Qualcomm

Canonicalization in MLIR – Uniqueness & Equivalence

Mohammed Javed Absar

Principal Engineer, Qualcomm Technologies International, Ltd.

Agenda

- Cover the basics – explain canonicalization in MLIR today
- Describe the problems with canonicalizer
- Understand the problem from the perspective of the principles of ‘canonical-form’
- Provide some solutions and different perspectives
- Conclusion

Canonicalization in MLIR Today - Basics

Rewrite of the IR at the same abstraction level as original but in simpler form.

test.mlir

```
func.func @transpose_scalar_broadcast1(%value: vector<1xf32>) -> vector<1x8xf32> {
  %bcast = vector.broadcast %value : vector<1xf32> to vector<8x1xf32>
  %t = vector.transpose %bcast, [1, 0] : vector<8x1xf32> to vector<1x8xf32>
  return %t : vector<1x8xf32>
}
```

```
$ mlir-opt -canonicalize test.mlir

func.func @transpose_scalar_broadcast1(%arg0: vector<1xf32>) -> vector<1x8xf32> {
    %0 = vector.broadcast %arg0 : vector<1xf32> to vector<1x8xf32>
    return %0 : vector<1x8xf32>
}
```

Canonicalization in MLIR - Basics

include/mlir/Dialect/Vector/IR/VectorOps.td

```
def Vector_TransposeOp :
  Vector_Op<"transpose", [Pure,
    DeclareOpInterfaceMethods<VectorUnrollOpInterface,...

  let hasCanonicalizer = 1;
  let hasFolder = 1;
  let hasVerifier = 1;
}
```

VectorOps.h.inc

```
class TransposeOp : public ::mlir::Op<TransposeOp, ::mlir::OpTrait::ZeroRegions,
       ...> {
  ...
  static void getCanonicalizationPatterns(::mlir::RewritePatternSet &results,
                                          ::mlir::MLIRContext *context);
}
```

Canonicalization in MLIR - Basics

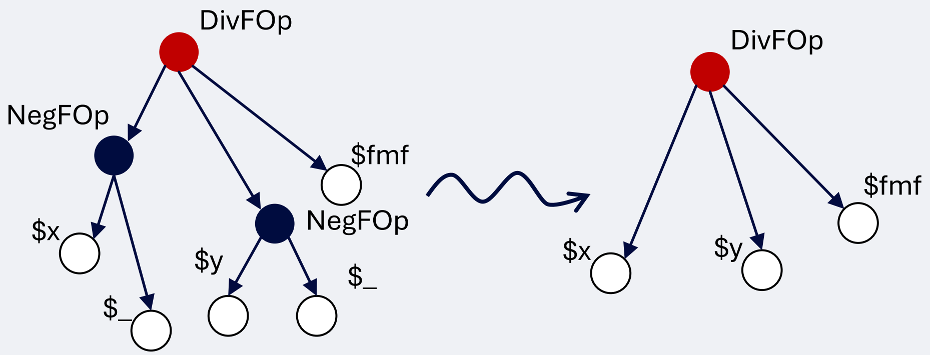
VectorOps.cpp

```
void vector::TransposeOp::getCanonicalizationPatterns(
    RewritePatternSet &results, MLIRContext *context) {

    results.add<FoldTransposeCreateMask, FoldTransposedScalarBroadcast,
              TransposeFolder, FoldTransposeSplat>(context);
}



// Folds transpose(broadcast(<scalar>)) into brodcast(<scalar>).
struct FoldTransposedScalarBroadcast final
    : public OpRewritePattern<vector::TransposeOp> {
  using OpRewritePattern::OpRewritePattern;

  LogicalResult matchAndRewrite(vector::TransposeOp transposeOp,
                                PatternRewriter &rewriter) const override {
    auto bcastOp = transposeOp.getVector().getDefiningOp<vector::BroadcastOp>();
    if (!bcastOp)
      return failure();

    ...
      rewriter.replaceOpWithNewOp<vector::BroadcastOp>(
          transposeOp, transposeOp.getResultVectorType(), bcastOp.getSource());
      return success();
    }
..
```

Canonicalization in MLIR -Basics

ArithCanonicalization.td

```
// divf(negf(x), negf(y)) -> divf(x,y)
// (retain fastmath flags of original divf)
def DivFOfNegF :
    Pat<(Arith_DivFOp (Arith_NegFOp $x, $_), (Arith_NegFOp $y, $_), $fmf),
        (Arith_DivFOp $x, $y, $fmf),
        [(Constraint<CPred<"$0.getType() == $1.getType()">> $x, $y)]>;

#endif // ARITH_PATTERNS
```



ArithCanonicalization.inc

```
struct DivFOfNegF : public ::mlir::RewritePattern {
  DivFOfNegF(::mlir::MLIRContext *context)
      : ::mlir::RewritePattern("arith.divf", 3, context, {"arith.divf"}) {}
  ::llvm::LogicalResult matchAndRewrite(::mlir::Operation *op0,
      ::mlir::PatternRewriter &rewriter) const override {

   auto odsLoc = rewriter.getFusedLoc({tblgen_ops[0]->getLoc(), tblgen_ops[1]->getLoc(), tblgen_ops[2]->getLoc()});
    …
    ::llvm::SmallVector<::mlir::Value, 4> tblgen_repl_values;
    ::mlir::arith::DivFOp tblgen_DivFOp_0;
    {
      ::llvm::SmallVector<::mlir::Value, 4> tblgen_values; (void)tblgen_values;
      ::mlir::arith::DivFOp::Properties tblgen_props; (void)tblgen_props;
      tblgen_values.push_back((*x.begin()));
      tblgen_values.push_back((*y.begin()));
      tblgen_props.fastmath = ::llvm::dyn_cast_if_present<decltype(tblgen_props.fastmath)>(fmf);
      tblgen_DivFOp_0 = rewriter.create<::mlir::arith::DivFOp>(odsLoc, tblgen_values, tblgen_props);
    }
```

Auto-Generated by RewriterGen.cpp

Example - Canonical Form

```
#map = affine_map<(d0, d1, d2) -> (d0, d1, d2)>

func.func @foo(%arg0 : tensor<?x?x?xf32>, %arg1 : tensor<?x?x?xf32>)
             -> (tensor<?x?x?xf32>, tensor<?x?x?xf32>) {
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %c2 = arith.constant 2 : index
  %0 = tensor.dim %arg0, %c0 : tensor<?x?x?xf32>
  %1 = tensor.dim %arg0, %c1 : tensor<?x?x?xf32>
  %2 = tensor.dim %arg0, %c2 : tensor<?x?x?xf32>
  %3 = tensor.empty(%0, %1, %2) : tensor<?x?x?xf32>

  %4, %5 = linalg.generic
          { indexing_maps = [#map, #map, #map, #map],
            iterator_types = ["parallel", "parallel", "parallel"]}
       ins(%arg0, %arg1 : tensor<?x?x?xf32>, tensor<?x?x?xf32>)
       outs(%3, %3 : tensor<?x?x?xf32>, tensor<?x?x?xf32>) {
     ^bb0(%arg2 : f32, %arg3 : f32, %arg4 : f32, %arg5 : f32):
       linalg.yield %arg3, %arg2 : f32, f32
  } -> (tensor<?x?x?xf32>, tensor<?x?x?xf32>)

return %4, %5 : tensor<?x?x?xf32>, tensor<?x?x?xf32>
}
```

Example - Canonical Form

```
$ mlir-opt -canonicalize test.mlir

module {
  func.func @foo(%arg0: tensor<?x?x?xf32>, %arg1: tensor<?x?x?xf32>) -> (tensor<?x?x?xf32>, tensor<?x?x?xf32>) {
    return %arg1, %arg0 : tensor<?x?x?xf32>, tensor<?x?x?xf32>
  }
}
```

Problems

```
%0 = tensor.transpose %0 [1, 0]
      : vector<1x5xf32> to tensor<5x1xf32>
```

```
%0 = tensor.shape_cast %0
      : vector<1x5xf32> to tensor<5x1xf32>
```

If the destination tensor of the insertion of a slice has the same number of elements as the slice, but with a shape that only differs by a prefix of unit-sized dimensions, and if the insertion happens at zero offsets, unit strides and with a size matching the size of the destination, the insertion covers all elements of the destination. The result of such an insertion is equivalent to the slice, with its shape expanded to the type of the destination.

```
%0 = tensor.insert_slice %slice into %x[0, 0, 0, 0, 0][1, 1, 1, 16, 32][1, 1, 1, 1, 1]
      : tensor<16x32xf32> into tensor<1x1x1x16x32xf32>
```

```
%0 = tensor.expand_shape %slice[[0,1,2,3], [4]]
      : tensor<16x32xf32> into tensor<1x1x1x16x32xf32>
```

matthias-springer commented on May 23, 2024 Member ···

One more data point: We've had similar discussions for other `tensor.extract/insert_slice`-related rewrite patterns. E.g.:

- `tensor.extract_slice(tensor.empty)` folding: `populateFoldTensorEmptyPatterns`
- `tensor.insert_slice(tensor.insert_slice)` folding: `populateFoldTensorSubsetOpPatterns` / `populateMergeConsecutiveInsertExtractSlicePatterns`
- `tensor.insert_slice(vector.transfer_write)` folding: `populateFoldTensorSubsetIntoVectorTransferPatterns`
- `tensor.insert_slice(tensor.collapse_shape)` folding: `populateReassociativeReshapeFoldingPatterns`
- etc.

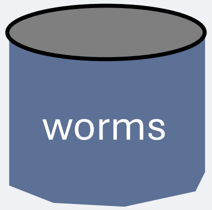
Some of these could have been canonicalization patterns. The problem is that some transformations are hard-coded to specific IR

Canonical Form in IR

An IR at the same abstraction level as the original but in simpler form for subsequent passes/transformations

Problems:

- Define ‘simpler’ form
- There is no single canonical form
- Is it optimization or canonicalization?
- Which canonical form is better ?
- Are you making it better or worse?
- Passes that come to rely on canonicalizer
- Canonicalization is specific to passes, targets
- Canonicalizer taking too long, too big



Principles and Purpose of Canonical Form

““Two programs are equivalent (i.e. for all inputs they produce same output) iff they have the same canonical form”

- Literal equivalence, Algorithmic equivalence, Behavioural equivalence

R. E. Noonan, ACM ‘75

An IR at the same abstraction level as the original but in simpler form for subsequent passes/transformations

Axioms of ‘Simpler Form’?

- Eliminating operation is good [e.g. “X + Identity_+ = X”]
- Folding is good [e.g. “$P_a(P_b(x)) \rightarrow P_{ab}(x)$”] ?
- Canonicalize towards fewer value use [e.g. Op(x,x,y) →Op’(x)]
- Reduce variety in the code to help optimizers/transformations

Kolmogorov Complexity / Entropy

“Length of shortest program producing the same output”

Simplest != Compact

e.g. loop pre-header, loop-exit-block

Why do we have multiple ‘simple’ forms?

What is canonical form?

Canonical form is the unique representation of an abstract object in the given context.

= **Uniqueness** representation : based on some mathematical property of the objects, or agreement on some standard form where uniqueness is possible.

= **Closure** : a mechanism so that objects in non-canonical form can be converted to the canonical form.

= **Equivalence** : two seemingly different things are same if the canonical representation is identical.

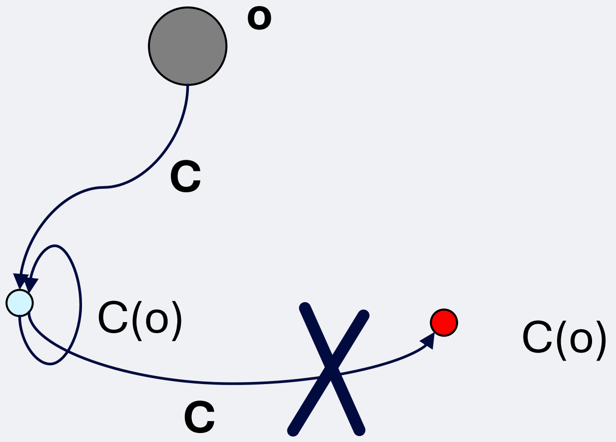
$$\{23, 3, 11\} \rightarrow \{3, 11, 23\} \quad \neq \quad \{23, 5, 4\} \rightarrow \{4, 5, 23\}$$

Transformations/Opt/Query built on top of the "Uniqueness, Closure, and Equivalence" properties.

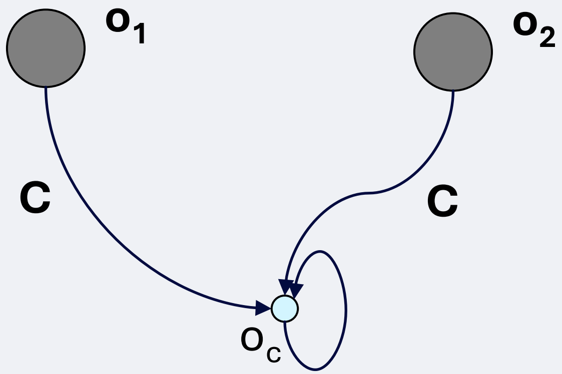
Canonical Form

o : an object or element of a group
C : canonicalizer function

Property 1: C(o) = C(C(o)) **Uniqueness, fixed-point**

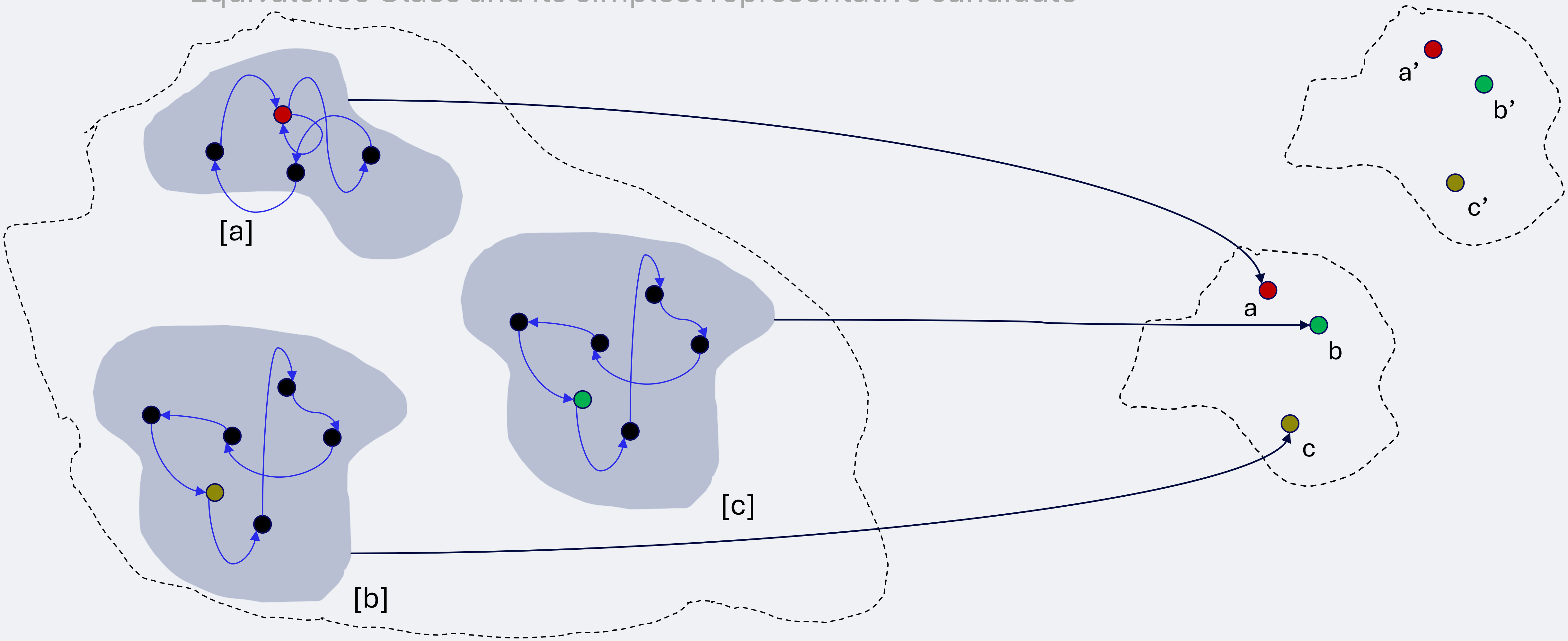


Property 2: $o_1 \sim o_2$ iff $c(o_1) = c(o_2)$ **Equivalence**



Canonical Form in IR

Equivalence Class and its simplest representative candidate



What is canonical form?

Canonical form is the unique representation of an abstract object in the given context.

= Uniqueness representation : based on some fundamental mathematical property of the objects, or agreement on some standard form where uniqueness is possible.

= Closure : a mechanism so that objects in non-canonical form can be converted to the canonical form.

= Equivalence : two seemingly different things are same if the canonical representation is identical.

E.g.

Any number n > 1 can be **represented** in **exactly one way** as product of prime numbers

(closure)

(unique)

Fundamental Theorem of Arithmetic

Canonical-form: 2^3x5^3

Equivalence class : 1000x1, 25x40, 8x125, 10x100, 5x 200, ..., $1x1x1x2^3x1x5^{3x1}$

equivalence class

Uniqueness

Equivalence

Closure

Different Context: 1+1+1 .... (thousand +1s)

Why use canonical form?

Canonical form is the unique representation of an abstract object in the given context.

- To simplify – remove redundancies, answer queries
- To infer if two objects which look different are in fact the same
- Build simpler 'transformations' that need operate easily on the canonical form
- Find problems with the representation – semantics, paradox

$$\begin{bmatrix} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & -2 & -3 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Reduced row echelon form : Gauss-Jordan

Conjunctive Normal Form : (¬P∨Q)∧(P∨¬Q)

$$\bigoplus Z_k$$

Any finite abelian group can be written a direct sum of cyclic group of prime-order.

Chomsky Normal Form:

S_0→ AbB | C
B → AA | AC
C → b | c
A → a | ε

Karnaugh map

slope intercept form: y = mx+c

Lambda calculus: beta normal form

Cayley Table

Database : Codd's normal form (1NF, 2NF, ..)

Fundamental Property – Basis of Canonical IR

Building blocks of Op, Graph

⬇

Orthogonality in and of Ops

No redundancy in representation, values, and computation

Central Problem : Competing 'simplest' versions of IR

Canonical Form of IR

- Canonical form of Op
  - By Definition
    - Eliminate redundancies at **op-definitions** (infer)
  - At construction
    - User supplied identity map ~ default map
  - Post construction - canonicalize, fold
    - `linalg.generic` unused args, unused computation

- Canonical form of Dialect (Design)
  - Non-overlapping Ops / Dialects

- Canonical form of IR sub-graph
  - A lattice structure – descent to simpler and simpler form

Context Matters

Problems

```
%0 = tensor.transpose %0 [1, 0]
        : vector<1x5xf32> to tensor<5x1xf32>
```

```
%0 = tensor.shape_cast %0
        : vector<1x5xf32> to tensor<5x1xf32>
```

If the destination tensor of the insertion of a slice has the same number of elements as the slice, but with a shape that only differs by a prefix of unit-sized dimensions, and if the insertion happens at zero offsets, unit strides and with a size matching the size of the destination, the insertion covers all elements of the destination. The result of such an insertion is equivalent to the slice, with its shape expanded to the type of the destination.

```
%0 = tensor.insert_slice %slice into %x[0, 0, 0, 0, 0][1, 1, 1, 16, 32][1, 1, 1, 1, 1]
        : tensor<16x32xf32> into tensor<1x1x1x16x32xf32>
```

```
%0 = tensor.expand_shape %slice[[0,1,2,3], [4]]
        : tensor<16x32xf32> into tensor<1x1x1x16x32xf32>
```

matthias-springer commented on May 23, 2024 Member ···

One more data point: We've had similar discussions for other `tensor.extract/insert_slice`-related rewrite patterns. E.g.:

- `tensor.extract_slice(tensor.empty)` folding: `populateFoldTensorEmptyPatterns`
- `tensor.insert_slice(tensor.insert_slice)` folding: `populateFoldTensorSubsetOpPatterns` / `populateMergeConsecutiveInsertExtractSlicePatterns`
- `tensor.insert_slice(vector.transfer_write)` folding: `populateFoldTensorSubsetIntoVectorTransferPatterns`
- `tensor.insert_slice(tensor.collapse_shape)` folding: `populateReassociativeReshapeFoldingPatterns`
- etc.

Some of these could have been canonicalization patterns. The problem is that some transformations are hard-coded to specific IR

Perspectives – Canonical Form

- An op has a canonical form either in general, per pass or per phase - contextual.
- Generic forms (higher abstraction) – harder to identify canonical form
- There's no true canonical form.
- Canonicalization== Pre-processing – "is things you do to prepare the IR for a kind of transformation"
- There's a bunch of different canonical forms for different objectives
- Canonicalization should not 'drop semantics' that cannot be recovered easily (discardable attributes).
- Need for better documentation and definition of best practices.
- Target dependent canonicalization
- canonicalization should not be required for correctness (of the IR semantics, verification, etc), it is often *required* for a pass to work at all (pattern matchers)
- Canonical form and Interfaces

Perspectives – Canonicalization Process

- As canonicalization is run often it should not be too computationally expensive.
- Canonicalization should not be required for correctness. They should work correctly with all instances of the canonicalization pass removed. But some say that in reality it is often required for some pass to work at all.
- Bar for something to be in the 'canonicalization pass’ should be high.
- Avoid cycles in canonicalization (i.e. a lattice like approach)
- Canonicalization is not a good place for complicated cost-models. - 'should always improve performance’.
- Try not to have lowering pipeline relying on a specific canonical form for correctness, - patterns that preserve the abstraction level make sense for canonicalization vs. patterns that, arguably, perform lowering/decomposition.

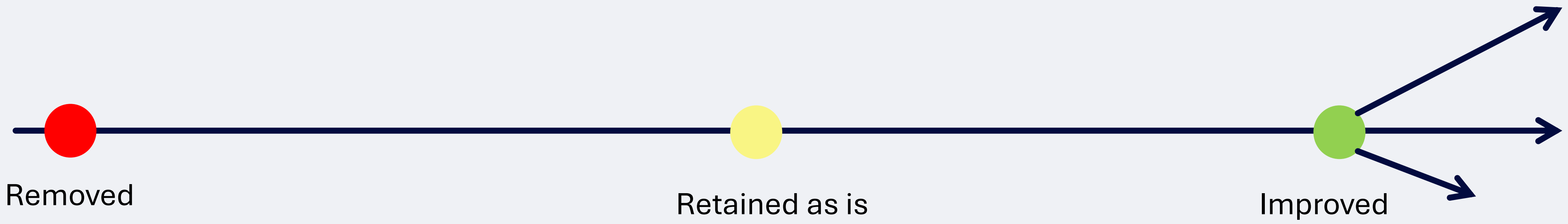
My perspective

- Aspiring for “a” canonical-form not realistic (overlapping ops already exist)
- View canonicalizer as ‘simplifier’ – it is useful to remove clutter in IR
- Keep it thin – OR restructure it to address different contexts

POLL

POLL

- Should ‘canonicalizer’ be removed?
- Should the ‘canonicalizer’ be retained as is?
- Should the ‘canonicalizer’ be improved?
- Should it –
  - Have levels?
  - Restructured to have more structure (instead of catch-all)?



Canonicalization in MLIR

```
module {
  func.func @tensor_bitcast_chain_ok(%arg0: tensor<2xi32>) -> tensor<2xf32> {
    %0 = tensor.bitcast %arg0 : tensor<2xi32> to tensor<2xui32>
    %1 = tensor.bitcast %0 : tensor<2xui32> to tensor<2xf32>
    return %1 : tensor<2xf32>
  }
}
```

```
$ mlir-opt -canonicalize test.mlir
module {
  func.func @tensor_bitcast_chain_ok(%arg0: tensor<2xi32>) -> tensor<2xf32> {
    %0 = tensor.bitcast %arg0 : tensor<2xi32> to tensor<2xf32>
    return %0 : tensor<2xf32>
  }
}
```

Canonicalization

```
scf.if %5#1 {
    memref.dealloc %base_buffer_10 : memref<f32, 2>
}
scf.if %true {
  memref.dealloc %alloc : memref<f16, 1>
}
scf.if %true {
  memref.dealloc %alloc_1 : memref<524288xf16, 1>
}
scf.if %true {
  memref.dealloc %alloc_2 : memref<1048576xf16>
}
scf.if %true {
  memref.dealloc %alloc_3 : memref<524288xf16, 1>
}
scf.if %true {
  memref.dealloc %alloc_4 : memref<524288xf16, 1>
}
```

Canonicalization

```
func.func @empty_insert_slice(%arg0 : tensor<0x2xi8>, %arg1 : tensor<3x3xi8>) -> tensor<3x3xi8> {
  %0 = tensor.insert_slice %arg0 into %arg1[0, 0] [0, 2] [1, 1] : tensor<0x2xi8> into tensor<3x3xi8>
  return %0 : tensor<3x3xi8>
}
-----------------------------------------  simplifies to  -----------------------------------
module {
  func.func @empty_insert_slice(%arg0: tensor<0x2xi8>, %arg1: tensor<3x3xi8>) -> tensor<3x3xi8> {
    return %arg1 : tensor<3x3xi8>
  }
}
```

```
module {
  func.func @empty_insert_slice(%arg0: tensor<3x3xi8>, %arg1: tensor<3x3xi8>) -> tensor<3x3xi8> {
    %inserted_slice = tensor.insert_slice %arg0 into %arg1[0, 0] [3, 3] [1, 1] : tensor<3x3xi8> into tensor<3x3xi8>
    return %inserted_slice : tensor<3x3xi8>
  }
}

-----------------------------------------  simplifies to  -----------------------------------
module {
  func.func @empty_insert_slice(%arg0: tensor<3x3xi8>, %arg1: tensor<3x3xi8>) -> tensor<3x3xi8> {
    return %arg0 : tensor<3x3xi8>
  }
}
```

Canonicalization

```
module {
  func.func @buffer_cast_of_tensor_load(%arg0: memref<?xf32>) -> memref<?xf32> {
    %0 = bufferization.to_tensor %arg0 : memref<?xf32> to tensor<?xf32>
    %1 = bufferization.to_memref %0 : tensor<?xf32> to memref<?xf32>
    return %1 : memref<?xf32>
  }
}----------------------------------------  simplifies to  -----------------------------------
module {
  func.func @buffer_cast_of_tensor_load(%arg0: memref<?xf32>) -> memref<?xf32> {
    return %arg0 : memref<?xf32>
  }
}
```

```
func.func @canonicalize_buffer_cast_of_tensor_load_different_address_space(%arg0: memref<?xf32, 2>)
                                                                                  -> memref<?xf32, 7> {
  %0 = bufferization.to_tensor %arg0 : memref<?xf32, 2> to tensor<?xf32, 7 : i64>
  %1 = bufferization.to_memref %0 : tensor<?xf32, 7 : i64> to memref<?xf32, 7>
  return %1 : memref<?xf32, 7>
}
----------------------------------------  simplifies to  -----------------------------------
func.func @canonicalize_buffer_cast_of_tensor_load_different_address_space(%arg0: memref<?xf32, 2>)
                                                                                  -> memref<?xf32, 7> {
  %c0 = arith.constant 0 : index
  %dim = memref.dim %arg0, %c0 : memref<?xf32, 2>
  %alloc = memref.alloc(%dim) : memref<?xf32, 7>
  memref.copy %arg0, %alloc : memref<?xf32, 2> to memref<?xf32, 7>
  return %alloc : memref<?xf32, 7>
}
```

Canonicalization

```
module {
  func.func @partial_sink(%arg0: tensor<10x10xf32>, %arg1: index, %arg2: index) -> tensor<?x?x?xf32> {
    %c10 = arith.constant 10 : index
    %cast = tensor.cast %arg0 : tensor<10x10xf32> to tensor<?x?xf32>
    %expanded = tensor.expand_shape %cast [[0, 1], [2]] output_shape [%arg1, %arg2, %c10]
                  : tensor<?x?xf32> into tensor<?x?x?xf32>
    %0 = linalg.add
               ins(%expanded, %expanded : tensor<?x?x?xf32>, tensor<?x?x?xf32>)
               outs(%expanded : tensor<?x?x?xf32>) -> tensor<?x?x?xf32>
    return %0 : tensor<?x?x?xf32>
  }
}
```

```
$ mlir-opt -canonicalize test.mlir
module {
  func.func @partial_sink(%arg0: tensor<10x10xf32>, %arg1: index, %arg2: index) -> tensor<?x?x?xf32> {
    %cast = tensor.cast %arg0 : tensor<10x10xf32> to tensor<?x10xf32>
    %expanded = tensor.expand_shape %cast [[0, 1], [2]] output_shape [%arg1, %arg2, 10]
                  : tensor<?x10xf32> into tensor<?x?x10xf32>
    %0 = linalg.add
               ins(%expanded, %expanded : tensor<?x?x10xf32>, tensor<?x?x10xf32>)
               outs(%expanded : tensor<?x?x10xf32>) -> tensor<?x?x10xf32>
    %cast_0 = tensor.cast %0 : tensor<?x?x10xf32> to tensor<?x?x?xf32>
    return %cast_0 : tensor<?x?x?xf32>
  }
}
```

Canonicalization

```
module {
  func.func @bf16_branch_vector(%arg0: vector<32x32x32xbf16>) -> vector<32x32x32xbf16> {
    %0 = arith.extf %arg0 fastmath<contract> : vector<32x32x32xbf16> to vector<32x32x32xf32>
    %1 = math.absf %0 : vector<32x32x32xf32>
    %2 = arith.truncf %1 fastmath<contract> : vector<32x32x32xf32> to vector<32x32x32xbf16>
    %3 = arith.extf %2 fastmath<contract> : vector<32x32x32xbf16> to vector<32x32x32xf32>
    %4 = math.sin %3 : vector<32x32x32xf32>
    %5 = arith.truncf %4 fastmath<contract> : vector<32x32x32xf32> to vector<32x32x32xbf16>
    return %5 : vector<32x32x32xbf16>
  }
}
```

```
$ mlir-opt -canonicalize test.mlir
module {
  func.func @bf16_branch_vector(%arg0: vector<32x32x32xbf16>) -> vector<32x32x32xbf16> {
    %0 = arith.extf %arg0 fastmath<contract> : vector<32x32x32xbf16> to vector<32x32x32xf32>
    %1 = math.absf %0 : vector<32x32x32xf32>
    %2 = math.sin %1 : vector<32x32x32xf32>
    %3 = arith.truncf %2 fastmath<contract> : vector<32x32x32xf32> to vector<32x32x32xbf16>
    return %3 : vector<32x32x32xbf16>
  }
}
```

Conclusion

- Canonical form is a useful concept
- Widely useful in other disciplines (mathematics)
- Applying canonical principles can be difficult in IR
  - True canonical form begins at IR design, dialect design level (orthogonality)
- Simpler approach
  - Canonical form is highly context/target dependent
  - Practically speaking, “it’s a pre-processing of IR that helps your set of transformation(s)”

Thank you

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.

Follow us on:
For more information, visit us at qualcomm.com & qualcomm.com/blog