Core Transformations in MLIR

I.e. letting MLIR optimize our programs for us!

Markus Böck (ETH Zurich)
Maximilian Bartel (roofline)

ACM Europe School on MLIR 2026 – August 11, 2026

`RewritePatterns` are meant to be together!

```
$ tutorial-opt input.mlir -list-simplify -arith-simplify \
  -func-simplify -list-simplify -arith-simplify…
```

Not only cumbersome, but worse at optimizations!

RewritePatterns are meant to be together!

```
$ tutorial-opt input.mlir -arith-simplify -list-simplify
```

```
%0 = list.map %li with (%a: i32) -> i64 {
  %e = arith.extsi %a : i32 to i64
  list.yield %e : i64
}
%1 = list.map %0 with (%b: i64) -> i32 {
  %t = arith.trunci %b : i64 to i32
  list.yield %t : i32
}
return %1 : !list.list<i32>
```

```
%0 = list.map %li with (%a: i32) -> i64 {
  %e = arith.extsi %a : i32 to i64
  %t = arith.trunci %e : i64 to i32
  list.yield %t : i32
}
return %0 : !list.list<i32>
```

RewritePatterns are meant to be together!

```
$ tutorial-opt input.mlir -list-simplify -arith-simplify
```

```
%0 = list.map %li with (%a: i32) -> i64 {
  %e = arith.extsi %a : i32 to i64
  list.yield %e : i64
}
%1 = list.map %0 with (%b: i64) -> i32 {
  %t = arith.trunci %b : i64 to i32
  list.yield %t : i32
}
return %1 : !list.list<i32>
```

```
return %li : !list.list<i32>
```

`canonicalize` combines all your `RewritePatterns`

- **Goal:** Simplify the entire IR regardless of dialects present
- Automatically runs all `RewritePatterns` registered as canonicalizations

```
$ tutorial-opt input.mlir -canonicalize
```

```
%0 = list.map %li with (%a: i32) -> i64 {
  %e = arith.extsi %a : i32 to i64
  list.yield %e : i64
}
%1 = list.map %0 with (%b: i64) -> i32 {
  %t = arith.trunci %b : i64 to i32
  list.yield %t : i32
}
return %1 : !list.list<i32>
```

```
return %li : !list.list<i32>
```

Defining `RewritePatterns` as canonicalizations

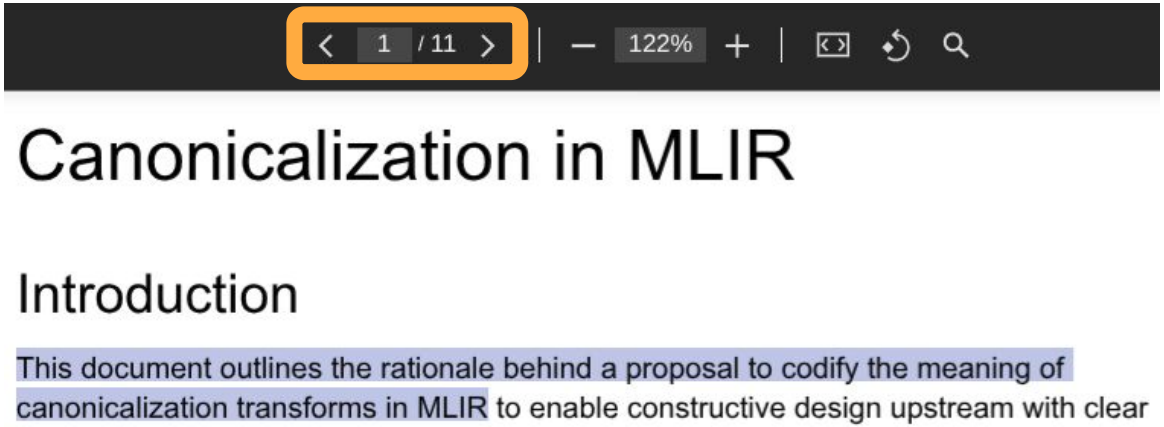
```
def List_MapOp : List_Op<"map", [...]> {
  …
  let hasCanonicalizer = 1;
  …
}
```

```
/// ListOps.cpp
void MapOp::getCanonicalizationPatterns(RewritePatternSet &results,
                                        MLIRContext *context) {
 results.add<YourRewritePatternsHere, NextPattern, …>(context);
}
```

What makes a `RewritePattern` a Canonicalization?

Canonical IR:“All programs that compute the same should have the same IR” 🌟

“All patterns that bring your IR into canonical IR should be run in canonicalization”



It’s a Design Choice

What makes a `RewritePattern` a Canonicalization?

Great Guidelines:

- Constant folding
  - `arith.addi 1, 0 -> 1`
- The less operations the better!
  - `addi(%x, muli(%y, -1)) → subi(%x, %y)`
- Is Information lost? Then No!
  - Litmus Test: *“Can I no longer perform an Optimization?”*
  - Otherwise: A *Lowering*
- Does it converge?

Things you don’t need to care about:

- Performance of the code!

Exercise 1: Porting from `list-simplify` to `canonicalize`

- `$ git checkout exercises/day-2-session-4`
- Think, which patterns we've written today are canonicalizations and which are lowerings?
  - `tutorial/list/Transforms/Exercise_2_3_4_ListSimplify.cpp`
  - `ReplaceEmptyFromElements`, `PeelFromElements`, `MergeConsecutiveMaps`, `LowerMapToWhileLoop`
  - Use the Criteria discussed previously if you want
  - Discuss with your neighbour if you like!
- Move these patterns such that they are run with the `canonicalize` pass
- Update tests that exercised these patterns to now use the `canonicalize` pass. (You probably want a new `canonicalize.mlir` file)

`RewritePatterns` can be verbose and overpowered

```
struct FoldFromElementsOfConstants : OpRewritePattern<FromElementsOp> {
 using OpRewritePattern::OpRewritePattern;

 LogicalResult matchAndRewrite(FromElementsOp op,
                               PatternRewriter &rewriter) const override {
   // ACTUAL LOGIC GOES HERE
 }
};
```

7 LOC of Boilerplate (+ registration)

`RewritePattern` can modify any Operation (non-local)!
Locality a nice guarantee for:

- `OpBuilder::createOrFold`
- Dataflow Analysis
- Any pass wanting immediate simplifications

Enabling the `fold` Method for your Operation

```
class Op<Dialect dialect, string mnemonic,
         list<Trait> props = []> {
 …
 // Whether this op has a folder.
 bit hasFolder = 1;
 …
}
```

Single Result Op

```
OpFoldResult MyOp::fold(FoldAdaptor adaptor) {
 ...
}
```

`return nullptr` on failure

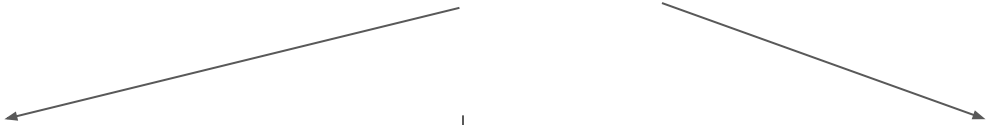
Otherwise

```
LogicalResult MyOp::fold(
      FoldAdaptor adaptor,
      SmallVectorImpl<OpFoldResult> &results) {
 ...
}
```

`return failure()` on failure (duh)

Local transformations permitted by `OpFoldResult`

```
class OpFoldResult : public PointerUnion<Value, Attribute>
```



Replaces the Op with the value:

```
OpFoldResult ReverseOp::fold(
      FoldAdaptor adaptor) {
 // list.reverse(list.reverse(%li)) -> %li
 if (auto producer = getInput()
                 .getDefiningOp<ReverseOp>())
   return producer.getInput();
 return nullptr;
}
```

Replaces the Op with a constant

```
%lb = arith.constant 1 : i32
%ub = arith.constant 4 : i32
%li = list.range %lb to %ub : !list.list<i32>
```

⬇️

```
%li = list.constant #list.list<[1, 2, 3]>
```

Checklist For Constant Folding in your Dialect!

1) Have an Attribute that can represent a constant of the result type

MLIR C++ Type	MLIR C++ Attribute
`IntegerType`	`IntegerAttr`
`IntegerType (i0)`	`BoolAttr`
`list::ListType`	`list::ListAttribute`

2) Have an Attribute that can represent a constant of the result type

```
def List_ConstantOp : List_Op<"constant", [ConstantLike]> {
 ...
 let arguments = (ins List_ListAttr:$value);
 let results = (outs List_ListType:$result);
 let hasFolder = 1;
}
```

Requirements

Checklist For Constant Folding in your Dialect!

3) The Dialect is responsible for materializing constants for its Operations!

```
def List_Dialect : Dialect {
 ...
 let hasConstantMaterializer = 1;
 let dependentDialects = [
   "::mlir::arith::ArithDialect"
 ];
}
```

```
Operation *ListDialect::materializeConstant(
   OpBuilder &builder, Attribute value) {
 if (auto listValue = dyn_cast<ListAttr>(value))
   return ConstantOp::create(builder, listValue);

 // Some operations have integer results!
 return arith::ConstantOp::materialize(
      builder, value);
}
```

The `FoldAdaptor` Parameter for Constant Folding

- Generated by ODS together with the Op as a mirror for folding

```
class PushBackOp : mlir::Op<...> {
public:
 …
 Value getInput();
 Value getItem();
 …
};
```

```
class PushBackOp::FoldAdaptor {
public:
 // Not-null if input is a constant.
 Attribute getInput();
 // Not-null if item is a constant.
 Attribute getItem();
};
```

- Use together with `dyn_cast_if_present` to check if operand is constant

```
OpFoldResult PushBackOp::fold(FoldAdaptor adaptor) {
 ListAttr constant = dyn_cast_if_present<ListAttr>(adaptor.getInput());
 auto item = dyn_cast_if_present<IntegerAttr>(adaptor.getItem());
 if (!constant || !item)
   return {};
```

Exercise 2: Write fold implementation ops

- Implement `fold` for at least the following Ops: `list.length`, `list.is_empty` and `list.reverse`
  - Focus on constant folding and using `FoldAdaptor`!
  - Implement replacements (e.g. double `list.reverse`)
    - Note that you are allowed to inspect IR structure in `fold`!
  - Note: `let hasFolder = 1;` is all you need in TableGen, constants and `materializeConstant` are already implemented for you.
- Stretch Goal: Do the same for `list.get_elements`, `list.peek_front`, `list.empty`, `list.pop_front` and `list.push_front`
- Attributes you'll need: `BoolAttr`, `IntegerAttr` and `ListAttr`. Use and look at their `get` methods if need be.
- Add tests! Use the canonicalize pass to exercise `fold`.

When to `fold` and when to call `RewritePatterns`

- `fold` when you can! (Rule of least power)
- Constant folding
  - `FoldAdaptor` is your friend
- Canonicalization only

- Use `RewritePattern`s when you need to create Operations etc.
  - `PatternRewriter` is your friend
- Can also be used for “lowering” or opinionated transformations

Many patterns add operations that are redundant

```
1  func.func @addr(%i: index, %j: index) -> (index, index) {
2      %c128   = arith.constant 128 : index
3      %0      = arith.muli %i, %c128 : index
4      %1      = arith.addi %0, %j : index
5      %c128_0 = arith.constant 128 : index
6      %2      = arith.muli %i, %c128_0 : index
7      %3      = arith.addi %2, %j : index
8      %c1     = arith.constant 1 : index
9      %4      = arith.addi %3, %c1 : index
10     return %1, %4 : index, index
11 }
```

-cse

```
1 func.func @addr(%i: index, %j: index) -> (index, index) {
2     %c128 = arith.constant 128 : index
3     %0 = arith.muli %i, %c128 : index
4     %1 = arith.addi %0, %j : index
5     %c1 = arith.constant 1 : index
6     %2 = arith.addi %1, %c1 : index
7     return %1, %2 : index, index
8 }
```

Common Subexpression Elimination (CSE): If an equivalent operation already dominates this one, reuse its result instead of recomputing it.

CSE only needs to know about side effects to work

```
1 // Pure = AlwaysSpeculatable + NoMemoryEffect
2 def My_AddOp : My_Op<"add", [Pure]> {
3   let arguments = (ins Index:$lhs, Index:$rhs);
4   let results   = (outs Index:$result);
5 }
```

The only thing you need to do to let your operation participate is to define your side-effects properly. The CSE pass does the rest.

This is a very nice example of a transformation that looks more abstract on the effect ops have instead of their specific semantics.

CSE General Algorithm idea

Walk the operations one and remember every expression already seen

```
1  def cse(ops):
2      seen = {}  # key -> op
3      for op in ops:
4          key = (op.name, op.operands,
5              op.attributes, op.result_types)
6          if key in seen:  # HIT
7              op.replace_all_uses_with(seen[key])
8              op.erase()
9          else:  # MISS
10             seen[key] = op
```

The key to lookup the operation consist of the name, operand, attributes and result_types. We are ignoring the value and the location.

Time O(n)
Memory O(n)
Only work for straight line code

CSE upstream implementation

```cpp
/// Attempt to eliminate a redundant operation.
LogicalResult CSEDriver::simplifyOperation(ScopedMapTy &knownValues,
                                           Operation *op,
                                           bool hasSSADominance) {
  // Don't simplify terminator operations.
  if (op->hasTrait<OpTrait::IsTerminator>())
    return failure();

  // Don't simplify operations with regions that have multiple blocks.
  // TODO: We need additional tests to verify that we handle such IR correctly.
  if (!llvm::all_of(op->getRegions(),
                    [](Region &r) { return r.empty() || r.hasOneBlock(); }))
    return failure();

  // Some simple use case of operation with memory side-effect are dealt with
  // here. Operations with no side-effect are done after.
  if (!isMemoryEffectFree(op)) {                                  // ← line 264
    // TODO: Only basic use case for operations with MemoryEffects::Read can be
    // eleminated now. [...]
    if (!hasSingleEffect<MemoryEffects::Read>(op))
      return failure();

    // Look for an existing definition for the operation.
    if (auto *existing = knownValues.lookup(op)) {
      if (existing->getBlock() == op->getBlock() &&
          !hasOtherSideEffectingOpInBetween(existing, op)) {
        replaceUsesAndDelete(knownValues, op, existing, hasSSADominance);
        return success();
      }
    }
    knownValues.insert(op, op);
    return failure();
  }

  // Look for an existing definition for the operation.
  if (auto *existing = knownValues.lookup(op)) {
    replaceUsesAndDelete(knownValues, op, existing, hasSSADominance);
    return success();
  }

  // Otherwise, we add this operation to the known values map.
  knownValues.insert(op, op);
  return failure();
}
```

CSE Side effect

```
func.func @effects(%arg0: memref<4xf32>, %arg1: index, %arg2: f32)
        -> (f32, memref<4xf32>, memref<4xf32>) {
    %0 = arith.addi %arg1, %arg1 : index
    %1 = arith.addi %arg1, %arg1 : index
    %alloc = memref.alloc() : memref<4xf32>
    %alloc_0 = memref.alloc() : memref<4xf32>
    %2 = memref.load %arg0[%0] : memref<4xf32>
    %3 = memref.load %arg0[%1] : memref<4xf32>
    memref.store %arg2, %arg0[%arg1] : memref<4xf32>
    %4 = memref.load %arg0[%0] : memref<4xf32>
    %5 = arith.addf %2, %3 : f32
    %6 = arith.addf %5, %4 : f32
    return %6, %alloc, %alloc_0 : f32, memref<4xf32>, memref<4xf32>
}
```

```
func.func @effects(%arg0: memref<4xf32>, %arg1: index, %arg2: f32)
        -> (f32, memref<4xf32>, memref<4xf32>) {
    %0 = arith.addi %arg1, %arg1 : index
    %alloc = memref.alloc() : memref<4xf32>
    %alloc_0 = memref.alloc() : memref<4xf32>
    %1 = memref.load %arg0[%0] : memref<4xf32>
    memref.store %arg2, %arg0[%arg1] : memref<4xf32>
    %2 = memref.load %arg0[%0] : memref<4xf32>
    %3 = arith.addf %1, %1 : f32
    %4 = arith.addf %3, %2 : f32
    return %4, %alloc, %alloc_0 : f32, memref<4xf32>, memref<4xf32>
}
```

Pure ops merge. The two memref.alloc are identical to the hash and still both stay, because allocating is an effect. Loads merge too, but only until something writes. The load after the memref.store keeps its own value.

CSE if-else block

```
func.func @scopes(%arg0: i1, %arg1: index)
        -> (index, index, index) {
    %c1 = arith.constant 1 : index
    %0 = arith.addi %arg1, %c1 : index
    %1 = scf.if %arg0 -> (index) {
      %c1_0 = arith.constant 1 : index
      %3 = arith.muli %arg1, %arg1 : index
      %4 = arith.addi %c1_0, %3 : index
      scf.yield %4 : index
    } else {
      %c1_0 = arith.constant 1 : index
      %3 = arith.muli %arg1, %arg1 : index
      %4 = arith.subi %3, %c1_0 : index
      scf.yield %4 : index
    }
    %2 = arith.muli %arg1, %arg1 : index
    return %1, %2, %0 : index, index, index
}
```

```
func.func @scopes(%arg0: i1, %arg1: index)
        -> (index, index, index) {
    %c1 = arith.constant 1 : index
    %0 = arith.addi %arg1, %c1 : index
    %1 = scf.if %arg0 -> (index) {
      %3 = arith.muli %arg1, %arg1 : index
      %4 = arith.addi %c1, %3 : index
      scf.yield %4 : index
    } else {
      %3 = arith.muli %arg1, %arg1 : index
      %4 = arith.subi %3, %c1 : index
      scf.yield %4 : index
    }
    %2 = arith.muli %arg1, %arg1 : index
    return %1, %2, %0 : index, index, index
}
```

Both branches look outward and find the same %c1, so both constants merge. The three arith.muli stay: branches cannot see each other, and nothing inside is visible after. CSE only reuses a definition that already dominates. Merging those would mean moving code.