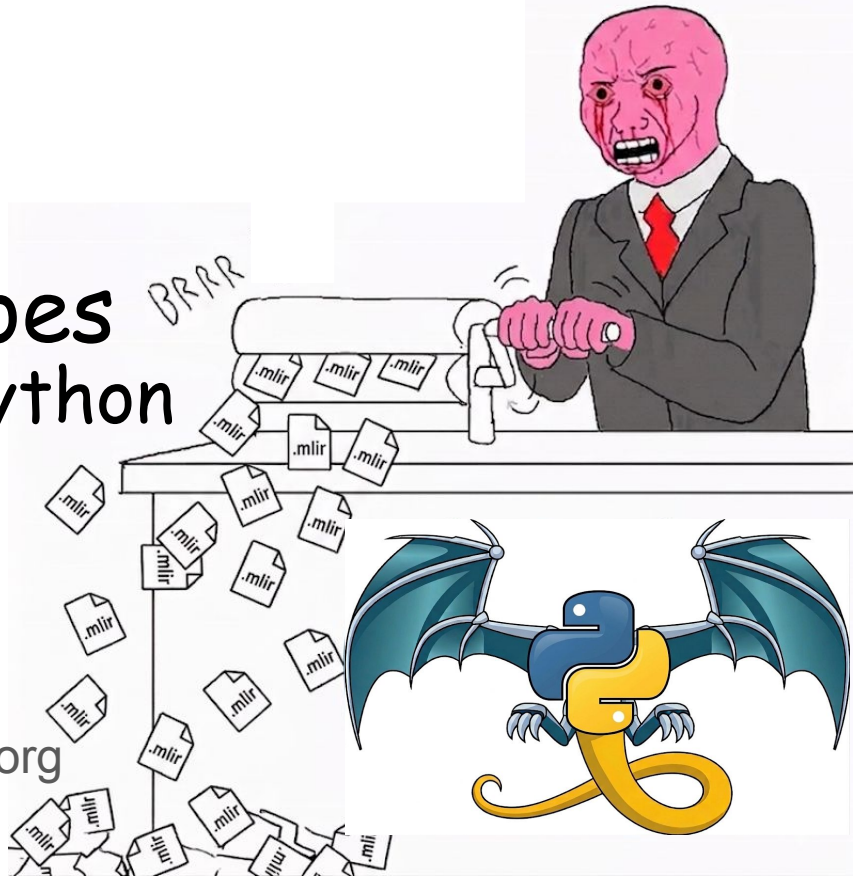
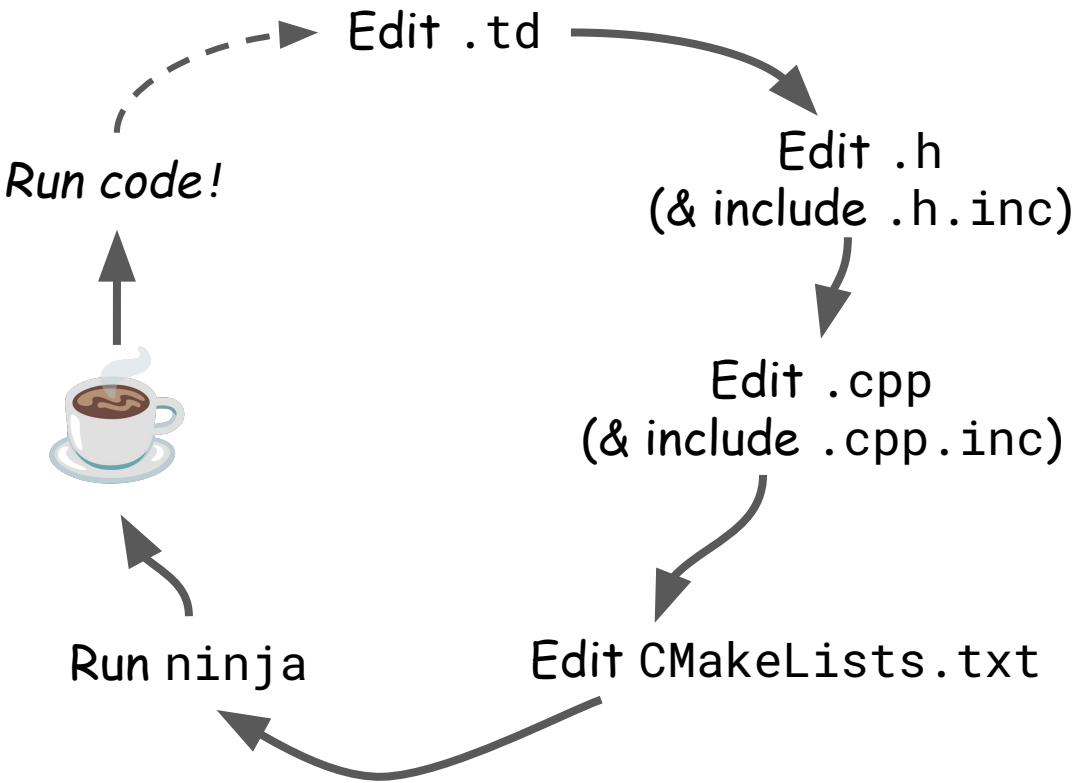
MLIR-iteration cycle goes
defining ops and rewrites in Python

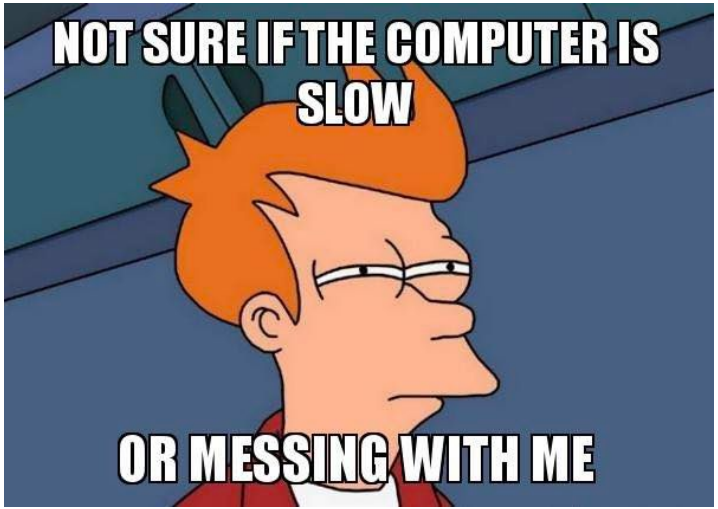


EuroLLVM 2026, Dublin

Rolf Morel, @rolfmorel, rolf.morel@intel.com
Mingyang Liu, @PragmaTwice, twice@apache.org

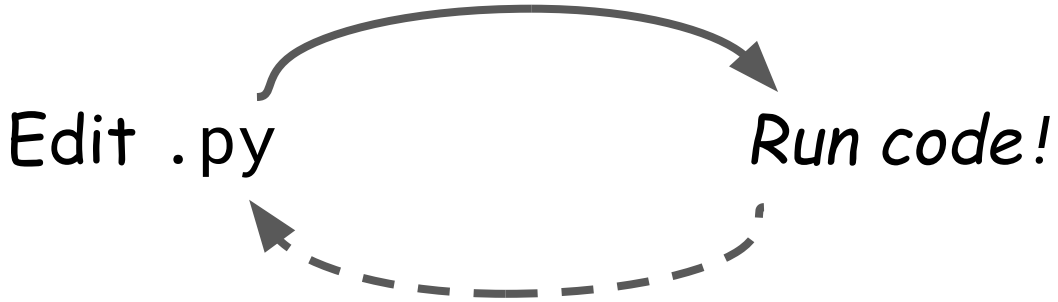
The problem: TableGen & C++-based iteration cycle





Disclaimer: this presentation makes transformative *use of meme templates and GenAI*

The solution*: Python-based iteration cycle



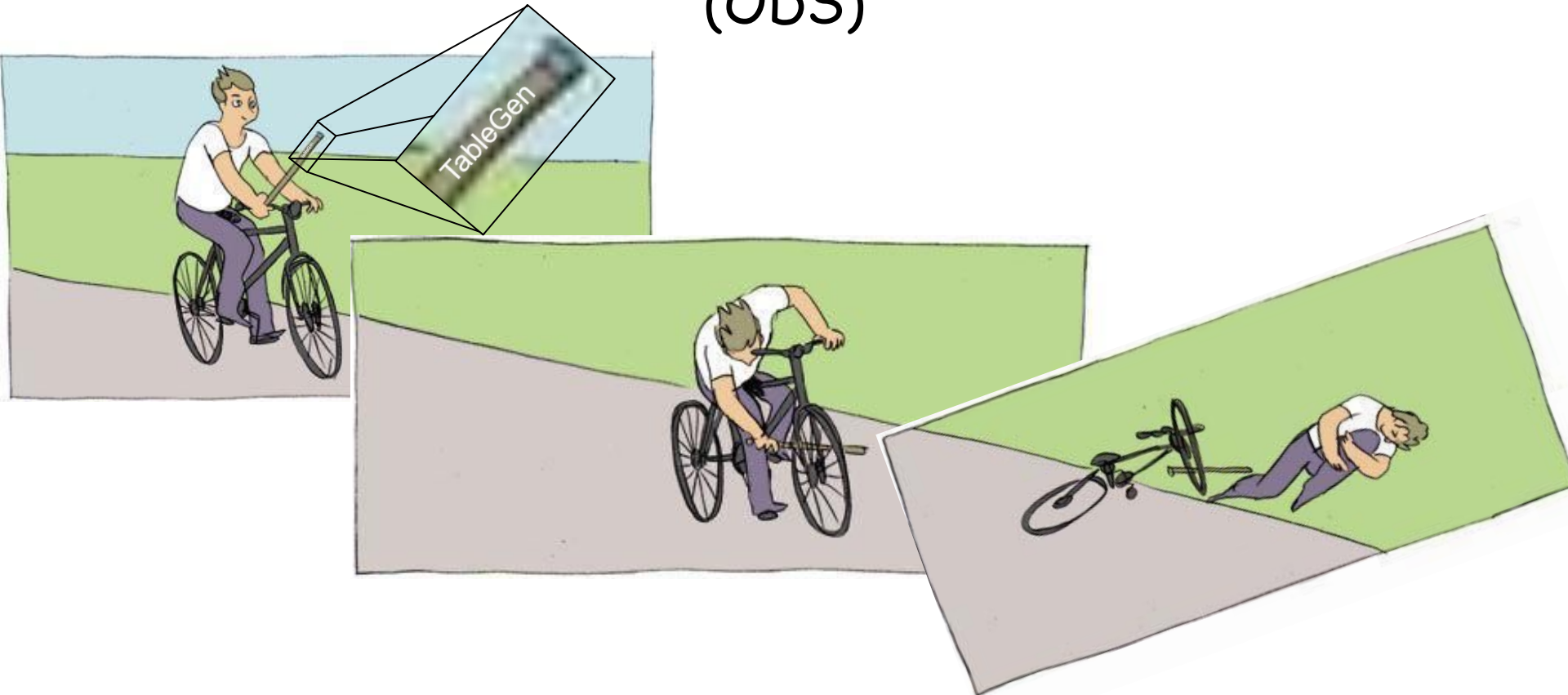
*: for prototyping

Writing your MLIR compiler in TableGen & C++

Writing your MLIR compiler in Python

A recap

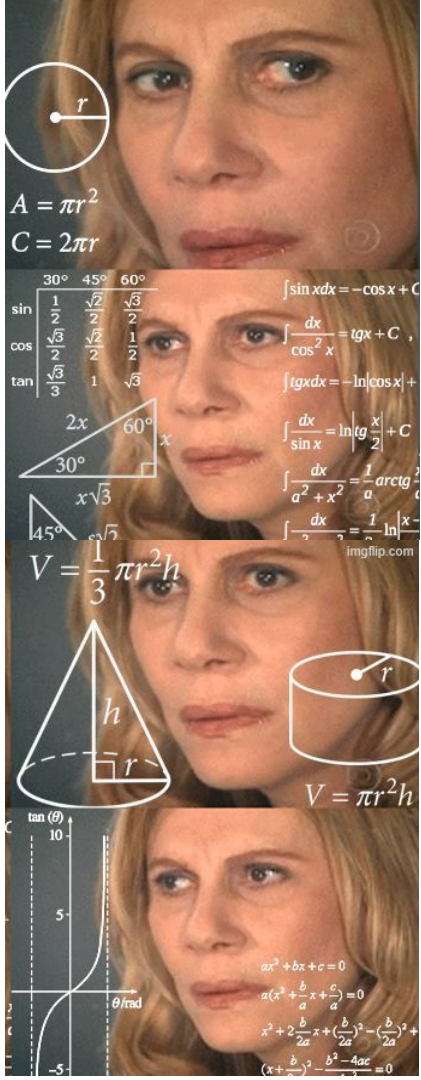
Encounters with the Operation Definition Specification (ODS)



Defining ops via ODS … i.e., with TableGen

mlir/include/mlir/Dialect/Arith/IR/ArithOps.td

```
class BooleanConditionOrMatchingShape<string condition, string result> :
    PredOpTrait<
      condition # " is signless i1 or has matching shape",
      Or<[TypeIsPred<condition, I1>,
          And<[SubstLeaves<"$_self", "$" # condition # ".getType()", IsShapedTypePred>,
               SubstLeaves<"$_self", "$" # result # ".getType()", IsShapedTypePred>,
               AllShapesMatch<[condition, result]>.predicate]>]>>;
def SelectOp : Arith_Op<"select", [Pure,
    AllTypesMatch<["true_value", "false_value", "result"]>,
    BooleanConditionOrMatchingShape<"condition", "result">,
    DeclareOpInterfaceMethods<InferIntRangeInterface, ["inferResultRangesFromOptional"]>,
    DeclareOpInterfaceMethods<SelectLikeOpInterface>]> {
  let summary = "select operation";

  let arguments = (ins BoolLike:$condition,
                       AnyType:$true_value,
                       AnyType:$false_value);
  let results = (outs AnyType:$result);
  let hasCanonicalizer = 1;
  let hasFolder = 1;
  let hasVerifier = 1;
  let hasCustomAssemblyFormat = 1;
}
```



Defining ops via ODS … which then generates

mlir/include/mlir/Dialect/Arith/IR/Arith.h

```
...
#include "mlir/Dialect/Arith/IR/ArithOps.h.inc"
```

mlir/lib/Dialect/Arith/IR/ArithOps.cpp

```
...
#include "mlir/Dialect/Arith/IR/ArithOps.cpp.inc"
```

```
...
SelectOp SelectOp::create(::mlir::OpBuilder &builder,
::mlir::Location location, ::mlir::ValueRange operands, const
Properties &properties, ::llvm::ArrayRef<::mlir::NamedAttribute>
discardableAttributes) {
  ::mlir::OperationState __state__(location, getOperationName());
  build(builder, __state__,
std::forward<decltype(operands)>(operands),
std::forward<decltype(properties)>(properties),
std::forward<decltype(discardableAttributes)>(discardableAttributes
));
  auto __res__ =
::llvm::dyn_cast<SelectOp>(builder.create(__state__));
  assert(__res__ && "builder didn't return the right type");
  return __res__;
}
...
```

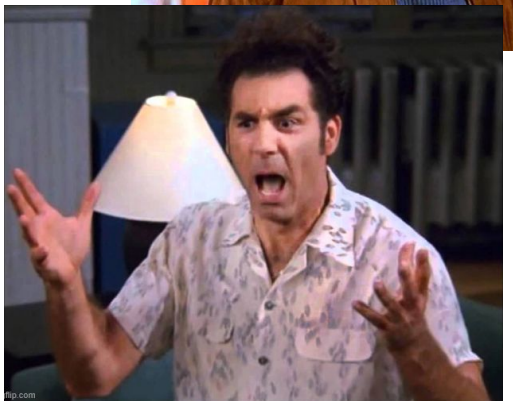
```
...
class SelectOpGenericAdaptorBase {
public:
  using Properties = ::mlir::EmptyProperties;
protected:
  ::mlir::DictionaryAttr odsAttrs;
  ::std::optional<::mlir::OperationName> odsOpName;
  Properties properties;
  ::mlir::RegionRange odsRegions;
public:
  SelectOpGenericAdaptorBase(::mlir::DictionaryAttr attrs, const
Properties &properties, ::mlir::RegionRange regions = {}) :
odsAttrs(attrs), properties(properties),
      odsRegions(regions) {  if (odsAttrs)
      odsOpName.emplace("arith.select",
                        odsAttrs.getContext());
...
```

Defining ops via ODS … which then (re-)compiles

```
$ ed mlir/.../ArithOps.cpp # change single char
$ ninja mlir-check
[0/586] Building CXX object tools/mlir/…
```

```
$ ed mlir/.../Arith/Utils/Utils.h # change single char
$ ninja mlir-check
[0/801] Building CXX object tools/mlir/…
```

```
$ ed mlir/.../ArithOps.td # change single char
$ ninja mlir-check
[0/1278] Building CXX object tools/mlir/…
```

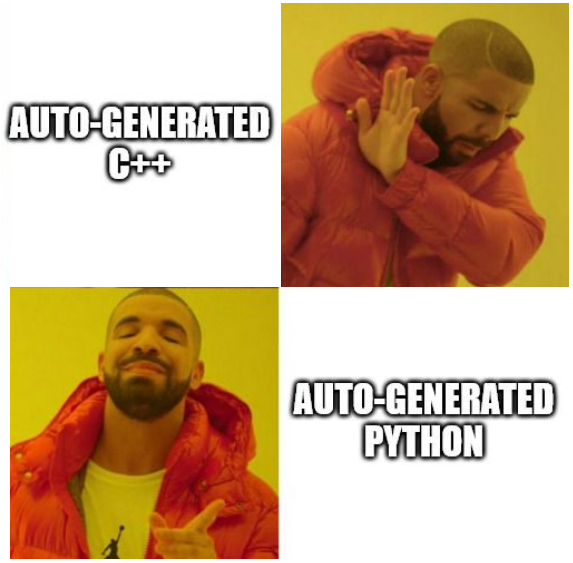


which brings us to …

Auto-generated Python bindings for ODS ops

build/tools/mlir/python/dialects/_arith_ops_gen.py

```python
@_ods_cext.register_operation(_Dialect)
class SelectOp(_ods_ir.OpView):
  OPERATION_NAME = "arith.select"

  _ODS_REGIONS = (0, True)

  def __init__(self, condition, true_value, false_value, *,
               results=None, loc=None, ip=None):
    operands = []
    attributes = {}
    regions = None
    operands.append(condition)
    operands.append(true_value)
    operands.append(false_value)
    _ods_context = _ods_get_default_loc_context(loc)
    _ods_successors = None
    super().__init__(self.OPERATION_NAME, self._ODS_REGIONS,
                     self._ODS_OPERAND_SEGMENTS, self._ODS_RESULT_SEGMENTS,
                     attributes=attributes, results=results,
                     operands=operands, successors=_ods_successors,
                     regions=regions, loc=loc, ip=ip)
  @builtins.property
  def condition(self) -> _ods_ir.Value:
    return self.operation.operands[0]
```



IR generation using Python bindings

.mlir

```
module {
  func.func @fma(%arg0: f32, %arg1: f32, %arg2: f32) -> f32 {
    %0 = arith.mulf %arg1, %arg2 : f32
    %1 = arith.addf %arg0, %0 : f32
    return %1 : f32
  }
}
```

.py

```
module = Module.create()
with InsertionPoint(module.body):
    f32 = F32Type.get()
    @func.func(f32, f32, f32)
    def fma(x, y, z):
        return arith.addf(x, arith.mulf(y, z))
```

[Using MLIR from C and Python by Alex Zinenko]

IR traversal using Python bindings

.py

```python
def traverse_ir(op: Operation):
    for region in op.regions:
        for block in region.blocks:
            for nested in block.operations:
                print(nested.attributes["my_attr"])
```

Taken from: Using MLIR from C and Python by Alex Zinenko

Basic IR manipulation using Python bindings

.py

```
operation.attributes["attr_name"] = ir.IntegerAttr.get(...)  # Attributes are assignable

operation.operands[0] = some_value                           # Operand list items are assignable

operation.operands[1].set_type(ir.SomeType.get(...))         # Types can be updated

operation.regions.append(...)
operation.regions[0].blocks.append(...)                      # Linked lists can be appended to


operation.erase()                                            # Operations can be erased
```

Taken from: [Using MLIR from C and Python by Alex Zinenko]

Apply C++-defined passes to IR

.py

```python
from mlir.passmanager import PassManager

pm = PassManager('any', module.context)
pm.add('convert-to-llvm')
pm.run(module.operation)

print(module)
```

.mlir

```
module {
  func.func @fma(%arg0: f32, %arg1: f32,
                 %arg2: f32) -> f32 {
    %0 = arith.mulf %arg1, %arg2 : f32
    %1 = arith.addf %arg0, %0 : f32
    return %1 : f32
  }
}
```

.mlir

```
module {
  llvm.func @fma(%arg0: f32, %arg1: f32,
                 %arg2: f32) -> f32 {
    %0 = llvm.fmul %arg1, %arg2 : f32
    %1 = llvm.fadd %arg0, %0 : f32
    llvm.return %1 : f32
  }
}
```

Using MLIR from C and Python by Alex Zinenko

Execute IR via ExecutionEngine

1 2 3

.py

```python
from mlir.execution_engine import ExecutionEngine
import ctypes

with Context():
    ee = ExecutionEngine(module)
    c_float_p = ctypes.c_float * 1
    res = c_float_p(0)
    ee.invoke("fma", c_float_p(1),
              c_float_p(2), c_float_p(3), res)
    assert res[0] == 7
```

.mlir

```
module {
  llvm.func @fma(%arg0: f32, %arg1: f32,
                 %arg2: f32) -> f32 {
    %0 = llvm.fmul %arg1, %arg2 : f32
    %1 = llvm.fadd %arg0, %0 : f32
    llvm.return %1 : f32
  }
}
```

7

What's new?

What's new?

Use Python to

1. Define rewrite patterns & conversion patterns
2. Define passes
3. Define ops & dialects
4. Define types & attributes
5. Implement op traits & interfaces

Defining rewrite patterns

```python
def addi_to_muli(op, rewriter):
    assert isinstance(op, arith.AddIOp)
    with rewriter.ip:
        new_op = arith.muli(op.lhs, op.rhs, loc=op.location)
    rewriter.replace_op(op, new_op.owner)
```

A matchAndRewrite function!

```python
patterns = RewritePatternSet()
patterns.add(arith.AddIOp, addi_to_muli)
patterns.add(...)

apply_patterns_and_fold_greedily(module, patterns.freeze())
```

Add a new rewrite pattern for `arith.addi`

Apply the patterns

Defining conversion patterns

```
converter = TypeConverter()
converter.add_conversion(...)
```

```
def convert_addi(op, adaptor, type_converter, rewriter):
    assert isinstance(op, arith.AddIOp)
    assert isinstance(adaptor, arith.AddIOpAdaptor)
    with rewriter.ip:
        new_op = smt.IntAddOp([adaptor.lhs, adaptor.rhs], loc=op.location)
    rewriter.replace_op(op, new_op)
```

A `matchAndRewrite` function! Use the adaptor class just like in C++

```
patterns = RewritePatternSet()
patterns.add_conversion(arith.AddIOp, convert_addi, converter)
patterns.add_conversion(...)
```

Add a new conversion pattern for `arith.addi` with that function and the type converter

```
target = ConversionTarget()
target.add_legal_dialect(smt._Dialect)
target.add_illegal_op(arith.AddIOp, ...)
```

Add legal/illegal ops/dialects via Python class names

```
apply_partial_conversion(module, target, patterns.freeze())
```

Defining passes

A handle to control the pass, e.g. signal failures

```
def custom_pass(op, pass_):
    # do something with `op`
    pass
```

A pass is just a Python function! No ODS needed

```
pm = PassManager("any")

pm.add(custom_pass)
pm.add("canonicalize, cse, convert-to-llvm, ...")

pm.run(module)
```

Mix C++-defined passes and Python-defined passes in your pass pipeline

Defining dialects and ops - Python vs ODS

.py

```python
class Demo(Dialect, name="demo"):
    pass
```

```python
class ConstantOp(Demo.Operation, name="constant"):
    value: IntegerAttr
    cst:   Result[IntegerType[32]] = infer_result()


class AddOp(Demo.Operation, name="add"):
    lhs: Operand[IntegerType[32]]
    rhs: Operand[IntegerType[32]]
    res: Result[IntegerType[32]] = infer_result()
```

.td

```
def Demo_Dialect : Dialect {
  let name = "demo";
  let cppNamespace = "::mlir::demo";
}

class Demo_Op<string mnemonic, list<Trait> traits = []> :
    Op<Demo_Dialect, mnemonic, traits>;
```

dialect

ops

```
def Demo_ConstantOp : Demo_Op<"constant"> {
  let arguments = (ins APIntAttr:$value);
  let results = (outs I32:$cst);
}

def Demo_AddOp : Demo_Op<"add"> {
  let arguments = (ins I32:$lhs, I32:$rhs);
  let results = (outs I32:$res);
}
```

Defining ops - à la Python's @dataclass syntax

operands/results/attrs constraints as type annotations

.py

"fields"

```python
class LeakyReluOp(Demo.Operation, name="leaky_relu"):
    operand: Operand[F32Type]                    # SSA-`Value` argument
    alpha:   FloatAttr                           # attribute argument
    result:  Result[F32Type] = infer_result()    # infers result type


f32 = F32Type.get()
val: Value = arith.constant(f32, -42.67)
lerelu = LeakyReluOp(val, alpha=FloatAttr.get(f32, 0.1))
assert isinstance(lerelu.result, Value) and lerelu.alpha == 0.1
```

field specifiers for __init__ parameters

emits

.mlir

```
%val = arith.constant -4.267000e+01 : f32
%result = "demo.leaky_relu"(%val) {alpha = 1.000000e-01 : f32} : (f32) -> f32
```

Defining types

```
class Array(Demo.Type, name="array"):
    elem_type: IntegerType[32] | IntegerType[64]
    length: IntegerAttr
```

One of i32 and i64

Any integer attribute

Result's type is any valid `Array`

```
class MakeArrayOp(Demo.Operation, name="make_array"):
    arr: Result[Array]
    len3_arr_of_i32s: Result[Array[
        IntegerType[32], IntegerAttr[IntegerType[32], 3]
    ]] = infer_result()
```

A result with a particular `Array` type specified

```
print(Array.get(i32, IntegerAttr.get(i32, 3)))
```

Defining attributes

```
class StrPairAttr(Demo.Attribute, name="str_pair"):
    first: StringAttr
    second: StringAttr
```

Any valid StrPairAttr

```
class TakeStrPairsOp(Demo.Operation, name="take_str_pairs"):
    pair: StrPairAttr
    ab_pair: StrPairAttr[StringAttr["a"], StringAttr["b"]]
```

A particular StrPairAttr specified, with arguments ("a", "b")

```
print(StrPairAttr.get(StringAttr.get("hello"), StringAttr.get("world")))
```

Defining ops & types & attrs … via IRDL

```
class LeakyReluOp(Demo.Operation,
                  name="leaky_relu"):
    operand: Operand[F32Type]
    alpha:   FloatAttr
    result:  Result[F32Type] = infer_result()
```

```
irdl.operation @leaky_relu {
    %0 = irdl.base "!builtin.f32"
    irdl.operands(input: %0)
    %1 = irdl.base "#builtin.float"
    irdl.attributes {"alpha" = %1}
    %2 = irdl.is f32
    irdl.results(res: %2)
}
```

Implementation detail as far as users are concerned!

IRDL: An IR Definition Language for SSA Compilers by Fehr et al

Defining & implementing traits on ops

```python
class ParentIsIfTrait(DynamicOpTrait):
    @staticmethod
    def verify_invariants(op) -> bool:
        if not isinstance(op.parent.opview, IfOp):
            op.location.emit_error(
                f"{op.name} should be put inside ..."
            )
            return False
        return True
```

Define the trait's verification via attaching a `@staticmethod`

Declare traits for your op here

```python
class YieldOp(
    Demo.Operation, name="yield",
    traits=[IsTerminatorTrait, ParentIsIfTrait]
):
    value: Operand[Any]

    def verify_invariants(self) -> bool:
        if self.parent.results[0].type != self.value.type:
            self.location.emit_error(
                "result type mismatch .."
            )
            return False
        return True
```

Add op-specific verification logic

Implement interfaces on ops: TransformOpInterface

.py

```
class GetNamedAttributeOp(TransformExt.Operation,
                          name="get_named_attribute"):
    target: Operand[transform.AnyOpType]
    attr_name: StringAttr
    param: Result[transform.AnyParamType[()]] = infer_result()

class TransformOpInterfaceExternalModel(transform.TransformOpInterface):
    @staticmethod
    def apply(op: GetNamedAttributeOp, _rewriter, results, state):
        target_op = state.get_payload_op(op.target)
        if assoc_attr := target_op.attributes.get(op.attr_name.value):
            results.set_param(op.param, [assoc_attr])
            return Success
        return SilenceableFailure

TransformOpInterfaceExternalModel.attach(GetNamedAttributeOp.OPERATION_NAME)
```

More examples in [llvm/lighthouse](), including op which interprets SMT ops in body

But what is it good for?!

A BrainFuck compiler

.py

```
class BfDialect(Dialect, name="bf"):
   pass

class PtrType(BfDialect.Type, name="ptr"):
   pass

class NextOp(BfDialect.Operation, name="next"):
   in_: Operand[PtrType]
   out: Result[PtrType[()]] = infer_result()

class WhileOp(BfDialect.Operation, name="while"):
   in_: Operand[PtrType]
   out: Result[PtrType[()]] = infer_result()
   body: Region

...
```

Define a region just like other fields!

character	Operation
>	bf.next(ptr) -> ptr
<	bf.prev(ptr) -> ptr
+	bf.inc(ptr)
-	bf.dec(ptr)
.	bf.output(ptr)
,	bf.input(ptr)
[	bf.while (ptr, region) -> ptr
]	bf.yield(ptr)

A BrainFuck compiler

.py

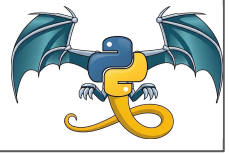
```python
def convert_bf_to_upstream(op, pass_):
    type_converter = TypeConverter()
    type_converter.add_conversion(convert_ptr_to_llvm_ptr)

    ... # define conversion functions
    patterns = RewritePatternSet()
    patterns.add_conversion(NextOp, convert_next, type_converter)
    patterns.add_conversion(...)  # add conversion patterns

    target = ConversionTarget()
    target.add_illegal_dialect(BfDialect)
    apply_partial_conversion(op, target, patterns.freeze())

    with InsertionPoint(op.opview.body):
        func.FuncOp("putchar", FunctionType.get([i32], [i32]))
        func.FuncOp("getchar", FunctionType.get([], [i32]))
        ... # insert init & lib functions
```

.mlir

```
"bf.main"() ({
^bb0(%arg0: !bf.ptr):
 %0 = "bf.while"(%arg0) ({
 ^bb0(%arg1: !bf.ptr):
   "bf.dec"(%arg1) : (!bf.ptr) -> ()
   "bf.yield"(%arg1) : (!bf.ptr) -> ()
 }) : (!bf.ptr) -> !bf.ptr
 "bf.yield"(%0) : (!bf.ptr) -> ()
}) : () -> ()
```

.mlir

```
func.func @bf_main(%arg0: !llvm.ptr) -> !llvm.ptr {
 %0 = scf.while (%arg1 = %arg0) : (!llvm.ptr) -> !llvm.ptr {
   %1 = llvm.load %arg1 : !llvm.ptr -> i8
   %2 = llvm.mlir.constant(0 : i8) : i8
   %3 = llvm.icmp "ne" %1, %2 : i8
   scf.condition(%3) %arg1 : !llvm.ptr
 } do {
 ^bb0(%arg1: !llvm.ptr):
   %1 = llvm.load %arg1 : !llvm.ptr -> i8
   %2 = llvm.mlir.constant(-1 : i8) : i8
   %3 = llvm.add %1, %2 : i8
   llvm.store %3, %arg1 : i8, !llvm.ptr
   scf.yield %arg1 : !llvm.ptr
 }
 return %0 : !llvm.ptr
}
```



A BrainFuck compiler

.py

```python
def execute(bf_code):
    module: Module = parse(bf_code)
    assert module.operation.verify()

    pm = PassManager()
    pm.add(convert_bf_to_upstream)
    pm.add("convert-scf-to-cf, convert-to-llvm")

    pm.run(module.operation)

    ee = ExecutionEngine(module)
    ee.lookup("bf_init")(0)
```

.py

```
execute(
```

.bf

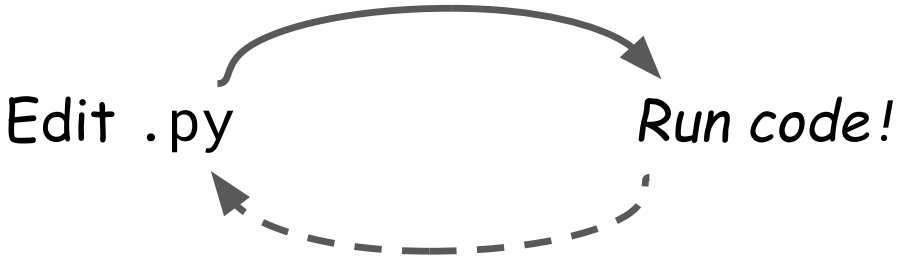
```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<
-]>>.>---.+++++++..+++.>>.<-.<.+++.------.-----
---.>>+.>++.
```

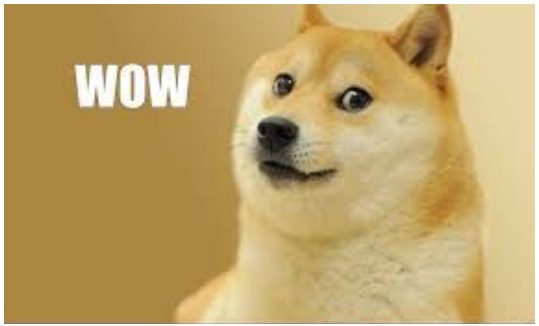
```
)
```

Hello World!

In all its glory: [mlir/test/python/integration/dialects/bf.py]

The solution: Python-based iteration cycle*





1. No need to (re-)compile! Just need a build/an install of llvm-project
2. Quick PROTOTYPING, good for weak machines, e.g. volunteers!
3. Try things out AND THEN move/upstream to C++ codebase
4. Meta-programming?!
5. …
6. Profit!!

Why not try it?!

```
$ pip install mlir-python-bindings -f "https://llvm.github.io/eudsl"

$ python
>>> from mlir import ir
>>> with ir.Context():
...     print(ir.Module.parse("arith.constant 10"))
...
module {
  %c10_i64 = arith.constant 10 : i64
}
```

Many thanks to @makslevental and to all the LLVM & MLIR volunteers!

In conclusion