MLIR Fundamentals: Pattern-based Transformations

Matthias Springer (NVIDIA)

ACM Europe School on MLIR 2026 – August 11, 2026

Recap: List Dialect

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

All operations have value semantics and have no memory effects. Lists cannot be modified in-place.

Quiz: Implement `list.from_elements` with Other Ops

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Implement `list.from_elements` with Other Ops

```
%r = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
```

```
%0 = list.empty : !list.list<i64>
%1 = list.push_back %0, %a : !list.list<i64>
%2 = list.push_back %1, %b : !list.list<i64>
%r = list.push_back %2, %c : !list.list<i64>
```

Quiz: Implement `list.from_elements` with Other Ops

```
%r = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
```

```
%0 = list.from_elements %a, %b : !list.list<i64>
%r = list.push_back %0, %c : !list.list<i64>
```

Quiz: Runtime Complexity of `list.push_back`

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Runtime Complexity of `list.push_back`

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.li
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Just an API. The complexity depends on the implementation / lowering. The name / API suggests a doubly-linked list, so probably O(1). But there's no way to tell for sure!

Quiz: Implement `list.get_elements` with Other Ops

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Implement `list.get_elements` with Other Ops

```
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
```

```
%d = list.peek_front %l : !list.list<i64> -> i64
%l2 = list.pop_front %l : !list.list<i64>
%e, %f = list.get_elements %l2 : (!list.list<i64>) -> (i64, i64)
```

Quiz: Implement `list.range` with Other Ops

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Implement `list.range` with Other Ops

```
%r = list.range %lb to %ub : !list.list<i32>
```

```
%l = list.empty : !list.list<i32>
%r = scf.for %iv = %lb to %ub step %c1 iter_args(%iter = %l) -> (!list.list<i32>) : i32 {
  %next = list.push_back %iter, %iv : !list.list<i32>
  scf.yield %next : !list.list<i32>
}
```

Quiz: Implement `list.range` with Other Ops

Induction Variable

Loop-Carried Variable

Types of Loop-Carried Variables

```
%r = scf.for %iv = %lb to %ub step %c1 iter_args(%iter = %l) -> (!list.list<i32>) : i32 {
  %next = list.push_back %iter, %iv : !list.list<i32>
  scf.yield %next : !list.list<i32>
}

// Conceptually, the same as this C++ code:
int32_t iv;
list_t iter = l;
for (iv = lb; iv < ub; iv += step) {
  iter = iter.push_back(iv);
}
```

Type of Induction Variable

Quiz: Implement `list.length` with Other Ops

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Implement `list.length` with Other Ops

```
%r = list.length %l : !list.list<i64> -> i32

// int count = 0;
// list rem = l;
// while (!is_empty(rem)) {
//   ++count;
//   rem = rem.pop_front()
// }
```

Quiz: Implement `list.length` with Other Ops

```
%r = list.length %l : !list.list<i64> -> i32

%c1 = arith.constant 1 : i32
%true = arith.constant true
%c0 = arith.constant 0 : i32
%tmp, %r = scf.while (%rem = %l, %cnt = %c0) : (!list.list<i32>, i32) -> (!list.list<i32>, i32) {
  %is_empty = list.is_empty %rem : !list.list<i32> -> i1
  %cond = arith.xori %is_empty, %true : i1
  scf.condition(%cond) %rem, %count : !list.list<i32>, i32
} do {
^bb0(%rem2: !list.list<i32>, %cnt2: i32):
  %rest = list.pop_front %rem2 : !list.list<i32>
  %next_cnt = arith.addi %cnt2, %c1 : i32
  scf.yield %rest, %next_cnt : !list.list<i32>, i32
}
```

Quiz: Implement `list.map` with Other Ops

```
%0 = list.empty : !list.list<i64>
%1 = list.is_empty %l : !list.list<i64> -> i1
%2 = list.length %l : !list.list<i64> -> i32
%3 = list.peek_front %l : !list.list<i64> -> i64
%4 = list.pop_front %l : !list.list<i64> -> !list.list<i64>
%5 = list.push_back %l, %a : !list.list<i64>
%6 = list.push_front %l, %a : !list.list<i64>
%7 = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
%d, %e, %f = list.get_elements %l : (!list.list<i64>) -> (i64, i64, i64)
%8 = list.range %lb to %ub : !list.list<i32>
%9 = list.reverse %l : !list.list<i64> -> !list.list<i64>
%10 = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Quiz: Implement `list.map` with Other Ops

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}

// list result = empty_list();
// int len = l.length();
// list rem = l;
// for (int i = 0; i < len; ++i) {
//   mapped = map_body(rem.peek_front());
//   rem = rem.pop_front()
//   result = result.push_back(mapped);
// }
```

Quiz: Implement `list.map` with Other Ops

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}

// list result = empty_list();
// list rem = l;
// while (!is_empty(rem)) {
//   mapped = map_body(rem.peek_front());
//   rem = rem.pop_front()
//   result = result.push_back(mapped);
// }
```

Quiz: Implement `list.map` with Other Ops

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

```
%empty = list.empty : !list.list<i64>
%true = arith.constant true
%tmp, %r = scf.while (%rem = %l, %acc = %empty) : (!list.list<i64>, !list.list<i64>) -> (!list.list<i64>, !list.list<i64>) {
  %is_empty = list.is_empty %rem : !list.list<i64> -> i1
  %cond = arith.xori %is_empty, %true : i1
  scf.condition(%cond) %rem, %acc : !list.list<i64>, !list.list<i64>
} do {
^bb0(%rem2: !list.list<i64>, %acc2: !list.list<i64>):
  %elt = list.peek_front %rem2 : !list.list<i64> -> i64
  %rest = list.pop_front %rem2 : !list.list<i64>
  %mapped = arith.addi %elt, %c1 : i64
  %next_acc = list.push_back %acc2, %mapped : !list.list<i64>
```

Quiz: Runtime Complexity of `list.map`?

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Based on the lowering from the previous slide.

Quiz: Runtime Complexity of `list.map`?

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Based on the lowering from the previous slide.

n times peek, n times pop, n times push, where n is the length of the list.

Quiz: Runtime Complexity of Double `list.map`?

```
%tmp = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}

%r = list.map %tmp with (%elem : i64) -> i64 {
  %g = arith.muli %elem, %c2 : i64
  list.yield %g : i64
}
```

Quiz: Runtime Complexity of Double `list.map`?

```
%tmp = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}

%r = list.map %tmp with (%elem : i64) -> i64 {
  %g = arith.muli %elem, %c2 : i64
  list.yield %g : i64
}
```

2n times peek, 2n times pop, 2n times push, where n is the length of the list.

Quiz: Any Way to Improve This?

```
%tmp = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}

%r = list.map %tmp with (%elem : i64) -> i64 {
  %g = arith.muli %elem, %c2 : i64
  list.yield %g : i64
}
```

2n times peek, 2n times pop, 2n times push, where n is the length of the list.

Quiz: Any Way to Improve This?

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  %g2 = arith.muli %g, %c2 : i64
  list.yield %g2 : i64
}
```

n times peek, n times pop, n times push, where n is the length of the list.

Coding Exercise 1: Lowering for `list.from_elements`

`Exercise_1_ListLowerFromElements.cpp`

```
%r = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
```

```
%0 = list.empty : !list.list<i64>
%1 = list.push_back %0, %a : !list.list<i64>
%2 = list.push_back %1, %b : !list.list<i64>
%r = list.push_back %2, %c : !list.list<i64>
```

Hint: Utilize `getOperation()->walk([&] (FromElementsOp op) { … } );` to find all `ToElementsOps`.
Location: Source location in .mlir file. Should be propagated when created a new op. (`op->getLoc()`)

~/mlir-summer-school-2026-hw$ build/tutorial/tutorial-opt test/list/Exercise_1_lower-from-elements.mlir -split-input-file -list-lower-from-elements

Rewrite Pattern Infrastructure

- Rewrite Pattern: A modular piece of C++ that matches IR and rewrites it.
- Match: Pattern-match certain IR.
- Rewrite: Modify IR.

Example: `list.from_elements` → `list.empty`

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

 LogicalResult matchAndRewrite(FromElementsOp op,
                               PatternRewriter &rewriter) const override {
   if (!op.getElements().empty())
     return rewriter.notifyMatchFailure(op, "list has elements");

   rewriter.replaceOpWithNewOp<EmptyOp>(op, op.getResult().getType());
   return success();
 }
};
```

Example: `list.from_elements` → `list.empty`

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

 LogicalResult matchAndRewrite(FromElementsOp op,
                               PatternRewriter &rewriter) const override {
   if (!op.getElements().empty())
     return rewriter.notifyMatchFailure(op, "list has elements");

   rewriter.replaceOpWithNewOp<EmptyOp>(op, op.getResult().getType());
   return success();
 }
};
```

EmptyOp::create + replaceAllUsesWith

Example: `list.from_elements` → `list.empty`

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

 LogicalResult matchAndRewrite(FromElementsOp op,
                               PatternRewriter &rewriter) const override {
   if (!op.getElements().empty())
     return rewriter.notifyMatchFailure(op, "list has elements");

   rewriter.replaceOpWithNe                                        );
   return success();
 }
};
```

You must not modify any IR if `matchAndRewrite` returns `failure`.

Example: `list.from_elements` → `list.empty`

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

 LogicalResult matchAndRewrite(FromElementsOp op,
                               PatternRewriter &rewriter) const override {
   if (!op.getElements().empty())
     return rewriter.notifyMatchFailure(op,

   rewriter.replaceOpWithNewOp<EmptyOp>(op,
   return success();
 }
};
```

`PatternRewriter` derived from `OpBuilder`. All IR modifications must go through the rewriter!

Example: Greedy Pattern Rewrite Driver

```
struct ListSimplifyPass : public impl::ListSimplifyBase<ListSimplifyPass> {

 void runOnOperation() override {
   RewritePatternSet patterns(&getContext());
   patterns.add<ReplaceEmptyFromElements>(patterns.getContext());

   if (failed(applyPatternsGreedily(getOperation(), std::move(patterns))))
     signalPassFailure();
 }
};
```

Example: Greedy Pattern Rewrite Driver

```
struct ListSimplifyPass : public impl::ListSimplifyBase<ListSimplifyPass> {

 void runOnOperation() override {
   RewritePatternSet patterns(&getContext());
   patterns.add<ReplaceEmptyFromElements>(patterns.getContext());

   if (failed(applyPatternsGreedily(getOperation(), std::move(patterns))))
     signalPassFailure();
 }
};
```

Applies all patterns to a fixed point.
(Until nothing changes anymore.)

Coding Exercise 2: Lowering of `list.from_elements`

```
%r = list.from_elements %a, %b, %c : (i64, i64, i64) -> !list.list<i64>
```

```
%0 = list.from_elements %a, %b : !list.list<i64>
%r = list.push_back %0, %c : !list.list<i64>
```

`Exercise_2_3_4_ListSimplify.cpp`

Coding Exercise 3: Merge two `list.map`

```
%tmp = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
%r = list.map %tmp with (%elem : i64) -> i64 {
  %g = arith.muli %elem, %c2 : i64
  list.yield %g : i64
}
```

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  %g2 = arith.muli %g, %c2 : i64
  list.yield %g2 : i64
}
```

Hint: `%tmp` may have users in addition to the second `list.map`.

Hint: Get familiar with `OpBuilder::clone` and the `IRMapping` API.

Coding Exercise 4: Lower `list.map`

```
%r = list.map %l with (%elem : i64) -> i64 {
  %g = arith.addi %elem, %c1 : i64
  list.yield %g : i64
}
```

Note: Make sure that the `list.map` merge pattern has a higher benefit!

Alternative implementation: Lower to `list.length` + `scf.for`.

```
%empty = list.empty : !list.list<i64>
%true = arith.constant true
%tmp, %r = scf.while (%rem = %l, %acc = %empty) : (!list.list<i64>, !list.list<i64>) -> (!list.list<i64>, !list.list<i64>) {
  %is_empty = list.is_empty %rem : !list.list<i64> -> i1
  %cond = arith.xori %is_empty, %true : i1
  scf.condition(%cond) %rem, %acc : !list.list<i64>, !list.list<i64>
} do {
^bb0(%rem2: !list.list<i64>, %acc2: !list.list<i64>):
  %elt = list.peek_front %rem2 : !list.list<i64> -> i64
  %rest = list.pop_front %rem2 : !list.list<i64>
  %mapped = arith.addi %elt, %c1 : i64
  %next_acc = list.push_back %acc2, %mapped : !list.list<i64>
```

Pattern-based Rewrite Drivers

- `walkAndApplyPatterns`: walk IR once, try to apply patterns to each op
- `applyPatternsGreedily`: apply patterns to ops until fixed point
- `applyPartialConversion` / `applyFullConversion`: dialect conversion
- Greedy pattern driver + dialect conversion can be called from transform dialect IR.

Best Practices and API Rules

- Prefer walk over more complex pattern drivers.
- Rewrite pattern: return “success” if an only if the IR was modified.
- Rewrite pattern: IR should verify after pattern application.
- All IR modification must go through the builder / rewriter (when available).
- Get used to `-debug`.

Resources

- Pattern Rewriting : Generic DAG-to-DAG Rewriting
- Dialect Conversion
- The State of Pattern-Based IR Rewriting in MLIR
- A Faster and Simpler Dialect Conversion Driver without Pattern Rollback
- 1:N Dialect Conversion in MLIR