Pattern-Based Rewrites in MLIR

Matthias Springer

Deepgreen MLIR Winter School – Jan 30, 2025

Outline

1. Overview of Traversal Mechanisms (API)
   a. IR Walk
   b. Pattern Walk Driver
   c. Greedy Pattern Rewriter
   d. Dialect Conversion
2. Best Practices for Greedy Pattern Rewrites and Dialect Conversion

Example Source Code:
https://github.com/llvm/llvm-project/commit/2cc29d9d14d06a791afdc5232a24dcfa369a76ef

IR Traversal Infrastructure in MLIR

IR Walk	Pattern Walk Driver	Greedy Rewrite	Transform Dialect	Dialect Conversion
	Pattern based	Pattern based		Pattern based
Visitor-based traversal of ops, regions or blocks.	Pattern-based rewrite of ops in a **single top-to-bottom traversal**.	**Fixed-point iteration** of pattern applications.	Matching IR via **handles** and rewriting IR via transform op application.	Pattern-based rewrite of **illegal ops** into legal ops in a single top-to-bottom traversal.

increasing complexity + runtime overhead

IR Walk

IR Walk

```
// Visitor-based traversal of topLevel.
// Dump topLevel and all nested ops.

Operation *topLevel;
WalkResult result = topLevel->walk([](Operation *op) {
  op->dump();
  return WalkResult::advance();  // optional
});
```

- `WalkResult::advance()`: Continue traversal.
- `WalkResult::skip()`: Use if op was erased. Continue traversal.
- `WalkResult::interrupt()`: Stop traversal.

IR Walk

```
// Visitor-based traversal of topLevel.
// Dump topLevel and all nested ops.

Operation *topLevel;
WalkResult result = topLevel->walk([](FuncOp funcOp) {
  funcOp->dump();
  return WalkResult::advance();  // optional
});
```

visit only functions ops

IR Walk

```
// Visitor-based traversal of topLevel.
// Dump topLevel and all nested ops.

Operation *topLevel;
WalkResult result = topLevel->walk([](FunctionOpInterface funcOp) {
  funcOp->dump();
  return WalkResult::advance();  // optional
});
```

visit only function-like ops

IR Walk

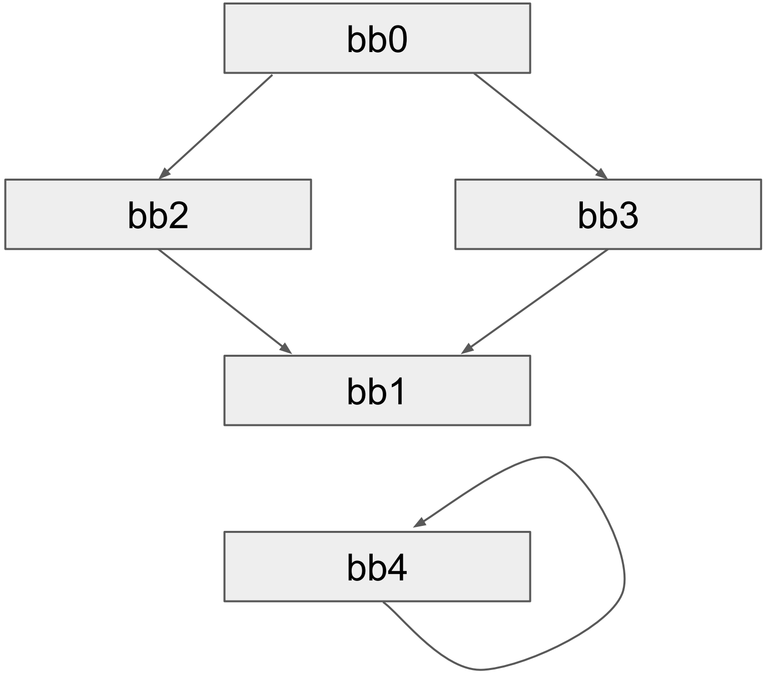
```
// Visitor-based traversal of topLevel.
// Dump topLevel and all nested ops.

Operation *topLevel;
topLevel->walk<Order, Iterator>([](Operation *op) {
  op->dump();
});
```

`ForwardIterator`: top-to-bottom
`ForwardDominanceIterator<>`: according to dominance (defs before uses)
`ReverseIterator`: bottom-to-top
`ReverseDominanceIterator<>`: according to reverse dominance

`WalkOrder::PostOrder`: visit an op after its nested ops
`WalkOrder::PreOrder`: visit an op before its nested ops

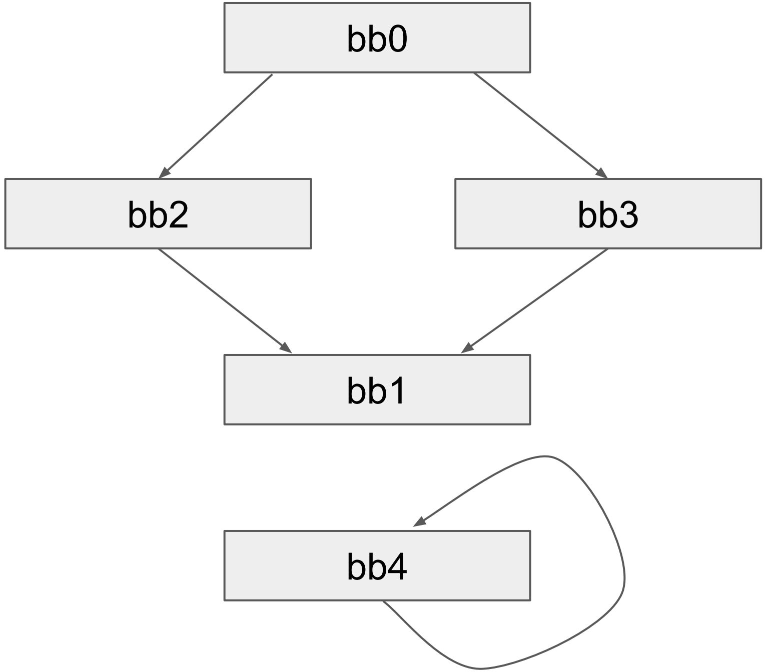
Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```



Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```



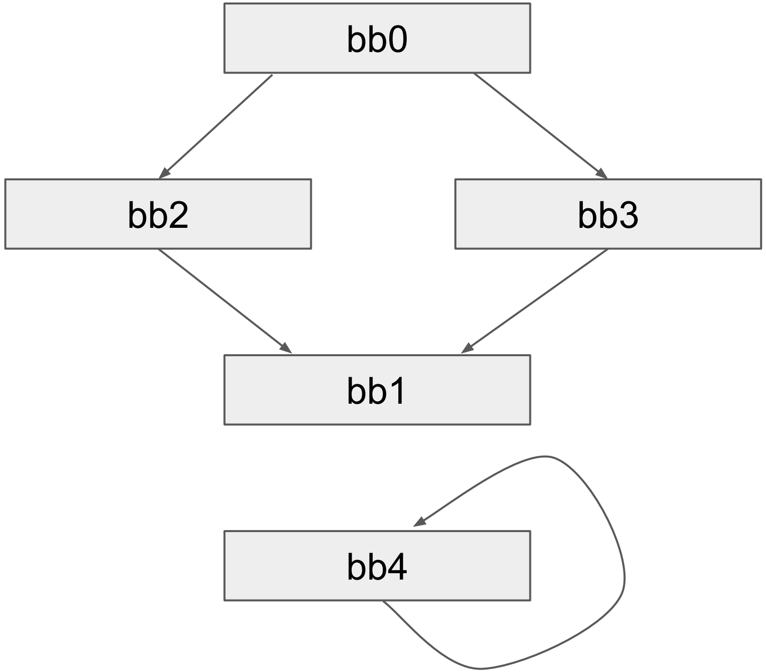
Forward, PreOrder:

bb0 | bb1 | bb2 | bb3 | bb4

func, op_1, op_2, cond_br, return, op_3, op_4, br, op_5, br, op_6, br

Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```

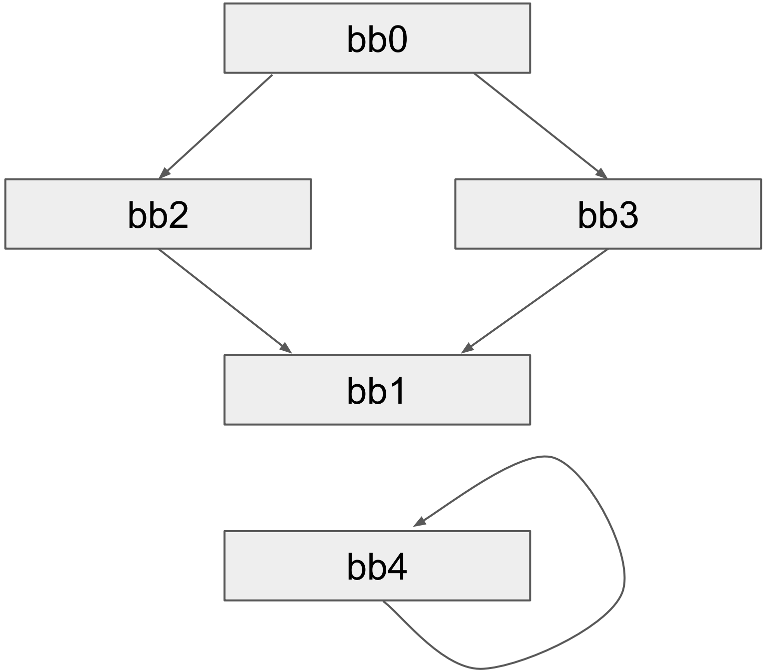


Forward, PostOrder:

bb0: op_1, op_2, cond_br, bb1: return, bb2: op_4, op_3, br, bb3: op_5, br, bb4: op_6, br, func

Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```



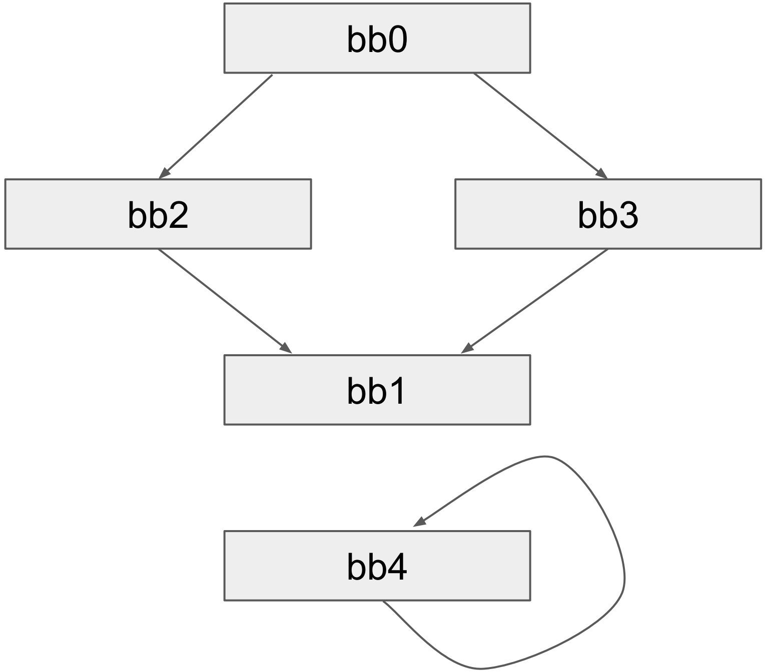
Reverse, PostOrder:

bb4 | bb3 | bb2 | bb1 | bb0

br, op_6, br, op_5, br, op_4, op_3, return, cond_br, op_2, op_1, func

Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```



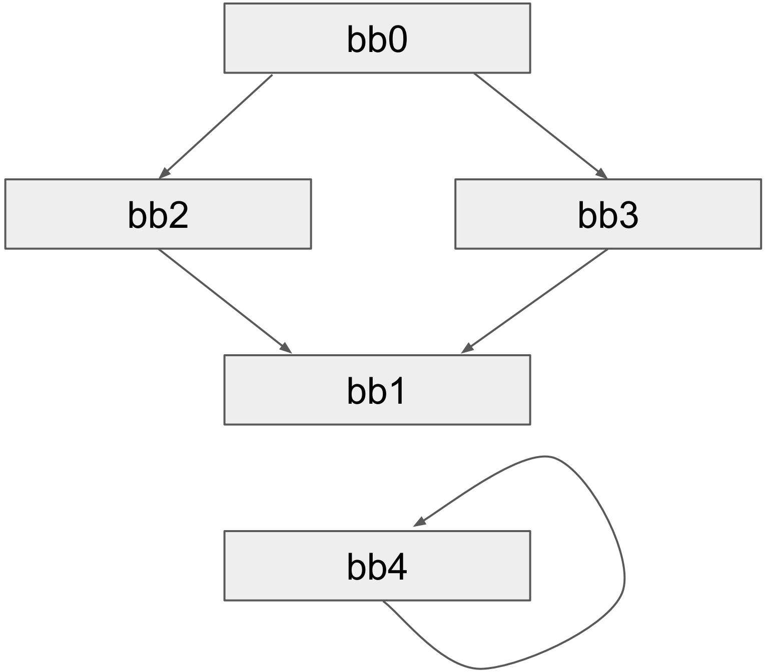
ForwardDominance, PostOrder:

bb0 | bb2 | bb1 | bb3

op_1, op_2, cond_br, op_4, op_3, br, return, op_5, br, func

Example: Order, Iterator

```
func.func @test_case() {
  "test.op_1"() : () -> ()
  %0 = "test.op_2"() : () -> (i1)
  cf.cond_br %0, ^bb2, ^bb3
^bb1:
  func.return
^bb2:
  "test.op_3"() ({
    "test.op_4"(%0) : (i1) -> ()
  }) : () -> ()
  cf.br ^bb1
^bb3:
  "test.op_5"(%0) : (i1) -> ()
  cf.br ^bb1
^bb4:
  "test.op_6"() : () -> ()
  cf.br ^bb4
}
```



ReverseDominance, PostOrder:

bb1 | bb2 | bb3 | bb0

return, br, op_4, op_3, br, op_5, cond_br, op_2, op_1, func

IR Walk

```
// Visitor-based traversal of topLevel.
// Dump all nested blocks.

Operation *topLevel;
topLevel->walk([](Block *block) {
  block->dump();
});
```

IR Walk

```
// Visitor-based traversal of topLevel.
// Dump all nested regions.

Operation *topLevel;
topLevel->walk([](Region *region) {
  region->dump();  // There is no Region::dump.
});
```

Patterns

What is a Pattern?

- `match`: C++ code that looks for certain IR.
- `rewrite`: C++ code that modifies IR.
- Typically combined into one function: `matchAndRewrite`

also: RewritePattern, OpInterfaceRewritePattern

Anatomy of a RewritePattern

when there are multiple patterns for an op:
try higher-benefit patterns first

```
class AddFoldPattern : public OpRewritePattern<arith::AddIOp> {

  AddFoldPattern(MLIRContext *ctx, PatternBenefit benefit = 1)
    : OpRewritePatterns<arith::AddIOp>(ctx, benefit) {}

  LogicalResult matchAndRewrite(AddIOp op, PatternRewriter &rewriter) const {
    std::optional<int64_t> lhs = getConstantIntValue(op.getLhs());
    std::optional<int64_t> rhs = getConstantIntValue(op.getRhs());
    if (!lhs || !rhs) return failure();
    rewriter.replaceOpWithNewOp<arith::ConstantOp>(
        op, rewriter.getIntegerAttr(op.getType(), *lhs + *rhs);
    return success();
  }

};
```

success or failure

Anatomy of a `PatternRewriter`

- General rule: Whenever you have a builder/rewriter, perform all IR changes through the builder/rewriter.
- *Example:* `op->erase()` ⇒ `rewriter.eraseOp(op)`
- Builder/rewriter is a thin wrapper around the MLIR IR API with an insertion point and an optional listener.

Why Patterns: Modularity

- Helps breaking down a pass into smaller **composable** pieces. (Pieces that can be understood, tested, developed individually.)
- Patterns can be **reused** in multiple passes. (And shared with other programmers. Special case: *canonicalization patterns*.)
- Patterns can be developed / debugged / understood in isolation.

Pattern Walk Driver

Pattern Walk Driver: `walkAndApplyPatterns`

```
RewritePatternSet patterns(ctx);
patterns.add<AddFoldPattern, SubFoldPattern, MulFoldPattern>(ctx);

// Post-order, forward walk traversal of ops (excluding `op`).
Operation *op;
walkAndApplyPatterns(op, std::move(patterns));
```

patterns may erase matched op and nested ops/blocks, but not other ops/blocks

Example: Manual IR walk instead of Pattern Walk

```
op->walk([](Operation *op) {
  if (auto addOp = dyn_cast<AddIOp>(op)) {
    // Try to rewrite arith.addi.
  } else if (auto subOp = dyn_cast<SubIOp>(op)) {
     // Try to rewrite arith.subi.
  } else if (auto mulOp = dyn_cast<MulIOp>(op)) {
    // Try to rewrite arith.muli.
  } else if ...
});
```

Greedy Pattern Driver

Greedy Pattern Driver: `applyPatternsAndFoldGreedily`

- Apply to all ops in a given scope until a fixed point is reached.
  - Pattern application
  - Fold operations: fold op in-place, or: fold to attribute / SSA Value
  - Simplify regions:
    - region DCE (ops + block args)
    - erase unreachable blocks
    - merge identical blocks
  - CSE constants
  - Remove dead operations (DCE)
- Worklist-based implementation:
  Put op back onto the worklist when something has changed in its vicinity.
- Ops may be visited multiple times. (Hard to predict the cost of the driver.)
- No guaranteed order of traversal.

Greedy Pattern Driver: `applyPatternsAndFoldGreedily`

- Apply to all ops in a given scope until a fixed point is reached.
  - Pattern application
  - Fold operations: fold op in-place, or: fold to attribute / SSA Value
  - Simplify regions:
    - region DCE
    - erase unre
    - merge ider
  - CSE constants
  - Remove dead op
- Worklist-based im
  Put op back onto ... ged in its vicinity.
- Ops may be visite ... ost of the driver.)
- No guaranteed or

```cpp
OpFoldResult arith::AddIOp::fold(FoldAdaptor adaptor) {
  // addi(x, 0) -> x
  if (matchPattern(adaptor.getRhs(), m_Zero()))
    return getLhs();

  // addi(subi(a, b), b) -> a
  if (auto sub = getLhs().getDefiningOp<SubIOp>())
    if (getRhs() == sub.getRhs())
      return sub.getLhs();

  // addi(b, subi(a, b)) -> a
  if (auto sub = getRhs().getDefiningOp<SubIOp>())
    if (getLhs() == sub.getRhs())
      return sub.getLhs();

  return constFoldBinaryOp<IntegerAttr>(
      adaptor.getOperands(),
      [](APInt a, const APInt &b) { return std::move(a) + b; });
}
```

Greedy Pattern Driver: applyPatternsAndFoldGreedily

```
RewritePatternSet patterns;
patterns.insert<MyPattern>(ctx);
GreedyRewriteConfig config;
LogicalResult status =
    applyAndFoldGreedily(op, std::move(patterns), config);
```

failure if the max. #iterations was exceeded without reaching a fixed point

Greedy Pattern Driver Configuration

```
class GreedyRewriteConfig {
public:
  // Applies only to the worklist initialization. Cannot enforce a rewrite/traversal order.
  bool useTopDownTraversal = false;

  // Disabled, Normal, Aggressive
  GreedySimplifyRegionLevel enableRegionSimplification = GreedySimplifyRegionLevel::Aggressive;

  // Can be used to abort the rewrite process if it takes too long.
  int64_t maxIterations = 10;
  int64_t maxNumRewrites = kNoLimit;
  static constexpr int64_t kNoLimit = -1;

  Region *scope = nullptr;

  // AnyOp, ExistingAndNewOps, ExistingOps
  GreedyRewriteStrictness strictMode = GreedyRewriteStrictness::AnyOp;

  RewriterBase::Listener *listener = nullptr;
};
```

DEMO:
`test-arith-reduce-float-bitwidth`

https://github.com/llvm/llvm-project/commit/2cc29d9d14d06a791afdc5232a24dcfa369a76ef

Canonicalization

- Special class of patterns that *simplify* IR or bring IR into a *canonical* form.
- Registered together with the op definition:
  `OpName::getCanonicalizationPatterns`

Example: Propagating static type information

```
%sz = arith.constant 5 : index
%0 = tensor.extract_slice %t[0][%sz][1] : tensor<10xf32> to tensor<?xf32>
```

⇒

```
%0 = tensor.extract_slice %t[0][5][1] : tensor<10xf32> to tensor<5xf32>
```

Do Not Rely on Canonicalizer Pass for Correctness

- *Problem 1:* Default max. #iterations is set to 10.
  - Rewrite process may finish **without reaching a fixed point**. The resulting IR is **not guaranteed to be in a canonical form**.
  - (Max. #iterations can be configured.)
- *Problem 2:* Canonicalizer pass performs a greedy pattern rewrite with all registered canonicalization patterns.
  - Populate **only required patterns** in a custom greedy pattern rewrite to **improve efficiency**.
  - New canonicalization patterns may be added by third parties and/or other dialects, potentially making the **compilation pipeline more fragile**.
  - What should be canonicalization and what not is actively being discussed.

Rewrite Pattern: Return `success` iff IR was Modified

- At least one `success`: Run another greedy pattern iteration.
- Only `failure`s: No further greedy pattern iteration.

- *Case 1:* Pattern returned `success` but did not modify the IR.
  - Pattern triggers another iteration and will match again.
  - **Infinite loop!**
- *Case 2:* Pattern returned `failure` but modifies the IR.
  - Another (or this) pattern may match if given the chance.
  - *Case 2.1:* Pattern returned `failure` half-way through `matchAndRewrite`. The next pattern will see the result of an **incomplete pattern application**.
  - *Case 2.2:* Programmer's intention was to return `success`. But this may be last iteration and the process finished **without reaching a fixed point**.

Rewrite Pattern: IR Should Verify after Pattern Application

- *Public Rewrite Pattern:* Pattern that is exposed to users via `populate…Patterns(RewritePatternSet &)` function.
  - Pattern may run together with **other patterns** in a large greedy pattern rewrite.
  - It is difficult to develop **composable patterns** if there is **no contract**.
  - If the IR at the beginning of a rewrite pattern is invalid, a pattern may crash or misbehave.
- By default, the greedy pattern rewrite process **may stop suddenly** when the max. #iterations is exhausted.
  - Ideally, IR at the end of a greedy pattern rewrite should verify. (Because that's often also the end of a pass.)
- Not a strict rule. MLIR requires valid IR only between pass boundaries.

Rewrite Pattern: Expensive Pattern Checks

- Compile MLIR with `MLIR_ENABLE_EXPENSIVE_PATTERN_API_CHECKS`.
- Enables additional “expensive checks” in greedy pattern rewrite driver:
  - Detects most cases where IR was modified but pattern returned `failure` (or vice versa). Implemented via operation fingerprint (hashing all operations).
  - Detects most cases where IR was modified without the rewriter. (Via operation fingerprint.)
  - Detects cases where IR does not verify after pattern application. (Expected to fail for some patterns. E.g., patterns that modify `FuncOp` and `CallOp` separately.)
- Should be used together with `LLVM_USE_SANITIZER="Address"`.
  - Fingerprint verification crashes if ops are erased without the rewriter (dangling pointers) and ASAN will provide useful information to debug.

Rewrite Pattern: Randomize Operation Ordering

- Greedy pattern driver does not guarantee any op traversal order.
  - `GreedyRewriteConfig::useTopDownTraversal` controls the initial worklist population order.
  - `PatternBenefit` controls pattern priority once an operation was selected.
- Additional patterns / changes to existing patterns can affect the traversal op order.
- Op traversal order can affect the output IR. Ideally, any traversal order should produce *equivalent* IR. Ideally, `FileCheck` tests should still pass.
- Set `MLIR_GREEDY_REWRITE_RANDOMIZER_SEED` to randomize the worklist. (Operation is picked from worklist at random.)

All IR Modifications Must Use Rewriter

Incorrect: Bypassing the Rewriter

```
op->erase();
value.replaceAllUsesWith(value2);
op->setAttr("name", attr);
op->moveBefore(op2);
op->clone();
…
```

Correct: Using the Rewriter

```
rewriter.eraseOp(op);
rewriter.replaceAllUsesWith(value, value2);
rewriter.modifyOpInPlace([&]() {op->setAttr(...)});
rewriter.moveOpBefore(op, op2);
builder.clone(*op);
…
```

- Greedy pattern driver listens to notifications to populate the worklist.
- Dialect conversion driver intercepts + delays certain API calls.
- Missing in-place modifications / IR creation:
  Rewrite process may finish **without reaching a fixed point**.
- Missing erasure: Driver **may crash** due to dangling pointers on the worklist.

Prefer Walk over Pattern Driver

Use greedy pattern rewrite if:

- Fixed-point pattern application is required.
  E.g.: A rewrite step creates an operation that must also be rewritten.
- The set of rewrite steps and/or operations is open-ended.

Use dialect conversion if:

- Many rewrite steps involve type conversions.
  E.g.: A value is replaced with a value of a different type.

Otherwise: Use an `Operation::walk` or a pattern walk: It's faster, simpler and more predictable!

Dialect Conversion

Dialect Conversion

3 categories: legal, illegal, unspecified

- A pattern driver that rewrites *not legal* ops as per `ConversionTarget`.
  - `target.addLegalOp<ModuleOp>();`
  - `target.addIllegalOp<TestFooOp>();`
  - ```
    target.addDynamicallyLegalOp<arith::AddIOp>([](arith::AddIOp op) {
      return !isa<Float32Type>(op.getResult());
    });
    ```
- *Partial conversion:* Attempt to rewrite all not legal operations. Fails if an explicitly illegal op survives the conversion (or was created).
- *Full conversion:* Attempt to rewrite all not legal operations. Fails if such an op survives the conversion (or was created).
    ```

Anatomy of a `ConversionPattern`

optional

```
class AddFOpConversion : public OpConversionPattern<arith::AddFOp> {

  AddFOpConversion(const TypeConverter &converter, MLIRContext *ctx,
                   PatternBenefit benefit = 1)
    : OpConversionPattern<arith::AddFOp>(converter, ctx, benefit) {}

  LogicalResult matchAndRewrite(AddFOp op, OpAdaptor adaptor,
                              ConversionPatternRewriter &rewriter) const {
    rewriter.replaceOpWithNewOp<arith::AddFOp>(
        op, adaptor.getLhs(), adaptor.getRhs());
    return success();
  }

};
```

`adaptor` (auto-generated C++ class) gives access to operand replacements

- *no type converter:* most recently mapped value
- *has type converter:* most recently mapped value, "casted" to converted type

TypeConverter: Type Conversion Rules

```
converter.addConversion([](Type t) { return t; });

converter.addConversion(...);

converter.addConversion(...);


converter.addConversion([](Float32Type t) {
  return Float16Type::get(t.getContext());
});
```

applied bottom-to-top

`ConversionPatternRewriter`

- A `PatternRewriter` with extra functionality.
- Supports replacements with different types:
  `rewriter.replaceOp(Operation *, ValueRange)`

  `op->getResultTypes()` does not have to match value types

- Can convert the signature of a basic block: `applySignatureConversion`
- Does not support the full PatternRewriter API.
  E.g., `replaceAllUsesWith` is not supported.

Example: Lowering via Dialect Conversion (1)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {

  %r1 = arith.addf %0, %1 : f32



  %r2 = arith.addf %r1, %1 : f32

  func.return %r2 : f32
}
```

Example: Lowering via Dialect Conversion (2)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {

  %r1 = arith.addf %0, %1 : f32        AddFOpConversion matchAndRewrite



  %r2 = arith.addf %r1, %1 : f32


  func.return %r2 : f32
}
```

Example: Lowering via Dialect Conversion (3)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32


  %r2 = arith.addf %t2, %1 : f32


  func.return %r2 : f32
}
```

Example: Lowering via Dialect Conversion (4)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32


  %r2 = arith.addf %t2, %1 : f32


  func.return %r2 : f32
}
```

AddFOpConversion matchAndRewrite

Example: Lowering via Dialect Conversion (5)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %t3, %t4 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

Example: Lowering via Dialect Conversion (6)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %t3, %t4 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

CSE

Example: Lowering via Dialect Conversion (7)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %t3, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

Example: Lowering via Dialect Conversion (8)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %t3, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

fold

Example: Lowering via Dialect Conversion (9)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %r1_new, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

Example: Lowering via Dialect Conversion (10)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %r1_new, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

DCE

Example: Lowering via Dialect Conversion (11)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16
  %t1 = unrealized_conversion_cast %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %r1_new, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32
  func.return %t5 : f32
}
```

`TypeConverter`: Materializations

Instead of `unrealized_conversion_cast`, insert a different op.

```
converter.addTargetMaterialization([](OpBuilder &b, Float16Type resultType,
                                      ValueRange inputs, Location loc) -> Value {
  if (!isa<Float32Type>(inputs[0])) return Value();
  return b.create<arith::TruncIOp>(loc, resultType, inputs[0]);
});

converter.addSourceMaterialization([](OpBuilder &b, Float32Type resultType,
                                      ValueRange inputs, Location loc) -> Value {
  if (!isa<Float16Type>(inputs[0])) return Value();
  return b.create<arith::ExtIOp>(loc, resultType, inputs[0]);
});
```

Materialization Callbacks

replacement value or replacement value of the replacement value, etc.

- Automatically inserted to reconcile type mismatches.
- **Target materialization:** Pattern expects a value of type `T` for an operand `V`, but the most recently mapped value (if any) has a different type. Driver inserts a target materialization to `T`.
- **Source materialization:** Value `V` of type `S` was replaced with a value of different type `T`, but an original use of `V` (expecting type `S`) survives the conversion. Driver inserts a source materialization to `S`.

Example: Lowering via Dialect Conversion (12)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = unrealized_conversion_cast %0 : f32 to f16          target materialization
  %t1 = unrealized_conversion_cast %1 : f32 to f16          target materialization
  ~~%r1 = arith.addf %0, %1 : f32~~
  %r1_new = arith.addf %t0, %t1 : f16
  ~~%t2 = unrealized_conversion_cast %r1_new : f16 to f32~~
  ~~%t3 = unrealized_conversion_cast %t2 : f32 to f16~~
  ~~%t4 = unrealized_conversion_cast %1 : f32 to f16~~
  ~~%r2 = arith.addf %t2, %1 : f32~~
  %r2_new = arith.addf %r1_new, %t1 : f16
  %t5 = unrealized_conversion_cast %r2_new : f16 to f32     source materialization
  func.return %t5 : f32
}
```

Example: Lowering via Dialect Conversion (13)

```
func.func @foo(%arg0: f32, %arg1: f32) -> f32 {
  %t0 = arith.truncf %0 : f32 to f16
  %t1 = arith.truncf %1 : f32 to f16
  %r1 = arith.addf %0, %1 : f32
  %r1_new = arith.addf %t0, %t1 : f16
  %t2 = unrealized_conversion_cast %r1_new : f16 to f32
  %t3 = unrealized_conversion_cast %t2 : f32 to f16
  %t4 = unrealized_conversion_cast %1 : f32 to f16
  %r2 = arith.addf %t2, %1 : f32
  %r2_new = arith.addf %r1_new, %t1 : f16
  %t5 = arith.extf %r2_new : f16 to f32
  func.return %t5 : f32
}
```

Actual Implementation

- Not all `unrealized_conversion_cast` ops are immediately materialized. (Some are created only on demand.)
- The driver maintains state in internal data structures. (You won’t see everything in IR.)
- Op replacement and op erasure is materialized at the very end of the conversion. (You will see a mixture of old / new IR during the conversion.)
- The driver can rollback (undo) pattern applications. (To be removed soon.)

DEMO: `test-arith-reduce-float-bitwidth-conversion`

https://github.com/llvm/llvm-project/commit/2cc29d9d14d06a791afdc5232a24dcfa369a76ef

Background: MemRef → LLVM Type Conversion

```
memref<?x?xf32, strided<[?, ?], offset: ?>>
⇒
!llvm.ptr, !llvm.ptr, i64, i64, i64, i64, i64
```

(allocated ptr, aligned ptr, <rank> sizes, <rank> strides, offset)

```
memref<*xf32>
⇒
i64, !llvm.ptr
```

(rank, ptr to descriptor)

Anatomy of a 1:N `ConversionPattern`

```
class AllocOpConversion : public OpConversionPattern<memref::AllocOp> {

  AllocOpConversion(const TypeConverter &converter, MLIRContext *ctx,
                    PatternBenefit benefit = 1)
    : OpConversionPattern<memref::AllocOp>(converter, ctx, benefit) {}

  LogicalResult matchAndRewrite(memref::AllocOp op, OpAdaptor adaptor,
                                ConversionPatternRewriter &rewriter) const {
    // ...
    // allocated_ptr, aligned_ptr, offset, <rank> sizes, <rank> strides
    SmallVector<Value> descriptor;
    rewriter.replaceOpWithMultiple(op, {descriptor});
    return success();
  }
};
```

Anatomy of a 1:N `ConversionPattern`

```
class RankOpConversion : public OpConversionPattern<memref::RankOp> {

  RankOpConversion(const TypeConverter &converter, MLIRContext *ctx,
                   PatternBenefit benefit = 1)
    : OpConversionPattern<memref::RankOp>(converter, ctx, benefit) {}

  LogicalResult matchAndRewrite(memref::RankOp op, OneToNOpAdaptor adaptor,
                                ConversionPatternRewriter &rewriter) const {
    if (!isa<MemRefType>(op.getMemref().getType()) return failure();
    rewriter.replaceOp(op, adaptor.getMemref()[0]);
    return success();
  }

};
```

ValueRange

Anatomy of a 1:N ConversionPattern

```
class RankOpConversion : public OpConversionPattern<memref::RankOp> {

  RankOpConversion(const TypeConverter &converter, MLIRContext *ctx,
                   PatternBenefit benefit = 1)
    : OpConversionPattern<memref::RankOp>(converter, ctx, benefit) {}

  LogicalResult matchAndRewrite(memref::RankOp op, OneToNOpAdaptor adaptor,
                                ConversionPatternRewriter &rewriter) const {
    SmallVector<Value> oneToOneOperands =
        getOneToOneAdaptorOperands(adaptor.getOperands());
    return matchAndRewrite(op, OpAdaptor(oneToOneOperands, adaptor),
                           rewriter);
  }

};
```

by default: call 1:1 implementation for compatibility

Example: `replaceOpWithMultiple`

```
ConversionPatternRewriter::replaceOpWithMultiple(
    Operation *, ArrayRef<ValueRange>);
```

Example: Replace “`test.foo`” with “`test.bar`”.

```
%r = "test.foo"() : () -> (i1)
%1:2 = "test.bar"() : () -> (i2, i2)

"test.qux"(%r) : (i1) -> ()
```

➡

```
%1:2 = "test.bar"() : () -> (i2, i2)
%r = unrealized_conversion_cast
     %1#0, %1#1 : i2, i2 to i1
"test.qux"(%r) : (i1) -> ()
```

Conversion Pattern: Return `success` if successful

- `success`: The matched op must have been erased or modified in such a way that it is legal (according to `ConversionTarget`).
- `failure`: All pattern modifications are rolled back (and another pattern runs).
  - Rollback is going to be removed with the new One-Shot Dialect Conversion driver. (Talk to me if you think that you need this feature or leave a comment on the public RFC.)
  - Same requirements as for rewrite patterns are going to apply for `failure`.

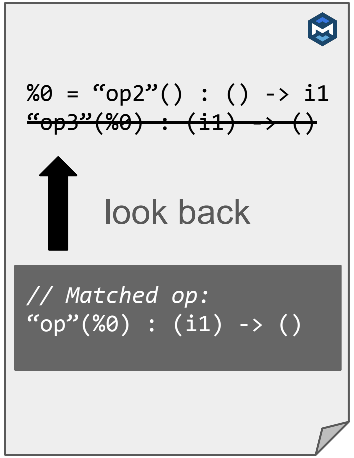
Conversion Pattern: Do Not Traverse IR

- Some IR changes (e.g., op erasure, updating uses) are **materialized in a delayed fashion** in a dialect conversion.
- Pattern implementations may see **outdated IR** ([related discussion]).

Example: Look back

```
LogicalResult matchAndRewrite(ConversionPatternRewriter r, Op op,
                              Adaptor adaptor) {
  // Check if `op` is the only user of the result of `op2`.
  auto op2 = op.getSource().getDefiningOp<Op2>();
  if (!op2) return failure();
  if (op2->getUsers().size() != 1) return failure();
  // ...
```

may include users that were already marked for erasure



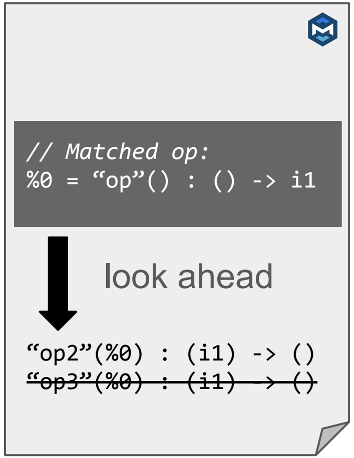
Conversion Pattern: Do Not Traverse IR

- Some IR changes (e.g., op erasure, updating uses) are **materialized in a delayed fashion** in a dialect conversion.
- Pattern implementations may see **outdated IR** (related discussion).

Example: Look ahead

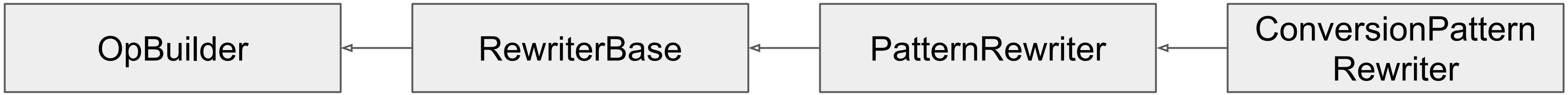
```
LogicalResult matchAndRewrite(ConversionPatternRewriter r, Op op,
                              Adaptor adaptor) {
  // Check if `op2` is the only user of the result of `op`.
  if (op.getResult()->getUsers().size() != 1) return failure();
  auto op2 = dyn_cast<Op2>(op.getResult()->getUsers().front());
  if (!op2) return failure();
  // ...
```

may include users that were already marked for erasure



Beware of Unsupported API

- `OpBuilder::setListener` / `getListener`
  - Dialect conversion framework and greedy pattern rewrite driver attach their own listeners.
  - Use `ConversionConfig::listener` / `GreedyRewriteConfig::listener`.
- Dialect conversion does not support `RewriterBase::replaceAllUsesWith`
  - Internal dialect conversion data structures operate on a per operation/block basis.
  - Replace operation: `RewriterBase::replaceOp`
  - Update block signature: `ConversionPatternRewriter::applySignatureConversion`



Rewrite Pattern: Do Not Use in Dialect Conversion

- API design suggests that `Conversion/RewritePattern` are compatible.
- But `ConversionPattern` API is **more restrictive** than `RewritePattern` API.
  - `PatternRewriter` exposes unsupported API, e.g.: `replaceAllUsesWith`.
  - Traversing IR is generally unsafe. You may see outdated IR or IR that was scheduled for erasure. (E.g.: value replacements are not visible yet, `getUses()` contains old uses, block still contains erased operations.)
  - Public `RewritePattern` can reasonably assume valid input IR, whereas IR is generally invalid after `ConversionPattern` application.
  - When creating new IR, operands of matched op should be accessed through the adaptor, but rewrite patterns do not have an adaptor.

`RewritePatternSet::add(std::unique_ptr<RewritePattern>)` + template overload

Conversion Pattern: Do Not Use in Greedy Rewrite

- API design suggests that Conversion/RewritePattern are compatible.
- Pattern implementation **will crash** when running in a greedy pattern rewrite. (Attempting to upcast PatternRewriter to ConversionPatternRewriter.)

RewritePatternSet::add(std::unique_ptr<RewritePattern>) + template overload

Dialect Conversion: Debugging Materialization Errors

```
error: failed to legalize unresolved materialization from () to 'i32' that remained live after conversion
  %0 = "test.illegal_op_a"() : () -> i32

note: see existing live user here: func.return %0 : i32
  return %0 : i32
```

- *Explanation:* A value was erased or replaced with a value of different type, but there are uses that were not updated.
- Set `ConversionConfig::buildMaterialization=false` and check output.

```
// mlir-opt test-legalize-erased-op-with-uses.mlir -test-legalize-unknown-root-patterns

func.func @remove_all_ops(%arg0: i32) -> i32 {
  %0 = builtin.unrealized_conversion_cast to i32
  return %0 : i32
}
```

op was erased but result still in use

not just for debugging…

Debugging with `-debug`

- Prints IR after each pattern application (and the name of the pattern).
- In case of dialect conversion: includes erased ops, replacements of values are not reflected yet.

matched op — pattern name

```
  * Pattern : 'func.func -> ()' {
Trying to match "(anonymous
namespace)::AnyFunctionOpInterfaceSignatureConversion"
    ** Insert Block into : 'func.func'(0x50c0000052c0)
    ** Insert  : 'cf.br'(0x50b0000d0ac0)
    ** Insert Block into : 'func.func'(0x50c0000052c0)
    ** Insert  : 'test.invalid'(0x507000016a60)
    ** Insert Block into : 'func.func'(0x50c0000052c0)
    ** Insert  : 'cf.br'(0x50b0000d0b70)
"(anonymous
namespace)::AnyFunctionOpInterfaceSignatureConversion"
result 1
```

erased IR

```
// *** IR Dump After Pattern Application ***
type mismatch for bb argument #0 of successor #0
mlir-asm-printer: 'builtin.module' failed to verify and will be
printed in generic form
"builtin.module"() ({
  "func.func"() <{function_type = () -> (), sym_name =
"test_undo_block_erase"}> ({
    "test.region"() ({
    }) {legalizer.erase_old_blocks, legalizer.should_clone} : () ->
    "test.return"() : () -> ()
  ^bb1(%0: f64):  // no predecessors
    %1 = "builtin.unrealized_conversion_cast"(%0) : (f64) -> i64
    %2 = "builtin.unrealized_conversion_cast"(%1) : (i64) -> f64
    "cf.br"(<<UNKNOWN SSA VALUE>>)[^bb3] : (i64) -> ()
  ^bb2(%3: f64):  // pred: ^bb3
    %4 = "builtin.unrealized_conversion_cast"(%3) : (f64) -> i64
    %5 = "builtin.unrealized_conversion_cast"(%4) : (i64) -> f64
```

bbarg from erased block

Dialect Conversion: Use Function + Control Flow Patterns

- `populateFunctionOpInterfaceTypeConversionPattern`:
  Generic pattern that converts the signature of any `FunctionOpInterface`.
- `populateSCFStructuralTypeConversions`:
  Generic patterns that convert SCF dialect ops.
- Customizable with a type converter.

Getting Started with the Dialect Conversion Infrastructure

- Type converters are optional.
- Argument/source/target materializations are optional.
- `applySignatureConversion` is optional in most cases. You can do almost everything with `inlineBlockBefore` and `replaceUsesOfBlockArgument`.
- `ConversionTarget` is mandatory.

Comparison of Pattern Drivers

Greedy Pattern Rewrite Driver

- `applyPatternsAndFoldGreedily()`
- `RewritePattern` + `PatternRewriter`

- Apply patterns to all ops.
- Also tries to fold + erase dead ops.
- No guaranteed IR traversal order.
- Process new, modified, … ops until a fixed point/cutoff is reached (via worklist).
- No rollback mechanism.
- No special handling for type changes.

Dialect Conversion

- `applyFull/PartialConversion()`
- `ConversionPattern` + `ConversionPatternRewriter`

- Apply patterns only to illegal ops.
- Also tries to fold selected ops (unsafe).
- Traverse by dominance ("top-to-bottom").
- Process new illegal ops (via recursion). Modified ops must be legal.
- Rolls back patterns on failure.
- Automatic type conversion (e.g., `replaceOp`) / materialization utilities.

Future Plans: One-Shot Dialect Conversion (RFC)

- Faster + more efficient: **No rollback** → no extra housekeeping
  - No more `ConversionValueMapping` (a kind of `IRMapping`)
  - No more stack of all IR changes
- Easier to understand/debug: **Immediately materialize all IR changes.**
  - You will always see the most recent IR.
  - Patterns can traverse the IR freely, etc.
- Compatible with `RewritePatterns`
- Support full `RewriterBase` / `PatternRewriter` API surface

```
applyPatternsAndFoldGreedily(moduleOp, /*empty*/frozenPatterns);      167ns/op
applyPartialConversion(moduleOp.get(), target, /*full*/patterns)      5398ns/op
```

M. Amini, J. Niu. How Slow is MLIR? EuroLLVM 2024 Keynote

Questions?

IR Walk
Pattern Walk
Greedy Pattern Rewrite Driver
1:1 Dialect Conversion
1:N Dialect Conversion
One-Shot Dialect Conversion
Listener Support
Fixed-point Iteration
Argument Materialization
Source Materialization
Target Materialization
Worklist Fuzzing / Randomization
Expensive Pattern Checks
Canonicalizer Pass

`RewritePattern`
`ConversionPattern`
`RewriterBase`
`PatternRewriter`
`ConversionPatternRewriter`
`matchAndRewrite`
`success` / `failure`
`buildMaterializations`
`replaceOpWithMultiple`
`OneToNOpAdaptor`
`walk`
`walkAndApplyPatterns`
`applySignatureConversion`