it encounters another i64 in arith operations

Type inference

Affine dialect is an important use case for us:

- `affine_range` turns a Python for loop into an `affine.for` loop
- **`affine.if` operator** supported by passing **`affine_set`** to an if statement
- **`affine_map`** performs `affine.load`/**`store`** on MemRefs

Explicitly defining symbol/dimension/affine_map

```
def lu(v0: Index, arg1: MemRefF64) -> Index:
    for arg2 in arange(S(v0)):
        for arg3 in arange(D(arg2)):
            for arg4 in arange(D(arg3)):
                arg1[am(D(arg2), D(arg3))] =
                    arg1[am(D(arg2), D(arg3))]
                    - (arg1[am(D(arg2), D(arg4))]
                    * arg1[am(D(arg4), D(arg3))])
```

Indicates that this is an affine for loop

Indicates that the MemRef arg1 should be indexed by an affine map

Indicates that arg3 is a dimension

Implicitly defining symbol/dimension/affine_map

```
def lu(v0: Index, arg1: MemRefF32) -> Index:
    for arg2 in arange(v0):
        for arg3 in arange(arg2):
            for arg4 in arange(arg3):
                arg1[arg2, arg3] = arg1[arg2, arg3] - (
                    arg1[arg2, arg4] * arg1[arg4, arg3]
                )
```

Defining `affine.if` using `integer_set`

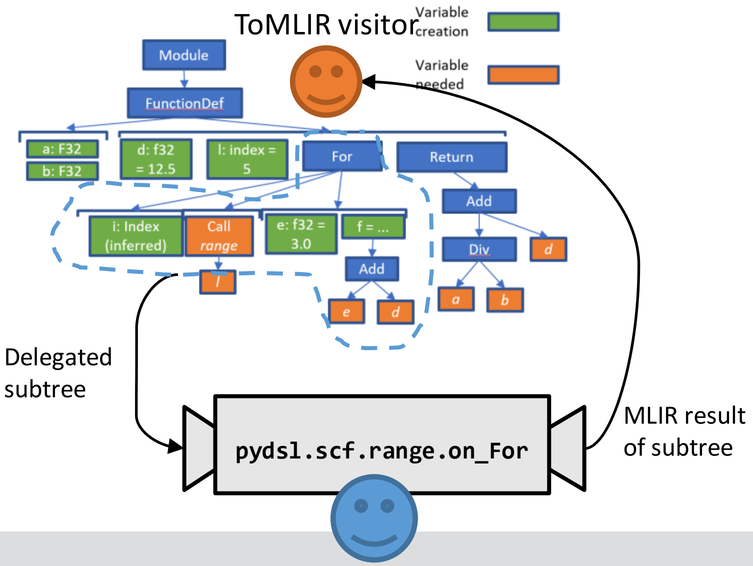
```
@compile(globals())
    def compare(m: MemRefCompare1, x: Index):
        if iset(6 <= x < 8 and x > 6 and 9 >= x):
            m[Index(0)] = UInt32(1)
        else:
            m[Index(0)] = UInt32(0)
```

Compiler extension through macro systems

```python
def heat(tsteps: Index, n: Index, A: MemF32, B: MemF32):
    a: F32 = 2.0
    b: F32 = 0.125
    for _ in arange(tsteps):
        """@tag("parallel")"""
        for i in arange(1, n-1):
            for j in arange(1, n-1):
                for k in arange(1, n-1):
                    B[i,j,k] = A[i,j,k] + b * (
                        A[i+1,j,k] - a * A[i,j,k] + A[i-1,j,k] + \
                        A[i,j+1,k] - a * A[i,j,k] + A[i,j-1,k] + \
                        A[i,j,k+1] - a * A[i,j,k] + A[i,j,k-1]
                    # ...
```

delegate For visitor compilation to the affine_range macro

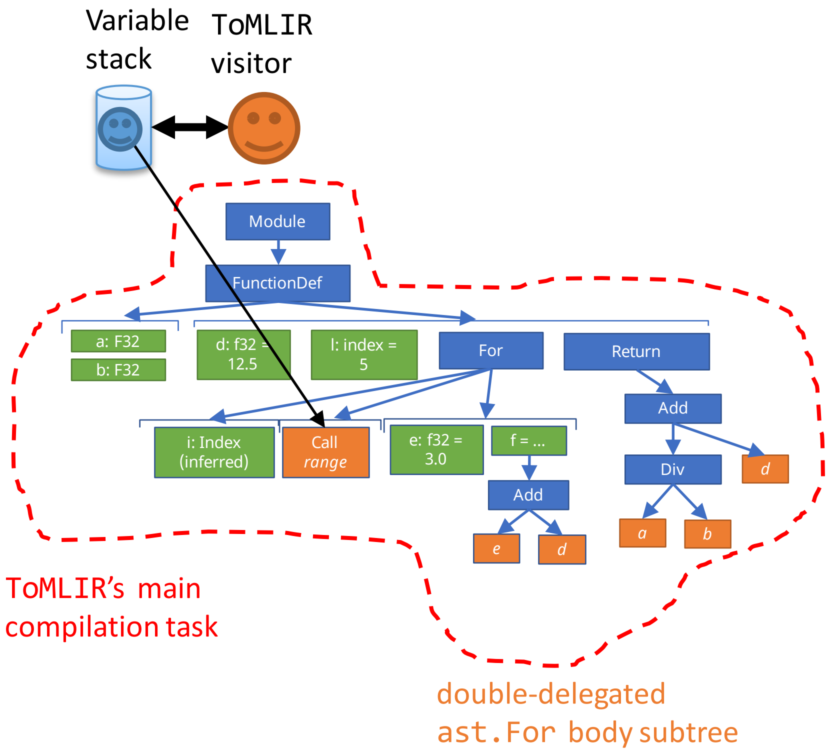
```python
class affine_range(IteratorMacro):
    def on_For(visitor: ToMLIRBase, node: ast.For) -> affine.AffineForOp:
        iter_arg = node.target
        iterator = node.iter

        lb = None
        step = 1

        args = iterator.args

        match len(args):
            case 1:
                ub = args[0]
            case 2:
                lb, ub = args
            case 3:
                lb, ub, step = args[0], args[1], args[2].value
            case _:
                # ...
```



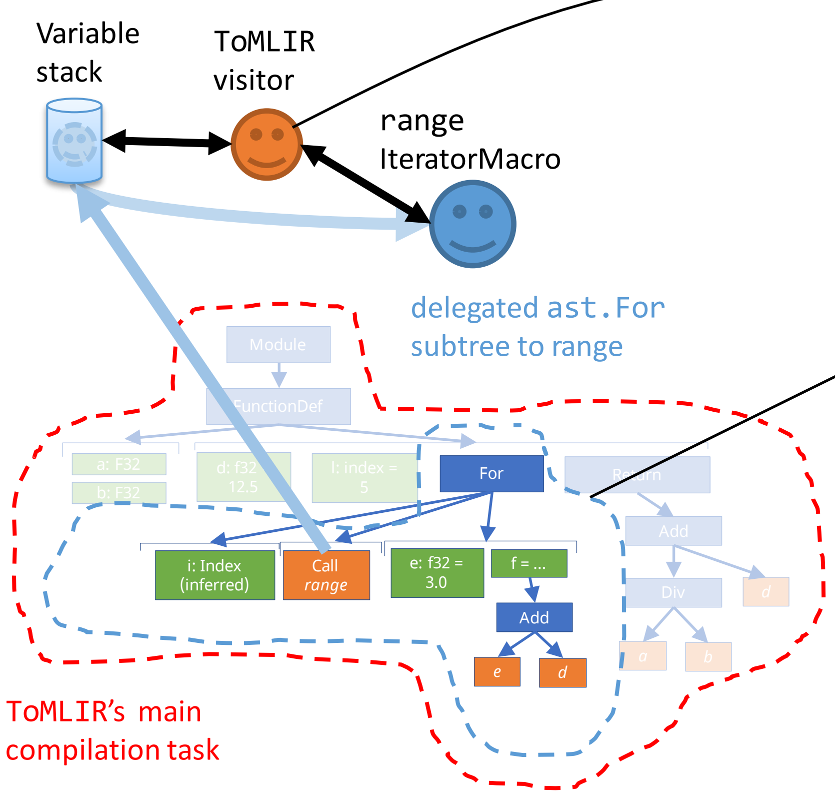
delegate Call visitor compilation to the tag macro

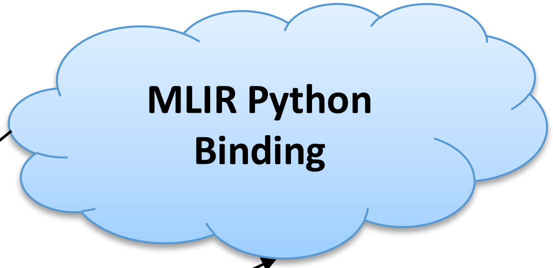
```python
@CallMacro.generate()
def tag(
    visitor: "ToMLIRBase", mlir: Compiled, attr_name: Evaluated[str]
) -> SubtreeOut:
    """
    Tags the `mlir` MLIR operation with a MLIR unit attribute with name
    `attr_name`.

    Arguments:
    - `mlir`: AST. The AST node whose equivalent MLIR Operator is to be tagged
      with the unit attribute
    - `attr_name`: str. The name of the unit attribute
    """
    target = get_operator(mlir)

    if type(attr_name) is not str:
        raise TypeError("Attribute name is not a string")

    target.attributes[attr_name] = UnitAttr.get()
    return mlir
```

Example: range IteratorMacro



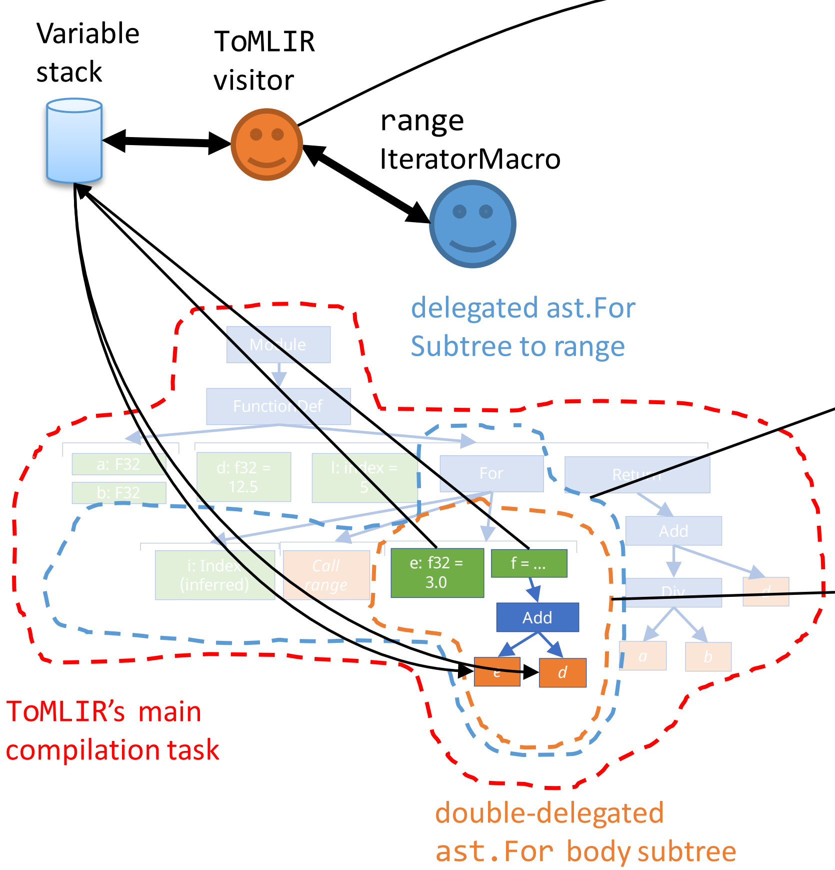
Example: range IteratorMacro






```
class range(IteratorMacro):
    def on_For(
        visitor: ToMLIRBase,
        node: ast.For) -> scf.ForOp:
        ...

for_op = scf.ForOp(start, stop, step)

assert type(iter_arg) is not ast.Tuple

with InsertionPoint(for_op.body):
            visitor.scope_stack.assign_name(
                iter_arg.id, Index(for_op.induction_variable)
            )

            for n in node.body:
                visitor.visit(n)

            # TODO: nothing will be yielded for now
            scf.YieldOp([])
```

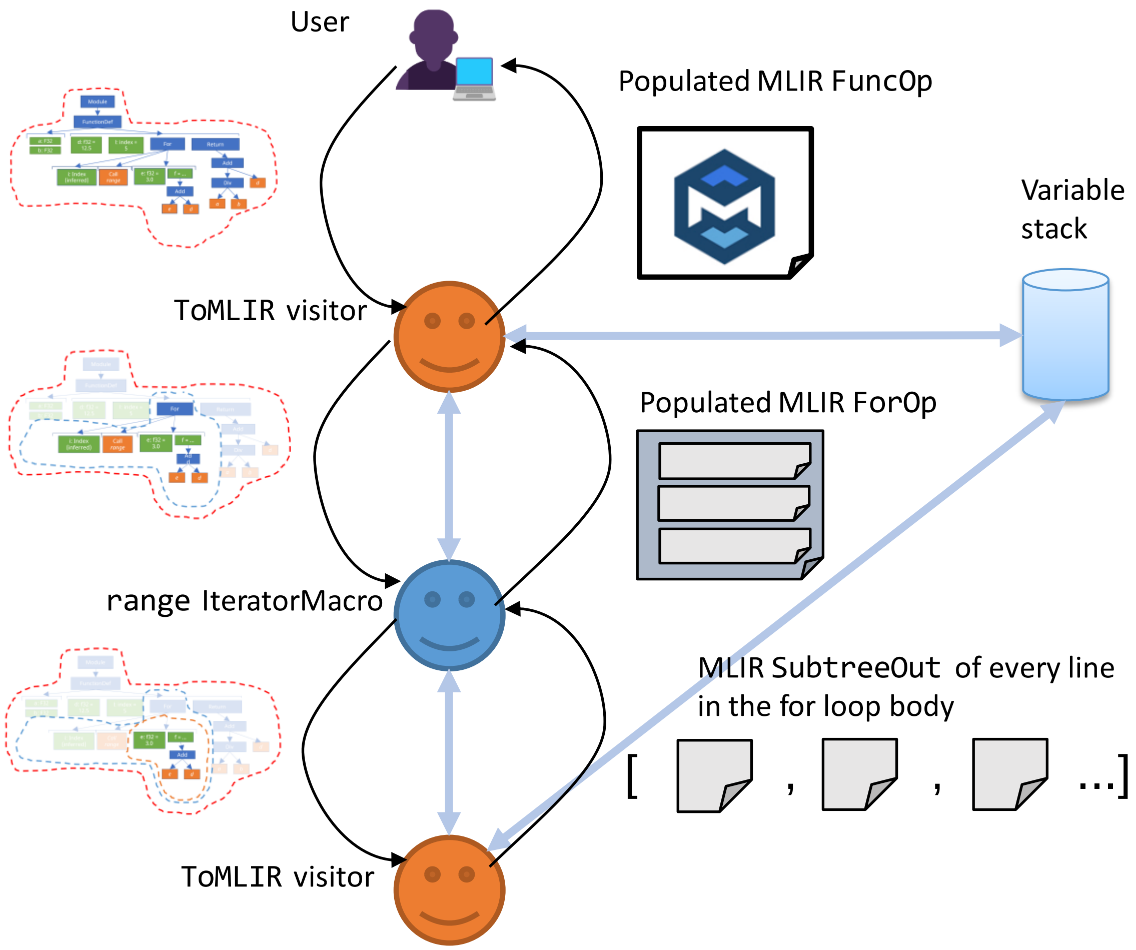
Example: range IteratorMacro



Definition of the range IteratorMacro (abridged)

```
class range(IteratorMacro):
    def on_For(
        visitor: ToMLIRBase,
        node: ast.For) -> scf.ForOp:
        ...

    with InsertionPoint(for_op.body):
            visitor.scope_stack.assign_name(
                iter_arg.id, Index(for_op.induction_variable)
            )

            for n in node.body:
                visitor.visit(n)

            # TODO: nothing will be yielded for now
            scf.YieldOp([])
```

Example: range IteratorMacro

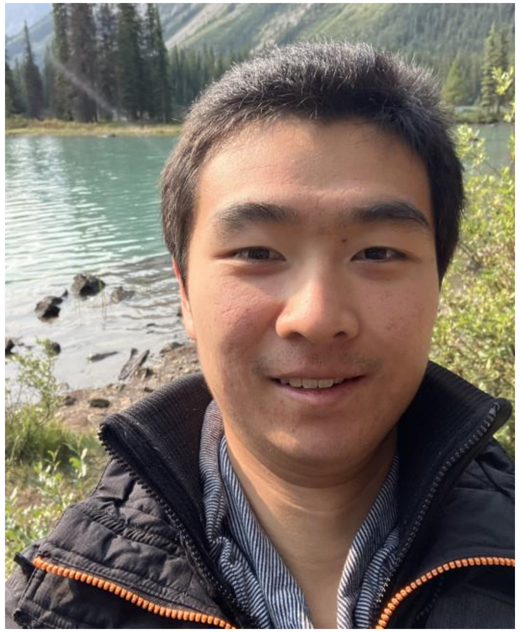


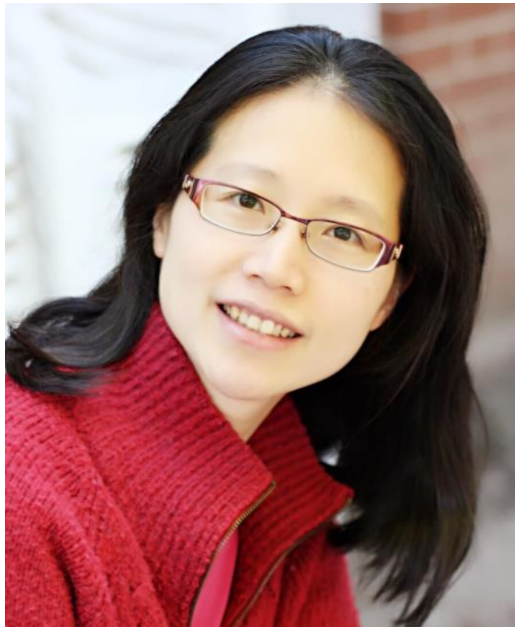
Why do we need Macro system?

- Lets you extend new features to ToMLIR
- ToMLIR is meant to cover basic Python syntax
- Macros cover anything domain-specific

Metaphor as a dialect

- PyDSL compiler = PyDSL "dialect"
- Macro = custom "operations"
- Macro member functions = "passes" that lowers "operations" to MLIR dialects to be supported





Kevin Lee
k323lee@uwaterloo.ca

Kai-Ting Wang
kai.ting.wang@huawei.com

Try out PyDSL

https://github.com/Huawei-CPLLab/PyDSL