The State of Pattern-Based IR Rewriting in MLIR

Matthias Springer
NVIDIA Switzerland

2024 LLVM Developers' Meeting – October 23, 2024

IR Traversal Infrastructure in MLIR

IR Walk	Greedy Rewrite	Transform Dialect	Dialect Conversion
	Pattern based		Pattern based
Visitor-based traversal of ops, regions or blocks.	**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.

Overview 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.

Greedy Pattern Rewrite Driver: What's New?

- **Listen to IR modifications** by attaching a `RewriteListener`.
- Integration into the **transform dialect**: `transform.apply_patterns`
- **Expensive Pattern Checks:** new debugging facilities for invalid API usage.
- Additional flags to control region simplification.
- All entry points take a `GreedyRewriteConfig` object.

Dialect Conversion: What's New?

- **Listen to *most* IR modifications** by attaching a `RewriteListener`. (Triggered when the conversion succeeded.)
- Integration into the **transform dialect**: `transform.apply_conversion_patterns`
- Source/target/argument **materializations are optional**.
- New supported API: `moveOpBefore` / `moveOpAfter`
- Many internal bug fixes and additional assertions. Mostly related to block signature conversions and rollbacks.

Best Practices

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`: It's faster, simpler and more predictable!

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**.

Conversion Pattern: Return `success` if successful

- `success`: The matched must have been erased or modified in such a way that it is not 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`.

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.

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.

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.

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: 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, the 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.)

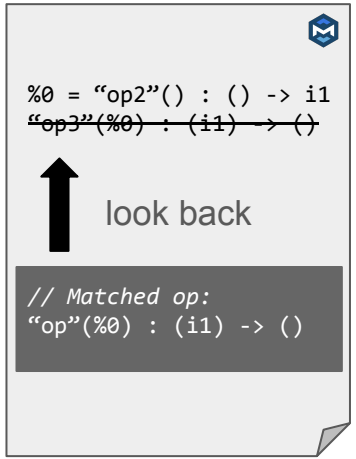
Conversion Pattern: Do Not Traverse IR

- 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



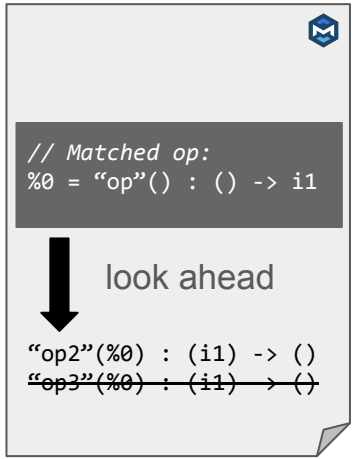
Conversion Pattern: Do Not Traverse IR

- 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

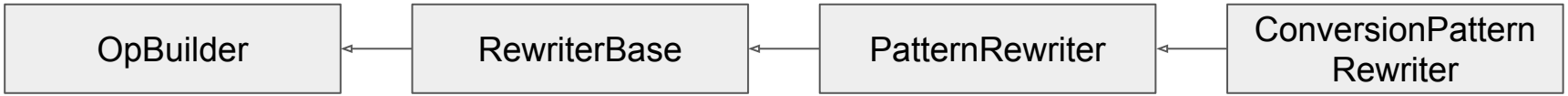


Dialect Conversion: Use Function + Control Flow Patterns

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

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.

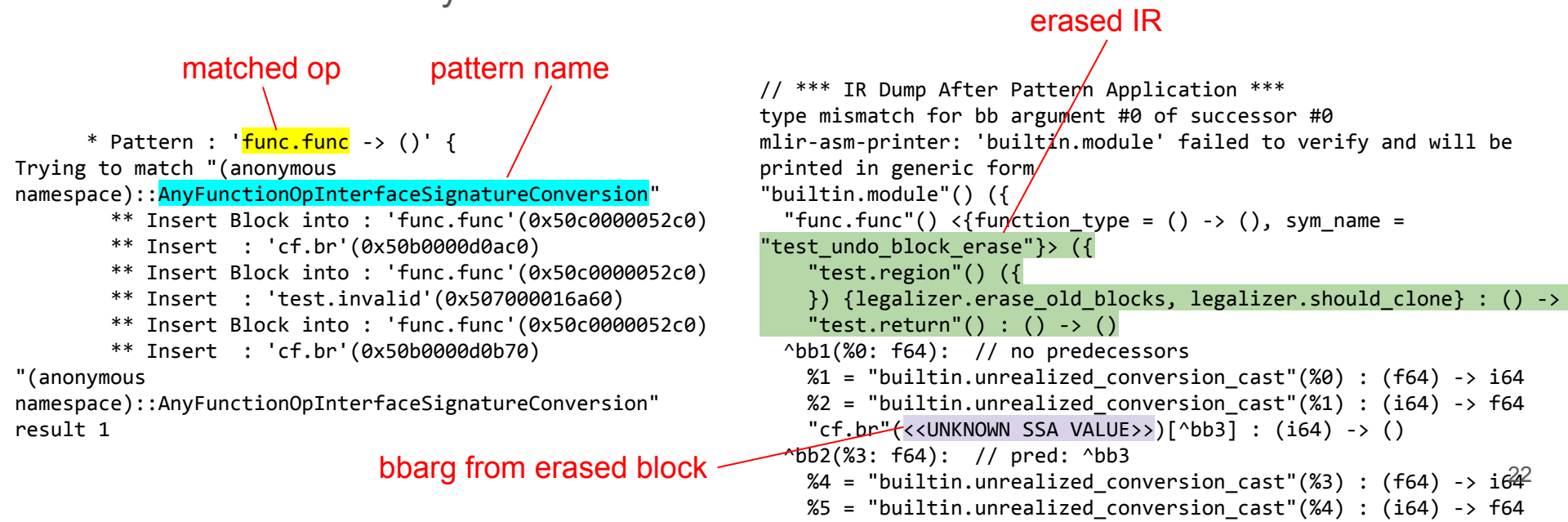
```
// mlir-opt test-legalize-erased-op-with-uses.mlir -test-legalize-unknown-root-patterns
func.func @remove_all_ops(%arg0: i32) -> i32 {
  %0 = builtin.unrealized_conversion_cast to i32
  return %0 : i32
}
```

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.



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.

Future Plans for Dialect Conversion

1:N Conversion Support (RFC)

- `ConversionPatternRewriter` already supports 1:N block argument replacements during block signature conversions.
- New API for replacing ops:
  `replaceOpWithMultiple(Operation *, ArrayRef<ValueRange>)`

one `ValueRange` per op result

Examples in MLIR:

- Sparse tensor → various storage specifier fields
- MemRef → offset, sizes, strides, base pointer, aligned pointer (currently: LLVM struct, aka *MemRef descriptor*)

1:N Conversion Support (RFC)

```
// 1:1 pattern entry point
LogicalResult ConversionPattern::matchAndRewrite(
    Operation *op, ArrayRef<Value> adaptor, ConversionPatternRewriter &r) {

}

// New: 1:N pattern entry point
LogicalResult ConversionPattern::matchAndRewrite(
    Operation *op, ArrayRef<ValueRange> adaptor, ConversionPatternRewriter &r) {
  // Default implementation: Call 1:N version
}
```

1:N Conversion Support (RFC)

```
// 1:1 pattern entry point
LogicalResult OpConversionPattern<FooOp>::matchAndRewrite(
    FooOp op, OpAdaptor adaptor, ConversionPatternRewriter &r) {

}

// New: 1:N pattern entry point
LogicalResult OpConversionPattern<FooOp>::matchAndRewrite(
    FooOp op, OneToNOpAdapator adaptor, ConversionPatternRewriter &r) {
  // Default implementation: Call 1:N version
}
```

1:N Conversion Support (RFC)

- No more argument materializations: Worked around missing 1:N support in `ConversionPattern`. **Only source/target materializations from now.**
  - Argument materialization: Converts 1:N block argument replacements into a single SSA value. Workaround in 1:1 dialect conversion because of 1:N limitations.
- **Delete 1:N dialect conversion** and 1:N type converter infrastructure (`OneToNTypeConversion.h`). Functionality now provided by the “main” dialect conversion.

One-Shot Dialect Conversion (RFC)

- Faster + more efficient: **No rollback** → no extra housekeeping
  - No more `ConversionValueMapping` (a king 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 `RewritePattern`s
- 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?

Manual IR Walk
Greedy Pattern Rewrite Driver
1:1 Dialect Conversion
1:N Dialect Conversion
One-Shot Dialect Conversion
Transform Dialect Integration
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`