

Louis / CIRCT Docker Promotion Time for Tomorrow

- `git clone https://github.com/Bynaryman/MLIR_ACM_Summer_School_2026_CIRCT`
- `cd MLIR_ACM_Summer_School_2026_CIRCT/tutorial`
- `docker build -t mlir-summer-school-2026-circt .`
- `docker run -it mlir-summer-school-2026-circt`

OR

- `docker run -it ghcr.io/bynaryman/mlir-summer-school-2026-circt:latest`



MLIR Fundamentals: Understanding Interfaces

Matthias Springer + Kunwar Grover
ACM Europe School on MLIR 2026 – August 12, 2026

Outline

1. What are interfaces and how are they implemented in MLIR?
2. Operation interfaces that abstract over operands / attributes / ...
3. Operation interfaces that model control flow
4. Operation interfaces that model operation semantics
5. Operation interfaces that drive analyses
6. Operation interfaces that drive transformations
7. Type interfaces
8. Attribute interfaces
9. Dialect interfaces



Interfaces

- Abstract over common functionality of a operation / type / attribute / dialect.
- Build generic analyses / transformations.
- Can be implemented when defining the operation or at a later point of time (even in a different compilation unit).



Defining an Interface

```
def OpAsmOpInterface : OpInterface<"OpAsmOpInterface"> {  
  let description = [{ ... }];  
  
  let methods = [  
    InterfaceMethod<[{  
      Get a special name to use when printing the results of this operation.  
    }], "void", "getAsmResultNames", (ins "OpAsmSetValueNameFn":$fn), "", "return;">  
  
    InterfaceMethod<...>  
  ];  
}
```

name of generated C++ class

documentation for the website

// Conceptually:
virtual void getAsmResultNames(OpAsmSetValueNameFn fn) {
 return;
}



Implementing an Interface during Op Definition

```
//=====//  
// ExtractOp  
//=====//  
  
def Tensor_ExtractOp : TensorOp<"extract", [  
  declareOpInterfaceMethods<OpAsmOpInterface, ["getAsmResultNames"]>,  
  Pure,  
  TypesMatchWith<"result type matches element type of tensor",  
    "tensor", "result",  
    "::llvm::cast<TensorType>($self).getElementType()">> {  
    let summary = "element extraction operation";  
    let description = [{  
      The `tensor.extract` op reads a ranked tensor and returns one element as  
      specified by the given indices. The result of the op is a value with the  
      same type as the elements of the tensor. The arity of indices must match  
      the rank of the accessed value. All indices should all be of `index` type.  
    }  
    Example:  
    ```mlir  
 %4 = tensor.extract %t[%1, %2] : tensor<4x4xi32>
 %5 = tensor.extract %rt[%1, %2] : tensor<?x?xi32>
    ```  
  ]};  
  
  let arguments = (ins AnyRankedTensor:$tensor, Variadic<Index>:$indices);  
  let results = (outs AnyType:$result);  
  let assemblyFormat = "$tensor `[' $indices `]' attr-dict `:` type($tensor)";  
  
  let hasCanonicalizer = 1;  
  let hasFolder = 1;  
  let hasVerifier = 1;  
};
```

Implements OpAsmOpInterface and
pre-declares getAsmResultNames.

```
// Conceptually:  
void getAsmResultNames(OpAsmSetValueNameFn fn) override;
```

```
%extracted0 = tensor.extract %t [%c1] : tensor<9xf32>
```



Implementing an Interface during Op Definition

```
//=====//
// ExtractOp
//=====//

def Tensor_ExtractOp : TensorOp<"extract", [
  declareOpInterfaceMethods<OpAsmOpInterface, ["getAsmResultNames"]>,
  Pure,
  TypesMatchWith<"result type matches element type of tensor",
  "tensor", "result",
  "::llvm::cast<TensorType>($self).getElementType()">> {
let summary = "element extraction operation";
let description = [
  The `tensor.extract` op reads a ranked tensor and returns one element as
  specified by the given indices. The result of the op is a value with the
  same type as the elements of the tensor. The arity of indices must match
  the rank of the accessed value. All indices should all be of `index` type.

Example:

```mlir
%4 = tensor.extract %t[%1, %2] : tensor<4x4xi32>
%5 = tensor.extract %rt[%1, %2] : tensor<?x?xi32>
```

  ];
};

let arguments = (ins AnyRankedTensor:$tensor, Variadic<Index>:$indices);
let results = (outs AnyType:$result);
let assemblyFormat = "$tensor '[' $indices `]' attr-dict `:' type($tensor)";

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

Implements OpAsmOpInterface and pre-declares getAsmResultNames.

// Conceptually:
void getAsmResultNames(OpAsmSetValueNameFn fn) **override**;

void TensorOp::getAsmResultNames(OpAsmSetValueNameFn fn) {
 fn(getResult(), "extracted");
}

%extracted0 = tensor.extract %t [%c1] : tensor<9xf32>



Implementing an Interface during Op Definition

```
//=====//  
// ExtractOp  
//=====//  
  
def Tensor_ExtractOp : Tensor_Op<"extract", [  
  DeclareOpInterfaceMethods<OpAsmOpInterface, ["getAsmResultNames"]>,  
  Pure,  
  TypesMatchWith<"result type matches element type of tensor",  
    "tensor", "result",  
    "::llvm::cast<TensorType>($self).getElementType()">] {  
  let summary = "element extraction operation";  
  let description = [  
    The `tensor.extract` op reads a ranked tensor and returns one element as  
    specified by the given indices. The result of the op is a value with the  
    same type as the elements of the tensor. The arity of indices must match  
    the rank of the accessed value. All indices should all be of `index` type.  
  
    Example:  
  
    ```mlir  
 %4 = tensor.extract %t[%1, %2] : tensor<4x4xi32>
 %5 = tensor.extract %rt[%1, %2] : tensor<?x?xi32>
    ```  
  ];  
  let arguments = (ins AnyRankedTensor:$tensor, Variadic<Index>:$indices);  
  let results = (outs AnyType:$result);  
  let assemblyFormat = "$tensor `[` $indices `]` attr-dict `:` type($tensor)";  
  
  let hasCanonicalizer = 1;  
  let hasFolder = 1;  
  let hasVerifier = 1;  
}
```

Expands to AlwaysSpeculatable and
NoMemoryEffects.

Expands to MemoryEffectOpInterface
with empty getEffects implementation.

```
// Conceptually:  
void getEffects(SmallVectorImpl<...> &) override { }
```

```
%extracted0 = tensor.extract %t [%c1] : tensor<9xf32>
```



Implementing an Interface via External Model

```
struct ExtractOpInterface
: public BufferizableOpInterface::ExternalModel<ExtractOpInterface,
        tensor::ExtractOp> {
    LogicalResult bufferize(Operation *op, RewriterBase &rewriter,
        const BufferizationOptions &options,
        BufferizationState &state) const { /* ... */ }
};

void mlir::tensor::registerBufferizableOpInterfaceExternalModels(
    DialectRegistry &registry) {
    registry.addExtension(+[](MLIRContext *ctx, tensor::TensorDialect *dialect) {
        ExtractOp::attachInterface<ExtractOpInterface>(*ctx);
    });
}
```

MLIRTensorTransforms

```
void TensorDialect::initialize() {
    addOperations<
#define GET_OP_LIST
#include "mlir/Dialect/Tensor/IR/TensorOps.cpp.inc"
    >();
    declarePromisedInterface<bufferization::BufferizableOpInterface, ExtractOp>();
}
```

MLIRTensorDialect



Using an Interface

```
Operation *op = ...;
```

```
// Asserts if the interface was promised but not registered.
```

```
auto bufferizableOp = cast<BufferizableOpInterface>(op);
```

```
bufferizableOp.bufferize(...);
```

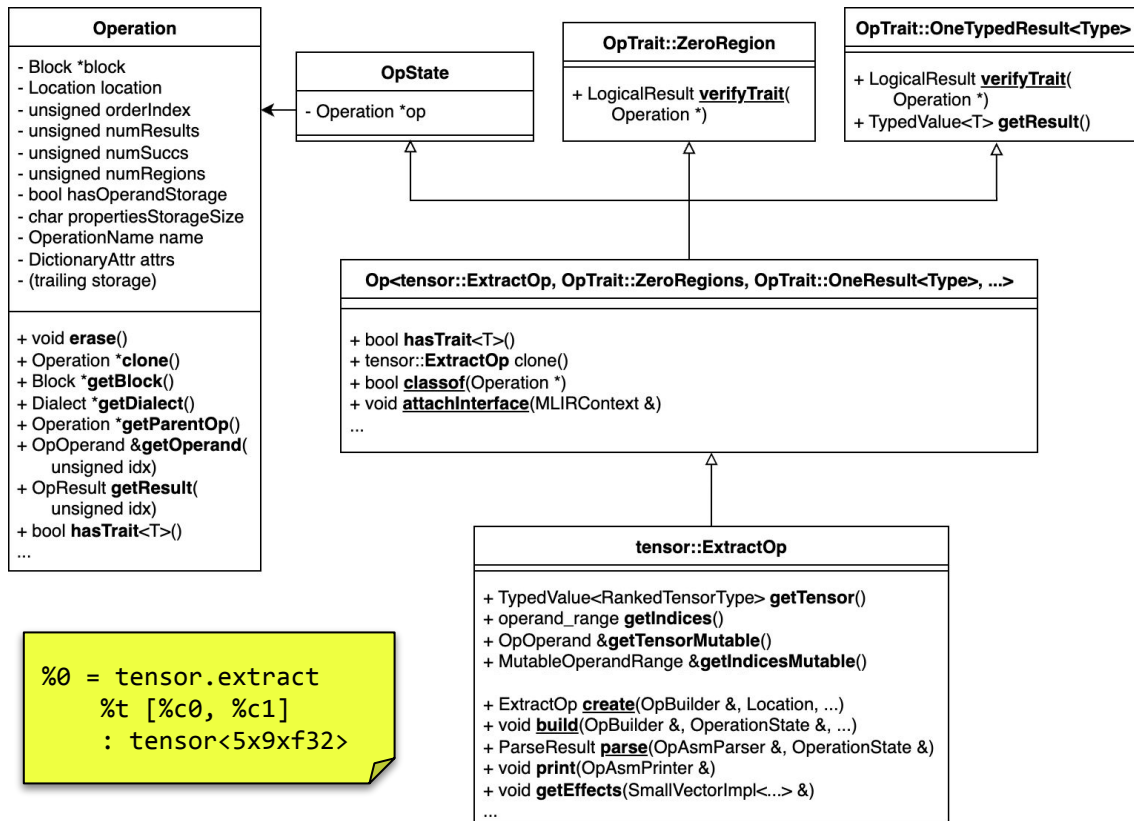
```
// This won't work if the interface was attached via external model.
```

```
tensor::ExtractOp extractOp = builder.create<tensor::ExtractOp>(...);
```

```
extractOp.bufferize(...);
```



Operations and Traits (simplified)

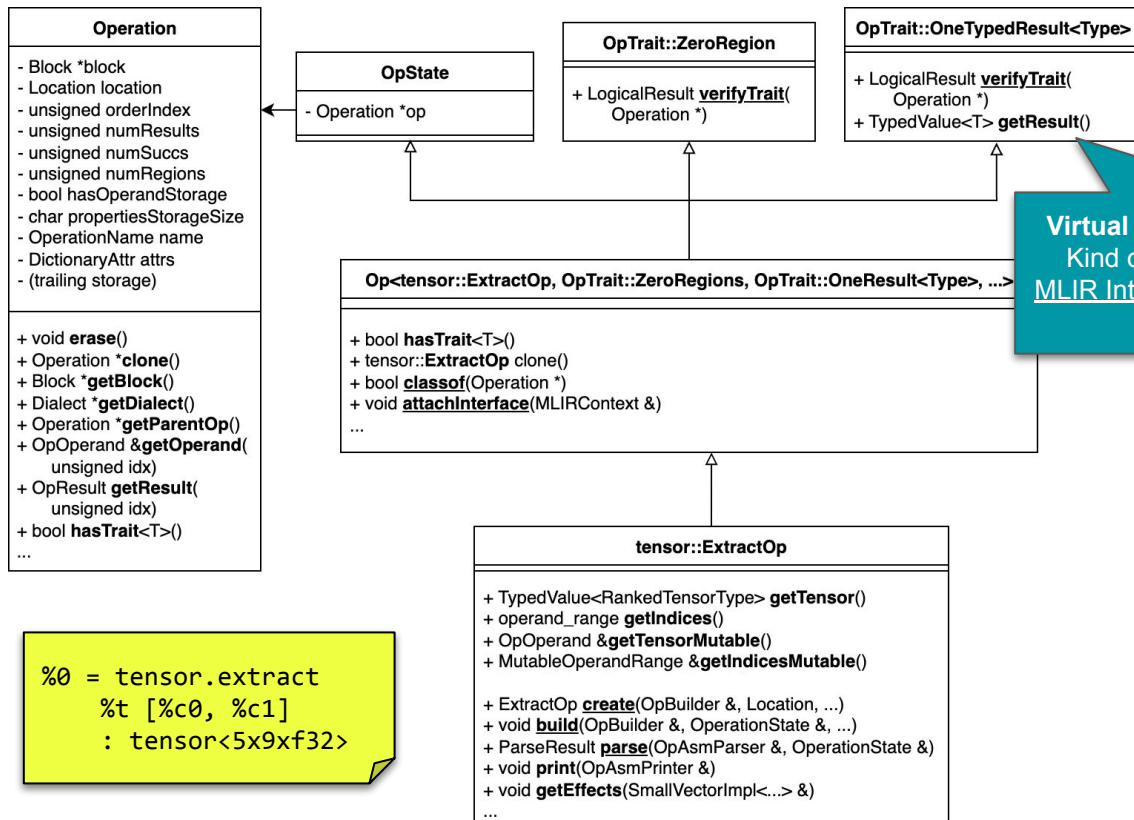


```
%0 = tensor.extract
    %t [%c0, %c1]
    : tensor<5x9xf32>
```

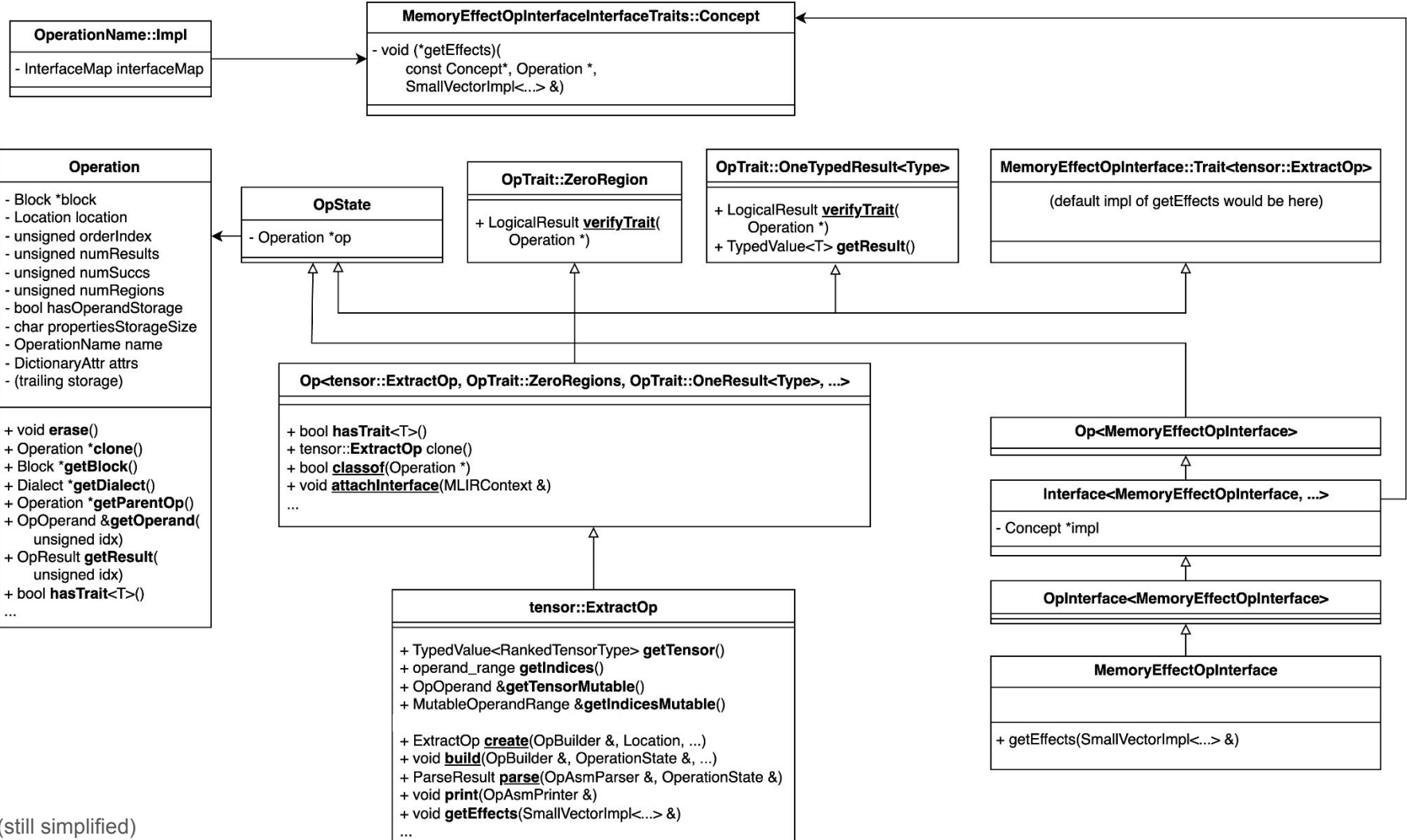
- All state is stored in Operation.
- `tensor::ExtractOp` is auto-generated and contains convenience functions with richer types, operand names, etc.
- `Op<...>` provides common convenience functions.
- Traits can provide common convenience function on a per-op basis.



Operations and Traits (simplified)



- All state is stored in `Operation`.
- `tensor::ExtractOp` is an example of an op interface. For details, see [Deep Dive on OpInterface Implementation](#) (Amini, EuroLLVM 2024).
- `Op<...>` provides common convenience functions.
- Traits can provide common convenience function on a per-op basis.



(still simplified)

Interfaces that Abstract over Operands / Attributes / ...



ShapedDimOpInterface

| ShapedDimOpInterface |
|---|
| + Value getShapedValue()
+ OpFoldResult getDimension() |

- Provides access to shaped value and dimension.
- Used in the Affine dialect to prevent a dependency on TensorDialect.
- Can the dependency on MemRefDialect be removed?

```
%0 = memref.dim %m, %c0 : memref<?xf32>  
  
%1 = tensor.dim %t, %c0 : tensor<?xf32>
```



OffsetSizeAndStrideOpInterface

OffsetSizeAndStrideOpInterface

```
+ unsigned getOffsetSizeAndStrideStartOperandIndex()  
+ std::array<unsigned, 3> getArrayAttrMaxRanks()  
+ OperandRange getOffsets()  
+ OperandRange getSizes()  
+ OperandRange getStrides()  
+ ArrayRef<int64_t> getStaticOffsets()  
+ ArrayRef<int64_t> getStaticSizes()  
+ ArrayRef<int64_t> getStaticStrides()  
+ SmallVector<OpFoldResult> getMixedOffsets()  
+ SmallVector<OpFoldResult> getMixedSizes()  
+ SmallVector<OpFoldResult> getMixedStrides()  
+ bool isDynamicOffset(unsigned idx)  
+ bool isDynamicSize(unsigned idx)  
+ bool isDynamicStride(unsigned idx)  
+ Value getDynamicOffset(unsigned idx)  
+ Value getDynamicSize(unsigned idx)  
+ Value getDynamicStride(unsigned idx)  
+ int64_t getStaticOffset(unsigned idx)  
+ int64_t getStaticSize(unsigned idx)  
+ int64_t getStaticStride(unsigned idx)  
+ unsigned getIndexOfDynamicOffset(unsigned idx)  
+ unsigned getIndexOfDynamicSize(unsigned idx)  
+ unsigned getIndexOfDynamicStride(unsigned idx)  
+ bool isSameAs(  
    OffsetSizeAndStrideOpInterface other,  
    function<bool(OpFoldResult, OpFoldResult)> cmp)  
+ bool hasUnitStride()  
+ bool hasZeroOffset()
```

- Provides uniform access to offsets, sizes, strides.
- Supports mixed static/dynamic values.
- Provides uniform op verification:
 - Matching number of offsets, sizes, strides.
 - Correct number of dynamic operands.

```
%0 = tensor.extract_slice %src [0, %0][15, %sz][1, 1]  
      : tensor<?x?xf32> to tensor<15x?xf32>  
  
%1 = tensor.insert_slice %src into %dest [0, %0][15, %sz][1, 1]  
      : tensor<15x?xf32> into tensor<?x1024xf32>  
  
%2 = memref.subview %buffer [0][15][2]  
      : memref<15xf32> to memref<5xf32, strided<[2]>>
```



OffsetSizeAndStrideOpInterface

OffsetSizeAndStrideOpInterface

```
+ unsigned getOffsetSizeAndStrideStartOperandIndex()  
+ std::array<unsigned, 3> getArrayAttrMaxRanks()  
+ OperandRange getOffsets()  
+ OperandRange getSizes()  
+ OperandRange getStrides()  
+ ArrayRef<int64_t> getStaticOffsets()  
+ ArrayRef<int64_t> getStaticSizes()  
+ ArrayRef<int64_t> getStaticStrides()  
+ SmallVector<OpFoldResult> getMixedOffsets()  
+ SmallVector<OpFoldResult> getMixedSizes()  
+ SmallVector<OpFoldResult> getMixedStrides()  
+ bool isDynamicOffset(unsigned idx)  
+ bool isDynamicSize(unsigned idx)  
+ bool isDynamicStride(unsigned idx)  
+ Value getDynamicOffset(unsigned idx)  
+ Value getDynamicSize(unsigned idx)  
+ Value getDynamicStride(unsigned idx)  
+ int64_t getStaticOffset(unsigned idx)  
+ int64_t getStaticSize(unsigned idx)  
+ int64_t getStaticStride(unsigned idx)  
+ unsigned getIndexOfDynamicOffset(unsigned idx)  
+ unsigned getIndexOfDynamicSize(unsigned idx)  
+ unsigned getIndexOfDynamicStride(unsigned idx)  
+ bool isSameAs(  
    OffsetSizeAndStrideOpInterface other,  
    function<bool(OpFoldResult, OpFoldResult)> cmp)  
+ bool hasUnitStride()  
+ bool hasZeroOffset()
```

- Provides uniform access to offsets, sizes, strides.
- Supports mixed static/dynamic values.
- Provides uniform op verification:
 - Matching number of offsets, sizes, strides.
 - Correct number of dynamic operands.

```
static_offsets = [0, -9223372036854775807]  
offsets = [%o]
```

```
%0 = tensor.extract_slice %src [0, %o][15, %sz][1, 1]  
      : tensor<?x?xf32> to tensor<15x?xf32>
```

```
%1 = tensor.insert_slice %src into %dest [0, %o][15, %sz][1, 1]  
      : tensor<15x?xf32> into tensor<?x1024xf32>
```

```
%2 = memref.subview %buffer [0][15][2]  
      : memref<15xf32> to memref<5xf32, strided<[2]>>
```



OffsetSizeAndStrideOpInterface

OffsetSizeAndStrideOpInterface

```
+ unsigned getOffsetSizeAndStrideStartOperandIndex()  
+ std::array<unsigned, 3> getArrayAttrMaxRanks()  
+ OperandRange getOffsets()  
+ OperandRange getSizes()  
+ OperandRange getStrides()  
+ ArrayRef<int64_t> getStaticOffsets()  
+ ArrayRef<int64_t> getStaticSizes()  
+ ArrayRef<int64_t> getStaticStrides()  
+ SmallVector<OpFoldResult> getMixedOffsets()  
+ SmallVector<OpFoldResult> getMixedSizes()  
+ SmallVector<OpFoldResult> getMixedStrides()  
+ bool isDynamicOffset(unsigned idx)  
+ bool isDynamicSize(unsigned idx)  
+ bool isDynamicStride(unsigned idx)  
+ Value getDynamicOffset(unsigned idx)  
+ Value getDynamicSize(unsigned idx)  
+ Value getDynamicStride(unsigned idx)  
+ int64_t getStaticOffset(unsigned idx)  
+ int64_t getStaticSize(unsigned idx)  
+ int64_t getStaticStride(unsigned idx)  
+ unsigned getIndexOfDynamicOffset(unsigned idx)  
+ unsigned getIndexOfDynamicSize(unsigned idx)  
+ unsigned getIndexOfDynamicStride(unsigned idx)  
+ bool isSameAs(  
    OffsetSizeAndStrideOpInterface other,  
    function<bool(OpFoldResult, OpFoldResult)> cmp)  
+ bool hasUnitStride()  
+ bool hasZeroOffset()
```

- Provides uniform access to offsets, sizes, strides.
- Supports mixed static/dynamic values.
- Provides uniform op verification:
 - Matching number of offsets, sizes, strides.
 - Correct number of dynamic operands.

TODO: These should not be interface methods.

```
%0 = tensor.extract_slice %src [0, %0][15, %sz][1, 1]  
      : tensor<?x?xf32> to tensor<15x?xf32>  
  
%1 = tensor.insert_slice %src into %dest [0, %0][15, %sz][1, 1]  
      : tensor<15x?xf32> into tensor<?x1024xf32>  
  
%2 = memref.subview %buffer [0][15][2]  
      : memref<15xf32> to memref<5xf32, strided<[2]>>
```



ViewLikeOpInterface

| ViewLikeOpInterface |
|--|
| + Value getViewSource()
+ Value getViewDest() |

- Provides uniform access to the view source / destination.
- Implies some sort of aliasing. **The interface description is vague.**
- Used by patterns
(memref.extract_aligned_pointer_as_index) and
bufferization-related passes to understand data flow.

```
%0 = memref.assume_alignment %buffer, 128 : memref<?xf32>
%1 = memref.cast %buffer : memref<?xf32> to memref<64xf32>
%2 = memref.subview %buffer[0][15][2] : memref<15xf32> to memref<5xf32, strided<[2]>>
%3 = llvm.getelementptr %ptr[%offset] : (!llvm.ptr, i64) -> !llvm.ptr, i32
%4 = ptr.ptr_add %x, %off : !ptr.ptr<#ptr.generic_space>, i32
%5 = amdgpu.fat_raw_buffer_cast %m : memref<...> to memref<...>
%6 = xegpu.create_tdsc %src, %cst : ui64, vector<4xindex> -> !xegpu.tensor_desc<...>
```



ViewLikeOpInterface

| ViewLikeOpInterface |
|--|
| + Value getViewSource()
+ Value getViewDest() |

- Provides uniform access to the view source / destination.
- Implies some sort of aliasing. **The interface description is vague.**
- Used by patterns

(memref.extract_alias
bufferization-related pa

A view-like operation "views" a buffer in a potentially different way. It takes in a (view of) buffer (and potentially some other operands) and returns another view of buffer.

```
%0 = memref.assume_alignment %buffer : memref<?xf32> to memref<64xf32>  
%1 = memref.cast %buffer : memref<?xf32> to memref<64xf32>  
%2 = memref.subview %buffer[0][15][2] : memref<15xf32> to memref<5xf32, strided<[2]>>  
%3 = llvm.getelementptr %ptr[%offset] : (!llvm.ptr, i64) -> !llvm.ptr, i32  
%4 = ptr.ptr_add %x, %off : !ptr.ptr<#ptr.generic_space>, i32  
%5 = amdgpu.fat_raw_buffer_cast %m : memref<...> to memref<...>  
%6 = xegpu.create_tdsc %src, %cst : ui64, vector<4xindex> -> !xegpu.tensor_desc<...>
```



ArgAndResultAttrsOpInterface

ArgAndResultAttrsOpInterface

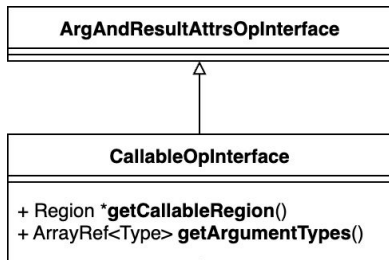
```
+ ArrayAttr getArgAttrsAttr()  
+ ArrayAttr getResAttrAttr()  
+ void setArgAttrsAttr(ArrayAttr attrs)  
+ void setResAttrAttr(ArrayAttr attrs)  
+ Attribute removeArgAttrsAttr()  
+ Attribute removeResAttrAttr()
```

- Provides uniform access to argument and result attributes.
- Applicable to calling ops and callable ops.
- One array attribute per argument / result.

```
func.func @main(%arg0 : tensor<?xf32> {bufferization.writable = true}) ...  
  
gpu.func @kernel(%arg0 : tensor<?xf32> {bufferization.writable = true}) ...
```



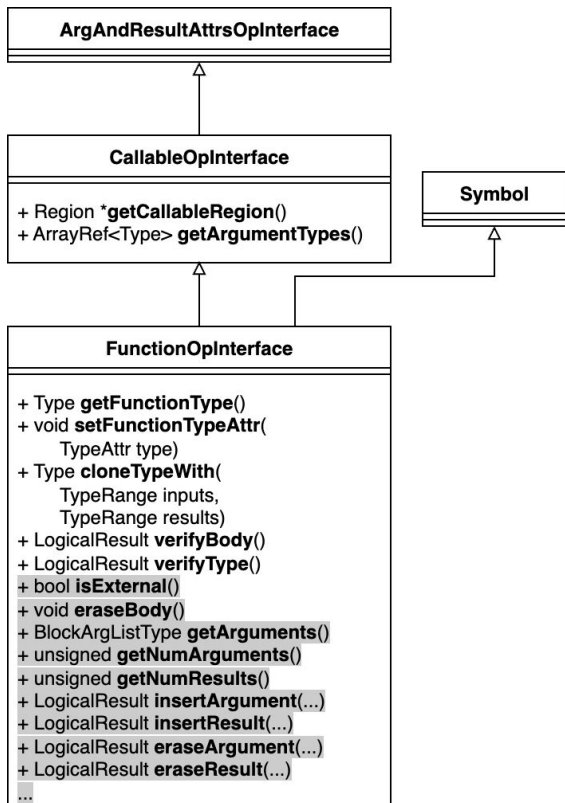
CallableOpInterface



- Something that can be called.
- Typically a function.
- Interfaces can inherit from other interfaces.



FunctionOpInterface



- Ops with a symbol and a callable region.
- Leading arguments of the entry block are function arguments.
- Has a function type.

```
func.func @main(%arg0 : tensor<?xf32>) -> tensor<?xf32> {  
    return %arg0 : tensor<?xf32>  
}
```

```
func.func private @foo() -> i32
```

```
llvm.func @malloc(i64) -> !llvm.ptr
```



DestinationStyleOpInterface

DestinationStyleOpInterface

```
+ MutableOperandRange getDpsInitsMutable()
+ OperandRange getDpsInits()
+ int64_t getNumDpsInits()
+ OpOperand *getDpsInitOperand(int64_t i)
+ void setDpsInitOperand(int64_t i, Value value)
+ int64_t getNumDpsInputs()
+ SmallVector<OpOperand*> getDpsInputOperands()
+ SmallVector<Value> getDpsInputs()
+ OpOperand *getDpsInputOperand(int64_t i)
+ bool isDpsInput(OpOperand *opOperand)
+ bool isDpsInit(OpOperand *opOperand)
+ bool isScalar(OpOperand *opOperand)
+ OpResult getTiedOpResult(OpOperand *opOperand)
+ OpOperand *getTiedOpOperand(OpResult opResult)
+ bool hasPureBufferSemantics()
+ bool hasPureTensorSemantics()
```

- Destination-style ops have two kind of operands: **inputs** and **inits**.
- Each init has a “tied” op result. They must have the same type. (Dynamic dimension sizes must match.)
- Used as a “hint” by the other transformations (tiling, bufferization). **Semantics are loosely defined.**

```
%0 = tensor.insert_slice %src into %dest [0][5][1]
      : tensor<5xf32> into tensor<?xf32>
```

```
%1 = linalg.matmul ins(%A, %B: tensor<4x8xf32>, tensor<8x12xf32>)
      outs(%C: tensor<4x12xf32>) -> tensor<4x12xf32>
```

```
%2 = linalg.generic ...
```

Interfaces that Model Control Flow



BranchOpInterface

| BranchOpInterface |
|---|
| + SuccessorOperands getSuccessorOperands (unsigned index)
+ std::optional<BlockArgument> getSuccessorBlockArgument (
unsigned operandIndex)
+ Block * getSuccessorForOperands (ArrayRef<Attribute> operands)
+ bool areTypesCompatible (Type lhs, Type rhs) |

| SuccessorOperands |
|--|
| - unsigned producedOperandCount
- MutableOperandRange forwardedOperands |

`getSuccessorOperands(0) = ???`
`getSuccessorOperands(1) = ???`

- Models control flow between basic blocks within a region.
- Verifies that operand types and block argument types match.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
func.func @foo(%a: i32, %b: i16, %c: i1) {  
  cf.cond_br %c, ^bb1(%b : i16), ^bb2(%a : i32, %a : i32)  
  ^bb1(%arg0: i16):  
    // ...  
  ^bb2(%arg1: i32, %arg2: i32):  
    // ...  
}
```

successors



BranchOpInterface

| BranchOpInterface |
|---|
| + SuccessorOperands getSuccessorOperands (unsigned index)
+ std::optional<BlockArgument> getSuccessorBlockArgument (
unsigned operandIndex)
+ Block * getSuccessorForOperands (ArrayRef<Attribute> operands)
+ bool areTypesCompatible (Type lhs, Type rhs) |

| SuccessorOperands |
|--|
| - unsigned producedOperandCount
- MutableOperandRange forwardedOperands |

```
getSuccessorOperands(0) = [%b]  
getSuccessorOperands(1) = [%a, %a]
```

- Models control flow between basic blocks within a region.
- Verifies that operand types and block argument types match.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
func.func @foo(%a: i32, %b: i16, %c: i1) {  
  cf.cond_br %c, ^bb1(%b : i16), ^bb2(%a : i32, %a : i32)  
  ^bb1(%arg0: i16):  
    // ...  
  ^bb2(%arg1: i32, %arg2: i32):  
    // ...  
}
```

successors



BranchOpInterface

| BranchOpInterface |
|---|
| + SuccessorOperands getSuccessorOperands (unsigned index)
+ std::optional<BlockArgument> getSuccessorBlockArgument (
unsigned operandIndex)
+ Block * getSuccessorForOperands (ArrayRef<Attribute> operands)
+ bool areTypesCompatible (Type lhs, Type rhs) |

| SuccessorOperands |
|--|
| - unsigned producedOperandCount
- MutableOperandRange forwardedOperands |

```
getSuccessorOperands(0) = [%b]  
getSuccessorOperands(1) = [%a, %a]  
getSuccessorBlockArgument(0) = %arg0  
getSuccessorBlockArgument(1) = %arg1  
getSuccessorBlockArgument(2) = %arg2
```

- Models control flow between basic blocks within a region.
- Verifies that operand types and block argument types match.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
func.func @foo(%a: i32, %b: i16, %c: i1) {  
  cf.cond_br %c, ^bb1(%b : i16), ^bb2(%a : i32, %a : i32)  
  ^bb1(%arg0: i16):  
    // ...  
  ^bb2(%arg1: i32, %arg2: i32):  
    // ...  
}
```

successors



RegionBranchOpInterface

RegionBranchOpInterface

```
+ OperandRange getEntrySuccessorOperands(  
    RegionBranchPoint)  
+ void getEntrySuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)  
+ void getSuccessorRegions(  
    RegionBranchPoint,  
    SmallVectorImpl<RegionSuccessor> &)  
+ ValueRange getSuccessorInputs(  
    RegionSuccessor)  
+ bool areTypesCompatible(Type, Type)
```

RegionBranchTerminatorOpInterface

```
+ MutableOperandRange getMutableSuccessorOperands(  
    RegionBranchPoint)  
+ void getSuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)
```

RegionBranchPoint

```
+ bool isParent()  
+ Region *getRegionOrNull()
```

RegionSuccessor

```
+ bool isParent()  
+ Region *getSuccessor()
```

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
%0 = scf.for %iv = %lb to %ub step %step  
    iter_args(%arg0 = %init) -> f32 {  
    %1 = arith.addf %arg0, %arg0 : f32  
    scf.yield %1 : f32  
}
```

```
getSuccessorRegions(BODY) = ???  
getSuccessorRegions(PARENT) = ???
```



RegionBranchOpInterface

RegionBranchOpInterface

```
+ OperandRange getEntrySuccessorOperands(  
    RegionBranchPoint)  
+ void getEntrySuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)  
+ void getSuccessorRegions(  
    RegionBranchPoint,  
    SmallVectorImpl<RegionSuccessor> &)  
+ ValueRange getSuccessorInputs(  
    RegionSuccessor)  
+ bool areTypesCompatible(Type, Type)
```

RegionBranchTerminatorOpInterface

```
+ MutableOperandRange getMutableSuccessorOperands(  
    RegionBranchPoint)  
+ void getSuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)
```

RegionBranchPoint

```
+ bool isParent()  
+ Region *getRegionOrNull()
```

RegionSuccessor

```
+ bool isParent()  
+ Region *getSuccessor()
```

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
%0 = scf.for %iv = %lb to %ub step %step  
    iter_args(%arg0 = %init) -> f32 {  
    %1 = arith.addf %arg0, %arg0 : f32  
    scf.yield %1 : f32  
}
```

```
getSuccessorRegions(BODY) = [BODY, PARENT]  
getSuccessorRegions(PARENT) = [BODY, PARENT]
```



RegionBranchOpInterface

RegionBranchOpInterface

```
+ OperandRange getEntrySuccessorOperands(  
    RegionBranchPoint)  
+ void getEntrySuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)  
+ void getSuccessorRegions(  
    RegionBranchPoint,  
    SmallVectorImpl<RegionSuccessor> &)  
+ ValueRange getSuccessorInputs(  
    RegionSuccessor)  
+ bool areTypesCompatible(Type, Type)
```

RegionBranchTerminatorOpInterface

```
+ MutableOperandRange getMutableSuccessorOperands(  
    RegionBranchPoint)  
+ void getSuccessorRegions(  
    ArrayRef<Attribute> operands,  
    SmallVectorImpl<RegionSuccessor> &)
```

RegionBranchPoint

```
+ bool isParent()  
+ Region *getRegionOrNull()
```

RegionSuccessor

```
+ bool isParent()  
+ Region *getSuccessor()
```

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

```
%0 = scf.for %iv = %lb to %ub step %step  
    iter_args(%arg0 = %init) -> f32 {  
    %1 = arith.addf %arg0, %arg0 : f32  
    scf.yield %1 : f32  
}
```

```
getSuccessorRegions(BODY) = [BODY, PARENT]  
getSuccessorRegions(PARENT) = [BODY, PARENT]  
getSuccessorInputs(BODY) = [%arg0]  
getSuccessorInputs(PARENT) = [%0]  
getEntrySuccessorOperands(BODY) = [%init]  
getEntrySuccessorOperands(PARENT) = [%init]  
getMutableSuccessorOperands(scf.yield) = [%1]
```



RegionBranchOpInterface

| RegionBranchOpInterface |
|--|
| + OperandRange getEntrySuccessorOperands (
RegionBranchPoint) |
| + void getEntrySuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |
| + void getSuccessorRegions (
RegionBranchPoint,
SmallVectorImpl<RegionSuccessor> &) |
| + ValueRange getSuccessorInputs (
RegionSuccessor) |
| + bool areTypesCompatible (Type, Type) |

| RegionBranchTerminatorOpInterface |
|---|
| + MutableOperandRange getMutableSuccessorOperands (
RegionBranchPoint) |
| + void getSuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |

| RegionBranchPoint |
|-----------------------------|
| + bool isParent() |
| + Region *getRegionOrNull() |

| RegionSuccessor |
|--------------------------|
| + bool isParent() |
| + Region *getSuccessor() |

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

before
region

after
region

```
%0, %1 = scf.while (%arg0 = %init) : (f32) -> (f32, f32) {  
  scf.condition(%c) %arg0, %arg0 : f32, f32  
} do {  
  ^bb0(%arg1: f32, %arg2: f32):  
    scf.yield %arg1 : f32  
}
```

```
getSuccessorRegions(PARENT) = ???  
getSuccessorRegions(BEFORE) = ???  
getSuccessorRegions(AFTER) = ???
```



RegionBranchOpInterface

| RegionBranchOpInterface |
|--|
| + OperandRange getEntrySuccessorOperands (
RegionBranchPoint) |
| + void getEntrySuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |
| + void getSuccessorRegions (
RegionBranchPoint,
SmallVectorImpl<RegionSuccessor> &) |
| + ValueRange getSuccessorInputs (
RegionSuccessor) |
| + bool areTypesCompatible (Type, Type) |

| RegionBranchTerminatorOpInterface |
|---|
| + MutableOperandRange getMutableSuccessorOperands (
RegionBranchPoint) |
| + void getSuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |

| RegionBranchPoint |
|-----------------------------|
| + bool isParent() |
| + Region *getRegionOrNull() |

| RegionSuccessor |
|--------------------------|
| + bool isParent() |
| + Region *getSuccessor() |

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

before
region

after
region

```
%0, %1 = scf.while (%arg0 = %init) : (f32) -> (f32, f32) {  
  scf.condition(%c) %arg0, %arg0 : f32, f32  
} do {  
  ^bb0(%arg1: f32, %arg2: f32):  
    scf.yield %arg1 : f32  
}
```

```
getSuccessorRegions(PARENT) = [BEFORE]  
getSuccessorRegions(BEFORE) = [AFTER, PARENT]  
getSuccessorRegions(AFTER) = [BEFORE]
```



RegionBranchOpInterface

| RegionBranchOpInterface |
|--|
| + OperandRange getEntrySuccessorOperands (
RegionBranchPoint) |
| + void getEntrySuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |
| + void getSuccessorRegions (
RegionBranchPoint,
SmallVectorImpl<RegionSuccessor> &) |
| + ValueRange getSuccessorInputs (
RegionSuccessor) |
| + bool areTypesCompatible (Type, Type) |

| RegionBranchTerminatorOpInterface |
|---|
| + MutableOperandRange getMutableSuccessorOperands (
RegionBranchPoint) |
| + void getSuccessorRegions (
ArrayRef<Attribute> operands,
SmallVectorImpl<RegionSuccessor> &) |

| RegionBranchPoint |
|-----------------------------|
| + bool isParent() |
| + Region *getRegionOrNull() |

| RegionSuccessor |
|--------------------------|
| + bool isParent() |
| + Region *getSuccessor() |

- Models control flow between regions.
- Verifies that types matches along control flow edges.
- Queried by dataflow analysis, live analysis, bufferization, ...

before
region

after
region

```
%0, %1 = scf.while (%arg0 = %init) : (f32) -> (f32, f32) {  
  scf.condition(%c) %arg0, %arg0 : f32, f32  
} do {  
  ^bb0(%arg1: f32, %arg2: f32):  
    scf.yield %arg1 : f32  
}
```

```
getSuccessorRegions(PARENT) = [BEFORE]  
getSuccessorRegions(BEFORE) = [AFTER, PARENT]  
getSuccessorRegions(AFTER) = [BEFORE]  
getSuccessorInputs(BEFORE) = [%arg0]  
getSuccessorInputs(AFTER) = [%arg1, %arg2]  
getSuccessorInputs(RESULT) = [%0, %1]  
getEntrySuccessorOperands(BEFORE) = [%init]  
getMutableSuccessorOperands(scf.condition) = [%arg0, %arg0]  
getMutableSuccessorOperands(scf.yield) = [%arg1]
```



LoopLikeOpInterface

| LoopLikeOpInterface |
|--|
| <pre>+ bool isDefinedOutsideOfLoop(Value) + SmallVector<Region *> getLoopRegions() + LogicalResult promoteIfSingleIteration(RewriterBase &) + std::optional<SmallVector<Value>> getLoopInductionVars() + std::optional<SmallVector<OpFoldResult>> getLoopLowerBounds() + std::optional<SmallVector<OpFoldResult>> getLoopUpperBounds() + std::optional<SmallVector<OpFoldResult>> getLoopSteps() + MutableArrayRef<OpOperand> getInitsMutable() + BlockArgListType getRegionIterArgs() + std::optional<MutableArrayRef<OpOperand>> getYieldedValuesMutable() + std::optional<ResultRange> getLoopResults() + FailureOr<LoopLikeOpInterface> replaceWithAdditionalYields(RewriterBase &, ValueRange, ...) + BlockArgument getTiedLoopRegionIterArg(OpOperand *) + BlockArgument getTiedLoopRegionIterArg(OpResult) + OpOperand *getTiedLoopInit(BlockArgument) + OpOperand *getTiedLoopInit(OpResult) + OpOperand *getTiedLoopYieldedValue(BlockArgument) + OpOperand *getTiedLoopYieldedValue(OpResult)</pre> |

- Provides uniform access to LB, UB, step, loop-carried variables and repetitive regions.
- Verifies that number and types of init values, region iter arguments, op results match. (Same as RegionBranchOpInterface, but better error messages.)
- Queried by LICM and Linalg hoisting (replaceWith...).

region iter arg init operand

```
%0 = scf.for %iv = %lb to %ub step %step
      iter_args(%arg0 = %init) -> f32 {
  %1 = arith.addf %arg0, %arg0 : f32
  scf.yield %1 : f32
}
```

results



LoopLikeOpInterface

| LoopLikeOpInterface |
|--|
| <pre>+ bool isDefinedOutsideOfLoop(Value) + SmallVector<Region *> getLoopRegions() + LogicalResult promoteIfSingleIteration(RewriterBase &) + std::optional<SmallVector<Value>> getLoopInductionVars() + std::optional<SmallVector<OpFoldResult>> getLoopLowerBounds() + std::optional<SmallVector<OpFoldResult>> getLoopUpperBounds() + std::optional<SmallVector<OpFoldResult>> getLoopSteps() + MutableArrayRef<OpOperand> getInitsMutable() + BlockArgListType getRegionIterArgs() + std::optional<MutableArrayRef<OpOperand>> getYieldedValuesMutable() + std::optional<ResultRange> getLoopResults() + FailureOr<LoopLikeOpInterface> replaceWithAdditionalYields(RewriterBase &, ValueRange, ...) + BlockArgument getTiedLoopRegionIterArg(OpOperand *) + BlockArgument getTiedLoopRegionIterArg(OpResult) + OpOperand *getTiedLoopInit(BlockArgument) + OpOperand *getTiedLoopInit(OpResult) + OpOperand *getTiedLoopYieldedValue(BlockArgument) + OpOperand *getTiedLoopYieldedValue(OpResult)</pre> |

- Provides uniform access to LB, UB, step, loop-carried variables and repetitive regions.
- Verifies that number and types of init values, region iter arguments, op results match. (Same as RegionBranchOpInterface, but better error messages.)
- Queried by LICM and Linalg hoisting (replaceWith...).

no IV, LB, UB, step

region iter arg

init operand

results

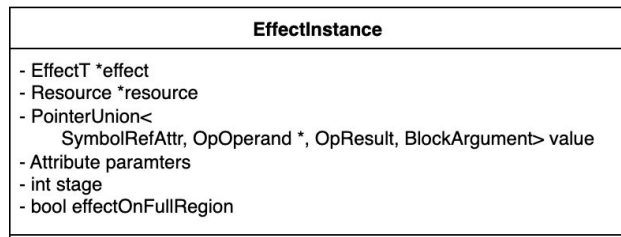
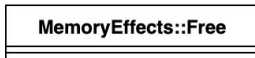
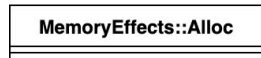
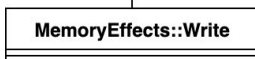
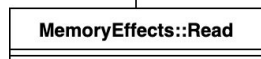
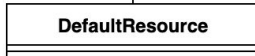
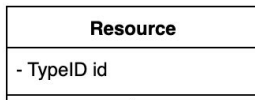
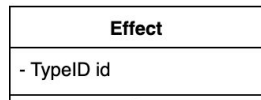
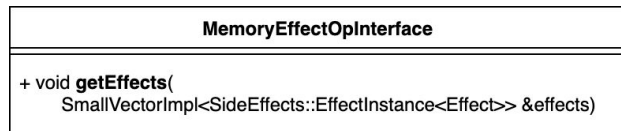
```
%0, %1 = scf.while (%arg0 = %init) : (f32) -> (f32, f32) {
  scf.condition(%c) %arg0, %arg0 : f32, f32
} do {
  ^bb0(%arg1: f32, %arg2: f32):
    scf.yield %arg1 : f32
}
```

does not fit scf.while

Interfaces that Model Operation Semantics



MemoryEffectsOpInterface



```
+ hasUnknownEffects(Operation *op)
+ hasSingleEffect<EffectTy>(Operation *op)
+ hasEffect<EffectTy>(Operation *op)
+ hasSingleEffect<...>(Operation *op, ValueTy val)
+ hasEffect<...>(Operation *op, ValueTy val)
+ mightHaveEffect<EffectTy>(Operation *op)
+ mightHaveEffect<...>(Operation *op, ValueTy val)
+ bool isTriviallyDead(Operation *op)
+ isMemoryEffectFree(Operation *op)
```

- Models **potential** side effects of an operation on a per-operand/result basis.
- Queried by various analyses and transformations such as CSE, hoisting, DCE (greedy rewriter).

```
%0 = memref.load %buffer[%c0, %c0] : memref<?x?xf32>
memref.store %val, %buffer[%c0, %c0] : memref<?x?xf32>
memref.alloc() : memref<128xf32>
vector.print %val : f32
transform.loop.promote_if_one_iteration %loop : !transform.op<"scf.for">
```



MemoryEffectsOpInterface

In C++:

```
void LoadOp::getEffects(SmallVectorImpl<SideEffects::EffectInstance<MemoryEffects::Effect>> &effects) {  
    effects.emplace_back(MemoryEffects::Read::get(), ← effect kind  
                        getMemref(), ← operand/result  
                        SideEffects::DefaultResource::get());  
}
```

In TableGen:

```
def LoadOp : MemRef_Op<"load", ...> {  
    let arguments = (ins Arg<AnyMemRef, "the reference to load from", [MemRead]>:$memref,  
                      Variadic<Index>:$indices);  
}
```



MemoryEffectsOpInterface

- Side effect modelling in MLIR is on an opt-in basis.
- If op does not implement the MemoryEffectsOpInterface, but the RecursiveMemoryEffects trait: side effects are **derived from the ops in the regions** of the operation.
- If op implements neither the MemoryEffectsOpInterface nor the RecursiveMemoryEffects trait: **side effects are unknown**. Analyses / transformations must treat it conservatively.
- NoMemoryEffects TableGen helper.



AlwaysSpeculatable (trait)

- Speculation means executing the operation at an earlier point of time. E.g., hoisting or moving to an earlier place in the IR.
- Typically not implemented by itself. Often implemented together with NoMemoryEffects. **Pure = NoMemoryEffects + AlwaysSpeculatable.**
- Most tensor / vector ops are pure.

```
%0 = tensor.empty() : tensor<5xf32>
```

```
%1 = tensor.extract %t [%pos] : f32 from tensor<128xf32>
```



AlwaysSpeculatable

```
%0 = ... : tensor<*xf32>
>false = arith.constant false
%c5 = arith.constant 5 : index

scf.if %c {
    %sz = tensor.dim %0, %c5 : tensor<*xf32>
    vector.print %sz : index
}
```



AlwaysSpeculatable

```
%0 = ... : tensor<*xf32>
```

```
%false = arith.constant false
```

```
%c5 = arith.constant 5 : index
```

```
scf.if %c {
```

```
    %sz = tensor.dim %0, %c5 : tensor<*xf32>
```

```
    vector.print %sz : index
```

```
}
```



Can you hoist the op?



AlwaysSpeculatable: Counter Example

```
%0 = ... : tensor<*xf32>
```

```
%false = arith.constant false
```

```
%c5 = arith.constant 5 : index
```

```
scf.if %c {
```

```
    %sz = tensor.dim %0, %c5 : tensor<*xf32>
```

```
    vector.print %sz : index
```

```
}
```

Can you hoist the op? No, because the op has UB.

The `tensor.dim` operation takes a tensor and a dimension operand of type `index`. It returns the size of the requested dimension of the given tensor. **If the dimension index is out of bounds, the behavior is undefined.**



ConditionallySpeculatable (interface)

| ConditionallySpeculatable |
|---|
| + Speculatability getSpeculatability() |

| Speculatability |
|--|
| + NotSpeculatable
+ Speculatable
+ RecursivelySpeculatable |

```
Speculation::Speculatability DimOp::getSpeculatability() {  
    auto constantIndex = getConstantIndex();  
    if (!constantIndex)  
        return Speculation::NotSpeculatable;  
  
    auto rankedSourceType = dyn_cast<RankedTensorType>(getSource().getType());  
    if (!rankedSourceType)  
        return Speculation::NotSpeculatable;  
  
    if (rankedSourceType.getRank() <= constantIndex)  
        return Speculation::NotSpeculatable;  
  
    return Speculation::Speculatable;  
}
```



speculatable if index is guaranteed to be in-bounds

AlwaysSpeculatable / ConditionallySpeculatable

- AlwaysSpeculatable (and Pure) is a nice property. It enables many transformations and makes some analyses simpler.
- Undefined Behavior (UB) is a big hammer: it prevents any speculation. (And the program can no longer continue executing in a meaningful way.)
- Best practice (?): Give your operations poison semantics instead of UB.
 - Poison = Deferred undefined behavior. Propagated by most ops (poison in $\Rightarrow$ poison out), but some ops turn it into immediate undefined behavior (e.g., `cf.cond_br`, memory access ptr).
 - Should `tensor.dim` be pure and have poison semantics?
 - Many ops (`tensor.extract`, `tensor.insert`, `tensor.extract_slice`, ...) are underspecified. They are pure, but their out-of-bounds semantics are not specified.

Interfaces that Drive Analyses



InferIntRangeInterface

| InferIntRangeInterface |
|--|
| + inferResultRanges(
ArrayRef<ConstantIntRanges> argRanges,
SetIntRangeFn setResultRanges) |

| ConstantIntRanges |
|--|
| - APInt uminVal
- APInt umaxVal
- APInt sminVal
- APInt smaxVal |
| + ConstantIntRanges rangeUnion(other)
+ ConstantIntRanges intersection(other) |

- Given the integer range of the operands, infer the range of the results.
- Models both signed and unsigned integer semantics.
- Queried by IntegerRangeAnalysis (data flow analysis).
- arith-int-range-narrowing: Pass that reduces the bitwidth of integer SSA values (to make the computation faster, use less memory).

```
func.func @test(%arg0: i8) {                                // range(%arg0) = [-128, 127]
  %0 = arith.constant 5 : i8                                // range(%0)   = [5, 5]
  %1 = arith.addi %arg0, %0 overflow<nsw> : i8              // range(%1)   = [-123, 127]
  ...
}
```



ValueBoundsOpInterface

- Populates a constraint set with lower/upper/equality bounds of integer-typed SSA values and/or dimension sizes of shaped values.
- Ops that implement DestinationStyleOpInterface are supported out of the box.
- Bounds can be expressed in terms of other SSA values.

```
%c1 = arith.constant 1 : i8           // %c1 = 1
%0 = arith.addi %a, %c1 : i8           // %a = %c1 + 1
%2 = arith.select %c, %0, %1 : i8      // %a <= %0 <= %a + 1
```

ValueBoundsOpInterface

```
+ void populateBoundsForIndexValue(
    Value value,
    ValueBoundsConstraintSet &cstr)
+ void populateBoundsForShapedValueDim(
    Value value, int64_t dim,
    ValueBoundsConstraintSet &cstr)
```

ValueBoundsConstraintSet::Variable

```
- AffineMap map
- ValueDimList mapOperand
```

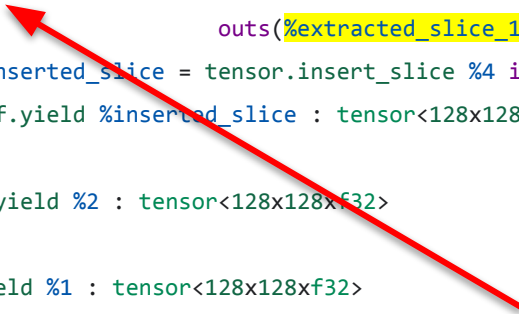
ValueBoundsConstraintSet

```
- FlatLinearConstraint cstr
- StopConditionFn stopCondition

+ LogicalResult computeBound(
    AffineMap &resultMap,
    ValueDimList &resultOperands,
    BoundType type,
    const Variable &var,
    StopConditionFn stopCondition)
+ LogicalResult computeDependentBound(
    ..., ValueRange dependencies)
+ LogicalResult computeIndependentBound(
    ..., ValueRange independencies)
+ FailureOr<int64_t> computeConstantBound(...)
+ bool compare(Variable lhs,
    ComparisonOperator cmp, Variable rhs)
+ FailureOr<bool> areOverlappingSlices(
    HyperrectangularSlice slice1,
    HyperrectangularSlice slice2)
+ FailureOr<bool> areEquivalentSlices(...)
```

Ex.: Matmul Tiling [4, 9, 4] – Rediscover Static Information

```
func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>, %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {  
  %0 = scf.for %arg3 = %c0 to %c128 step %c4 iter_args(%arg4 = %arg2) -> (tensor<128x128xf32>) {  
    %1 = scf.for %arg5 = %c0 to %c128 step %c9 iter_args(%arg6 = %arg4) -> (tensor<128x128xf32>) {  
      %2 = scf.for %arg7 = %c0 to %c128 step %c4 iter_args(%arg8 = %arg6) -> (tensor<128x128xf32>) {  
        %3 = affine.min affine_map<(d0) -> (-d0 + 128, 9)>(%arg5)  
        %extracted_slice = tensor.extract_slice %arg0[%arg3, %arg7] [4, 4] [1, 1] : tensor<128x128xf32> to tensor<4x4xf32>  
        %extracted_slice_0 = tensor.extract_slice %arg1[%arg7, %arg5] [4, %3] [1, 1] : tensor<128x128xf32> to tensor<4x?xf32>  
        %extracted_slice_1 = tensor.extract_slice %arg8[%arg3, %arg5] [4, %3] [1, 1] : tensor<128x128xf32> to tensor<4x?xf32>  
        %4 = linalg.matmul ins(%extracted_slice, %extracted_slice_0 : tensor<4x4xf32>, tensor<4x?xf32>)  
                           outs(%extracted_slice_1 : tensor<4x?xf32>) -> tensor<4x?xf32>  
        %inserted_slice = tensor.insert_slice %4 into %arg8[%arg3, %arg5] [4, %3] [1, 1] : tensor<4x?xf32> into tensor<128x128xf32>  
        scf.yield %inserted_slice : tensor<128x128xf32>  
      }  
      scf.yield %2 : tensor<128x128xf32>  
    }  
    scf.yield %1 : tensor<128x128xf32>  
  }  
  return %0 : tensor<128x128xf32>  
}
```



compute constant UB for dim(%4, 1) → 10 (open bound)

Ex.: Matmul Tiling [4, 9, 4] – Rediscover Static Information

```
func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>, %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {
  %0 = scf.for %arg3 = %c0 to %c128 step %c4 iter_args(%arg4 = %arg2) -> (tensor<128x128xf32>) {
    %1 = scf.for %arg5 = %c0 to %c128 step %c9 iter_args(%arg6 = %arg4) -> (tensor<128x128xf32>) {
      %2 = scf.for %arg7 = %c0 to %c128 step %c4 iter_args(%arg8 = %arg6) -> (tensor<128x128xf32>) {
        %3 = affine.min affine_map<(d0) -> (-d0 + 128, 9)>(%arg5)
        %extracted_slice = tensor.extract_slice %arg0[%arg3, %arg7] [4, 4] [1, 1] : tensor<128x128xf32> to tensor<4x4xf32>
        %extracted_slice_0 = tensor.extract_slice %arg1[%arg7, %arg5] [4, %3] [1, 1] : tensor<128x128xf32> to tensor<4x?xf32>
        %extracted_slice_1 = tensor.extract_slice %arg8[%arg3, %arg5] [4, %3] [1, 1] : tensor<128x128xf32> to tensor<4x?xf32>
        %4 = linalg.matmul ins(%extracted_slice, %extracted_slice_0 : tensor<4x4xf32>, tensor<4x?xf32>)
        outs(%extracted_slice_1 : tensor<4x?xf32>)
        %inserted_slice = tensor.insert_slice %4 into %arg8
        scf.yield %inserted_slice : tensor<128x128xf32>
      }
    }
    scf.yield %2 : tensor<128x128xf32>
  }
  scf.yield %1 : tensor<128x128xf32>
}
return %0 : tensor<128x128xf32>
}
```

Constraint set: 4 variables

| (%4 | %extracted_slice_1 | %3 | %arg7 | const) | |
|-----|--------------------|----|-------|--------|------|
| 1 | -1 | 0 | 0 | 0 | = 0 |
| 0 | 1 | -1 | 0 | 0 | = 0 |
| 0 | 0 | -1 | -1 | 128 | >= 0 |
| 0 | 0 | -1 | 0 | 9 | >= 0 |

compute constant UB for dim(%4, 1) → 10 (open bound)

FlatLinearConstraints (MLIR Presburger library)



$$Ax + b = 0$$

$$Ax + b \geq 0$$

one variable per SSA value

coefficients stored as a matrix

- Linear combination of variables
- Multiplication/division of variables is not supported
- Corresponds to the “flattened form” of AffineExprs
- Relevant API:
 - project out a variable (Fourier-Motzkin elimination)
 - compute LB/UB of a variable
 - check if constraint set is “empty”



ValueBoundsOpInterface

```
struct AddIOpInterface
{
    public ValueBoundsOpInterface::ExternalModel<AddIOpInterface, AddIOp> {
void populateBoundsForIndexValue(Operation *op, Value value,
                                ValueBoundsConstraintSet &cstr) const {
    auto addIOp = cast<AddIOp>(op);
    assert(value == addIOp.getResult() && "invalid value");

    cstr.bound(value) == cstr.getExpr(addIOp.getLhs()) + cstr.getExpr(addIOp.getRhs());
}
};
```

op result or block argument

get affine expr for SSA value

affine expression, constant or SSA value

comparison operators(==, <, <=, >=, >) are overloaded

not a (dataflow) analysis!

Computing Bounds: Worklist-Driven IR Analysis

push to worklist + add variable

computeBound(value)



FIFO worklist

project out all values (variables) for which stop_condition does not hold

compute bound

materialize bound in IR

$v := \text{pop from worklist}$

stop_condition(v)?

true

false

populate constraints for v

push to worklist + add variable
if seen for the first time

cmp operator
adds (in)equality
to constraint set

`cstr.bound(v) == cstr.getExpr(operand)`

ValueBoundsOpInterface::populateBoundsForIndexValue



ValueBoundsOpInterface Limitations

- Cannot model overflows: integers have infinite precision (`DynamicAPIInt`).
- Non-linear constraints (e.g., multiplying two SSA values) cannot be modelled.
- Cannot distinguish between signed/unsigned integer semantics.
- Performance issues: A brand new constraint set is built for each query.
(Entire IR is traversed for each query, until the stop condition is reached.)



ReifyRankedShapedTypeOpInterface

| ReifyRankedShapedTypeOpInterface |
|---|
| + LogicalResult reifyResultShapes (
OpBuilder &builder,
SmallVector<SmallVector<OpFoldResult>> &results) |

- Materialize (dynamic) dimension sizes as SSA values.
- Simplifies memref/tensor.dim ops:
-resolve-ranked-shaped-type-result-dims

```
%0 = tensor.pad %arg0 low[2, %arg1, 3, %arg1] high[3, 3, %arg1, 2] {  
  ^bb0(%arg2: index, %arg3: index, %arg4: index, %arg5: index):  
    tensor.yield %pad_value : f32  
} : tensor<1x2x2x?xf32> to tensor<6x?x?x?xf32>
```



```
%c6 = arith.constant 6 : index  
%c2 = arith.constant 2 : index  
%c3 = arith.constant 3 : index  
%0 = arith.addi %c2, %arg1 : index  
%1 = arith.addi %0, %c3 : index  
%dim = tensor.dim %arg0, %c3 : tensor<1x2x2x?xf32>  
%2 = arith.addi %dim, %0 : index  
  
⇒ [ %c6, %1, %1, %2 ]
```



ReifyRankedShapedTypeOpInterface

- Can this interface be replaced by ValueBoundsOpInterface?
- ValueBoundsConstraintSet can be queried to compute an equality bound for an op result %0 in terms of other SSA values.
- Certain arithmetic operations cannot be represented in the constraint set:
%0 = arith.muli %a, %b : i32
- Current ValueBoundsConstraintSet may be too inefficient.

Interfaces that Drive Transformations



BufferizableOpInterface

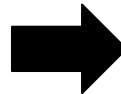
| BufferizableOpInterface |
|---|
| + bool bufferizesToAllocation (Value)
+ bool bufferizesToMemoryRead (OpOperand &)
+ bool bufferizesToMemoryWrite (OpOperand &)
+ bool bufferizesToElementwiseAccess (...)
+ bool mustBufferizeInPlace (OpOperand &)
+ AliasingValueList getAliasingValues (OpOperand &)
+ AliasingOpOperandList getAliasingOpOperands (Value value)
+ LogicalResult bufferize (RewriterBase &, const BufferizationOptions &)
+ bool isWritable (Value)
+ bool isNotConflicting (OpOperand *uRead, OpOperand *uWrite)
+ LogicalResult verifyAnalysis ()
+ FailureOr<BufferLikeType> getBufferType (Value)
+ bool isRepetitiveRegion (unsigned idx)
+ bool isParallelRegion (unsigned idx) |

| AliasingOpOperand |
|--|
| - OpOperand *operand
- BufferRelation relation
- bool isDefinite |

| AliasingValue |
|---|
| - Value value
- BufferRelation relation
- bool isDefinite |

- -one-shot-bufferize converts tensor IR to memref IR.
- Tensors are “abstract” values that do not have memory locations. Memrefs are pointers + metadata (shape, etc.).
- Bufferization typically happens towards the end of a compilation pipeline.

```
func.func @test(%a: tensor<5xf32>)  
  -> tensor<5xf32> {  
    %cst = arith.constant 0.0 : f32  
    %0 = linalg.fill ins(%cst) outs(%a)  
    -> tensor<5xf32>  
    %r = tensor.extract %a[0] : tensor<5xf32>  
    vector.print %r : f32  
    return %0 : tensor<5xf32>  
  }
```



```
func.func @test(%a: memref<5xf32>)  
  -> memref<5xf32> {  
    %cst = arith.constant 0.0 : f32  
    %copy = memref.alloc() : memref<5xf32>  
    memref.copy %a, %copy  
    linalg.fill ins(%cst) outs(%copy)  
    %r = memref.load %a[0] : memref<5xf32>  
    vector.print %r : f32  
    return %copy : memref<5xf32>  
  }
```



History of One-Shot Bufferize

- One-Shot Bufferize used to be a monolithic component, with a hard-coded list of supported operations. (Living in Linalg/Transforms.)
- Today, One-Shot Bufferize is [interface-based](#), extensible and interface implementations live in their respective dialects.

```
LogicalResult mlir::linalg::bufferizeOp(
    Operation *op, BlockAndValueMapping &bvm, BufferizationAliasInfo &aliasInfo,
    AllocationCallbacks allocationFns,
    DenseMap<FuncOp, FunctionType> *bufferizedFunctionTypes) {
    OpBuilder b(op->getContext());
    return TypeSwitch<Operation *, LogicalResult>(op)
        // Skip BufferCast and TensorLoad ops.
        .Case<memref::BufferCastOp, memref::TensorLoadOp>(<
            [&](auto) { return success(); })
        .Case<ExtractSliceOp, InitTensorOp, InsertSliceOp, LinalgOp, scf::ForOp,
            scf::IfOp, tensor::CastOp, TiledLoopOp, VectorTransferOpInterface>(<
            [&](auto op) {
                LDBG("Begin bufferize:\n" << op << '\n');
                return bufferize(b, op, bvm, aliasInfo, allocationFns);
            })
        .Case<tensor::DimOp, tensor::ExtractOp, ReturnOp, linalg::YieldOp,
            scf::YieldOp>([&](auto op) {
            LDBG("Begin bufferize:\n" << op << '\n');
            return bufferize(b, op, bvm, aliasInfo);
        })
        .Case([&](CallOpInterface op) {
            LDBG("Begin bufferize:\n" << op << '\n');
            if (!bufferizedFunctionTypes)
```



BufferizableOpInterface

- Models future memory read/write side effects:
 - `bufferizesToMemoryRead`: does the bufferized op have a “read” side effect?
 - `bufferizesToMemoryWrite`: does the bufferized op have a “write” side effect?
 - Cannot reason about tensor subsets yet.
- Models future alias sets:
 - `getAliasingOpOperand`: can a given op result share the same buffer with an operand?
 - `getAliasingOpResult`: can a given operand share the same buffer with an op result?
 - `%0 = tensor.insert %f into %t[0] : tensor<5xf32>`
 - `%t` and `%0` can share the same buffer. (In the absence of a RaW conflict.)
 - Can bufferize to: `memref.store %f, buffer(%t) : memref<5xf32>`
- Interface function that actually bufferizes the IR.

default implementation for `DestinationStyleOpInterface`



Read-after-Write (RAW) Conflict

// Definition

```
%1 = linalg.fill ins(%cst1) outs(...) -> tensor<5xf32>
```

// Conflicting Write → Cannot reuse *buffer(%1)* because we still need the value of %1.

```
%2 = linalg.fill ins(%cst2) outs(%1) -> tensor<5xf32>
```

aliasingOpOperand(%2) = %1

⇒ Try to reuse *buffer(%2) = buffer(%1)*

bufferizes to mem write

// Read

```
%3 = tensor.extract %1[%c0] : tensor<5xf32>
```

bufferizes to mem read

Bufferization of tensor.extract_slice / insert_slice

`%t = ... // definition` alias set: {%t, %0, %1, %2}

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

`%1 = linalg.fill ins(%cst) outs(%0) -> tensor<5xf32>`

bufferizes to mem write

`%2 = tensor.insert_slice %1 into %t [0][5][1] : tensor<5xf32> into tensor<10xf32>`

bufferizes to mem read

CONFLICT?

Bufferization of `tensor.extract_slice` / `insert_slice`

`%t = ... // definition` alias set: {`%t`, `%0`, `%1`, `%2`}

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

`%1 = linalg.fill ins(%cst) outs(%0) -> tensor<5xf32>`

bufferizes to mem write

`%2 = tensor.insert_slice %1 into %t [0][5][1] : tensor<5xf32> into tensor<10xf32>`

bufferizes to mem read

No, because `tensor.insert_slice` reads only the part of `%t` that is not written by `linalg.fill`. (Tensor subsets are *non-overlapping*.)

Bufferization of tensor.gather / scatter (not implemented)

```
%t = ... // definition
```

```
%0 = tensor.gather %t ...
```

alias set: {%t, %0, %1, %2}

```
%1 = linalg.fill ins(%cst) outs(%0) -> tensor<5xf32>
```

bufferizes to mem write

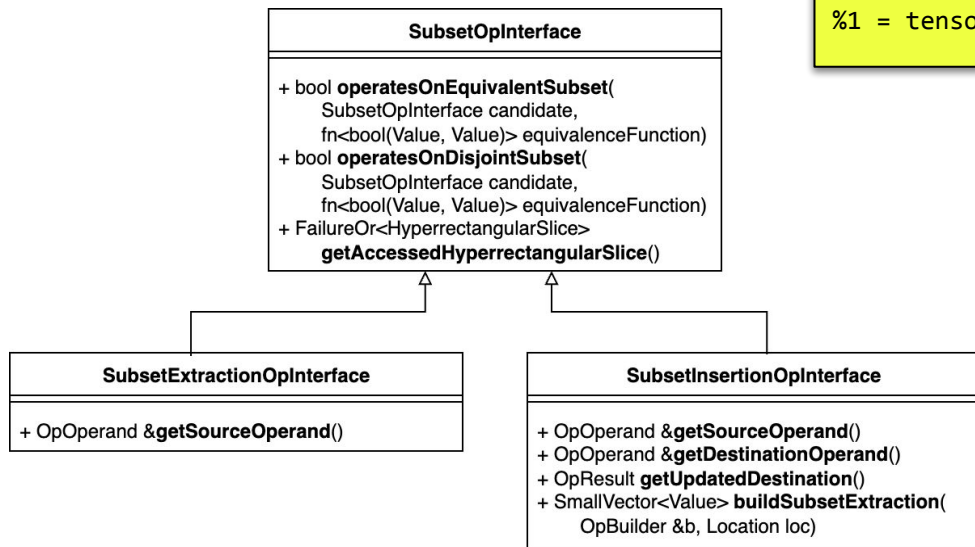
```
%2 = tensor.scatter %1 into %t ...
```

bufferizes to mem read

CONFLICT?



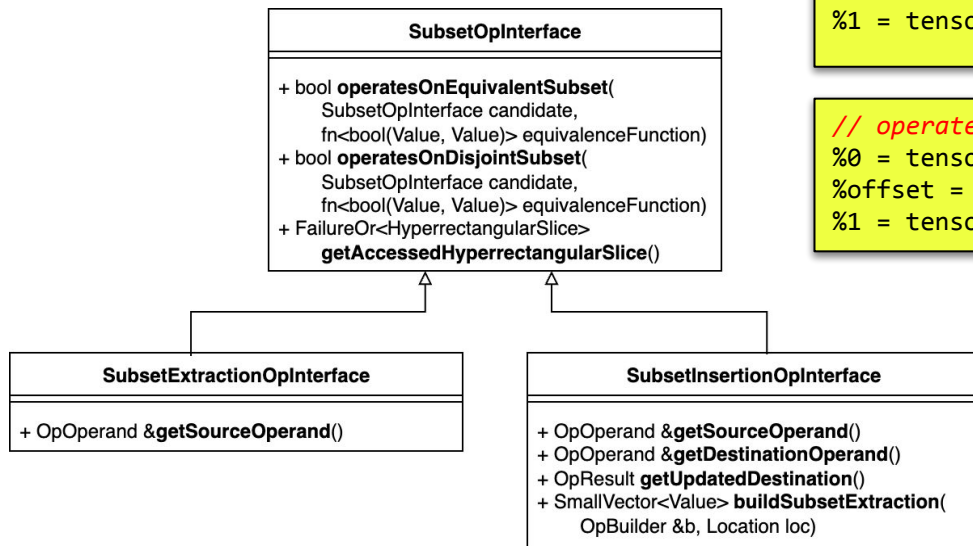
SubsetInsertion / ExtractionOpInterface



```
// operatesOnEquivalentSubset ⇒ true
%0 = tensor.extract_slice %t [9][5][1] : ...
%1 = tensor.insert_slice ... into %t [9][5][1] : ...
```



SubsetInsertion / ExtractionOpInterface

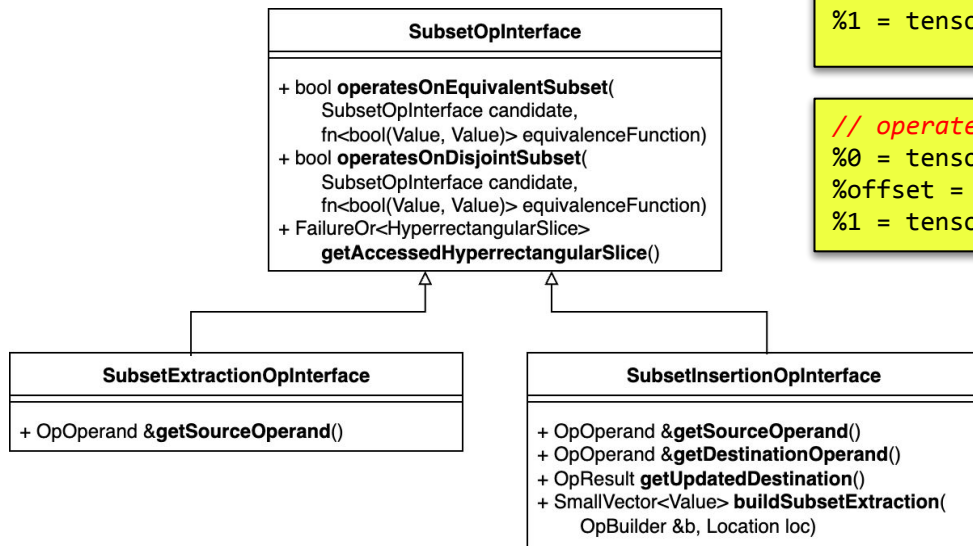


```
// operatesOnEquivalentSubset ⇒ true
%0 = tensor.extract_slice %t [9][5][1] : ...
%1 = tensor.insert_slice ... into %t [9][5][1] : ...
```

```
// operatesOnEquivalentSubset ⇒ ???
%0 = tensor.extract_slice %t [9][5][1] : ...
%offset = arith.addi %c4, %c5 : index
%1 = tensor.insert_slice ... into %t [%offset][5][1] : ...
```



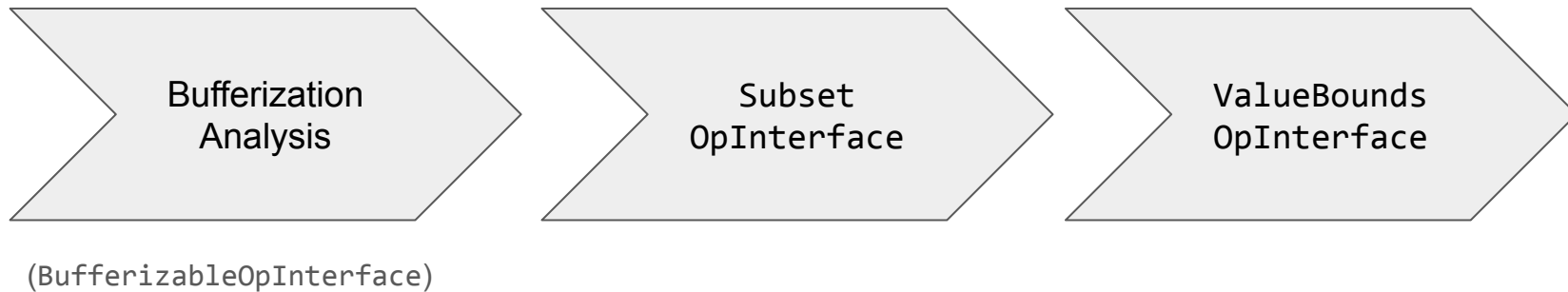
SubsetInsertion / ExtractionOpInterface



```
// operatesOnEquivalentSubset ⇒ true
%0 = tensor.extract_slice %t [9][5][1] : ...
%1 = tensor.insert_slice ... into %t [9][5][1] : ...
```

```
// operatesOnEquivalentSubset ⇒ true
%0 = tensor.extract_slice %t [9][5][1] : ...
%offset = arith.addi %c4, %c5 : index
%1 = tensor.insert_slice ... into %t [%offset][5][1] : ...
```

operatesOnEquivalentSubset queries
ValueBoundsConstraintSet
::areEquivalentSlices(...)





TilingInterface

TilingInterface

+ SmallVector<IteratorType> **getLoopIteratorTypes()**
 + SmallVector<Range> **getIterationDomain**(OpBuilder &)
 + FailureOr<TilingResult> **getTiledImplementation**(
 OpBuilder &, ArrayRef<OpFoldResult> offsets,
 ArrayRef<OpFoldResult> sizes)
 + LogicalResult **getResultTilePosition**(...)
 + FailureOr<TilingResult> **generateResultTileValue**(...)
 + FailureOr<TilingResult> **getTiledImplementationFromOperandTiles**(...)
 + LogicalResult **getIterationDomainTileFromOperandTiles**(...)
 + LogicalResult **getIterationDomainTileFromResultTile**(...)
 + LogicalResult **generateScalarImplementation**(...)

TilingResult

- SmallVector<Operation *> tiledOps
 - SmallVector<Value> tiledValues
 - SmallVector<Operation *> generatedSlices

+ FailureOr<TiledLinalgOp> **tileLinalgOp**(
 RewriterBase &, LinalgOp, const LinalgTilingOptions &)
 + FailureOr<SCFTilingResult> **tileUsingSCF**(
 RewriterBase &, TilingInterface op, const SCFTilingOptions &)

- Used by the Linalg tiling infrastructure to tile an operation with tensor semantics.
- Best understood by studying MLIR test cases.

```
// CHECK: %[[TD0:.]] = scf.for {{.*}} to {{.*}} step {{.*}} iter_args(%[[TC0:.]] = %[[TC]]) ... {
// CHECK:   %[[TD1:.]] = scf.for {{.*}} to {{.*}} step {{.*}} iter_args(%[[TC1:.]] = %[[TC0]]) ... {
// CHECK:     %[[TD2:.]] = scf.for {{.*}} to {{.*}} step {{.*}} iter_args(%[[TC2:.]] = %[[TC1]]) ... {
// CHECK:       %[[sTA:.]] = tensor.extract_slice %[[TA]][{{.*}}]
// CHECK:       %[[sTB:.]] = tensor.extract_slice %[[TB]][{{.*}}]
// CHECK:       %[[sTC:.]] = tensor.extract_slice %[[TC2]][{{.*}}]
// CHECK:       %[[sTD:.]] = linalg.matmul ins(%[[sTA]], %[[sTB]] : tensor<4x4xf32>, tensor<4x4xf32>)
// CHECK-SAME:                                outs(%[[sTC]] : tensor<4x4xf32>) -> tensor<4x4xf32>
// CHECK:       %[[TD:.]] = tensor.insert_slice %[[sTD]] into %[[TC2]][{{.*}}]
// CHECK:       scf.yield %[[TD]] : tensor<128x128xf32>
// CHECK:       scf.yield %[[TD2]] : tensor<128x128xf32>
// CHECK:       scf.yield %[[TD1]] : tensor<128x128xf32>
%0 = linalg.matmul ins(%arg0, %arg1: tensor<128x128xf32>, tensor<128x128xf32>)
outs(%arg2: tensor<128x128xf32>) -> tensor<128x128xf32>
```



TransformOpInterface

TransformOpInterface

+ DiagnosedSilencableFailure **apply**(
TransformRewriter &,
TransformResults &,
TransformState &)

TransformState

- MapVector<Region *, std::unique_ptr<Mappings>>
mappings
- InvalidatedHandleMap invalidatedHandles

- Implemented by transform dialect ops.
- `apply()` executes the transform dialect op.

```
module attributes {transform.with_named_sequence} {  
  transform.named_sequence @__transform_main(  
    %arg1: !transform.any_op {transform.readonly}) {  
    %0 = transform.structured.match ops{["linalg.matmul"]} in %arg1  
    : (!transform.any_op) -> !transform.any_op  
    %1, %loops:3 = transform.structured.tile_using_for %0  
    tile_sizes [4, 4, 4]  
    : (!transform.any_op)  
    -> (!transform.any_op, !transform.any_op,  
        !transform.any_op, !transform.any_op)  
    transform.yield  
  }  
}
```

Type Interfaces



FloatType

| FloatType |
|--|
| + llvm::fltSemantics &getSemantics()
+ FloatType scaleElementBitwidth()
+ unsigned getWidth()
+ unsigned getFPManitssaWidth() |

| fltSemantics |
|---|
| - ExponentType maxExponent
- ExponentType minExponent
- unsigned precision
- unsigned sizeInBits
- fltNonfiniteBehavior nonFiniteBehavior
- fltNanEncoding nanEncoding
- bool hasZero
- bool hasSignedRepr
- bool hasSignBitInMSB |

| |
|--|
| fltSemantics &IEEEhalf()
fltSemantics &BFloat()
fltSemantics &IEEEsingle()
fltSemantics &IEEEdouble()
fltSemantics &IEEEquad()
fltSemantics &PPCDoubleDouble()
fltSemantics &PPCDoubleDoubleLegacy()
fltSemantics &Float8E5M2()
fltSemantics &Float8E5M2FNUZ()
fltSemantics &Float8E4M3()
fltSemantics &Float8E4M3FN()
fltSemantics &Float8E4M3FNUZ()
fltSemantics &Float8E4M3B11FNUZ()
fltSemantics &Float8E3M4()
fltSemantics &FloatTF32()
fltSemantics &Float8E8M0FNU()
fltSemantics &Float6E3M2FN()
fltSemantics &Float6E2M3FN()
fltSemantics &Float4E2M1FN()
fltSemantics &x87DoubleExtended() |
|--|

| APFloat |
|---|
| - const fltSemantics *sem
- union(IEEEFloat, DoubleAPFloat) storage |
| + opStatus add(const APFloat &, roundingMode)
+ opStatus subtract(const APFloat &, roundingMode)
+ opStatus multiply(const APFloat &, roundingMode)
+ opStatus divide(const APFloat &, roundingMode)
+ opStatus remainder(const APFloat &)
+ opStatus mod(const APFloat &)
+ opStatus fusedMultiplyAdd(const APFloat &, const APFloat & roundingMode)
+ opStatus roundToIntegral(roundingMode)
+ void changeSign()
+ void copySign(const APFloat &)
+ APFloat makeQuiet()
+ opStatus convert(const fltSemantics &, roundingMode, bool *losesInfo)
+ double convertToDouble()
+ opStatus convertToInteger(APSInt &Result, roundingMode, bool *IsExact)
... |

f32
f64
f8E4M3FN

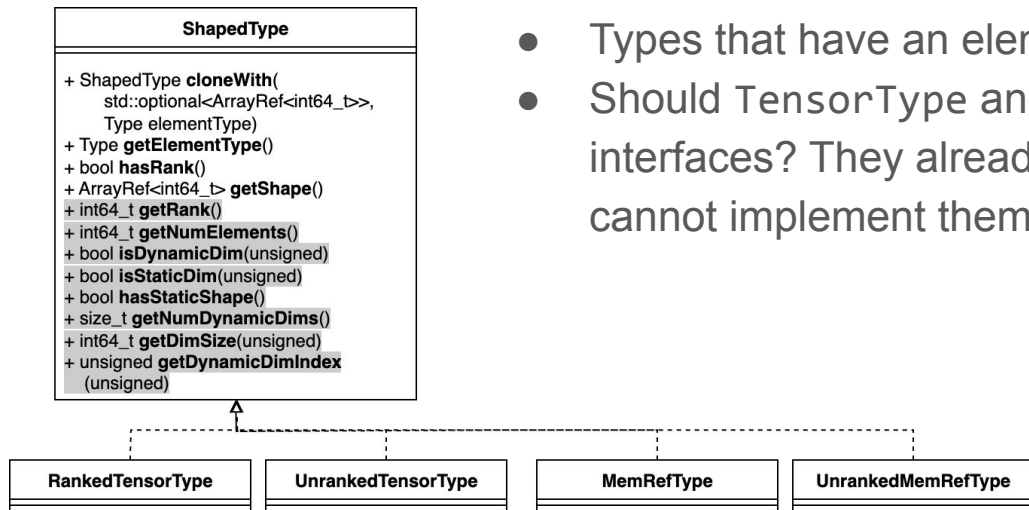
- FP types are defined in LLVM (fltSemantics; code is a mess, needs rewrite...).
- FP values are represented as APFloat.
- MLIR types / attributes are just a wrapper around these LLVM objects.

IntegerType Interface?

- Does not exist today. Has been requested a few times by OSS folks.
 - Define your own integer type, while being able to reuse `arith` dialect ops.
- Example: Integer type that carries divisibility information. Does not compose well with `arith` operations (`AllTypesMatch`).



ShapedType



- Types that have an element type and **may have** a shape.
- Should TensorType and BaseMemRefType be public interfaces? They already behave like interfaces, but you cannot implement them on new types.

```
tensor<10x20xf32>
tensor<*xf32>
memref<10x20xf32>
memref<*xf32>
```

Attribute Interfaces



TypedAttrInterface

| TypedAttrInterface |
|-------------------------------|
| + Type <code>getType()</code> |

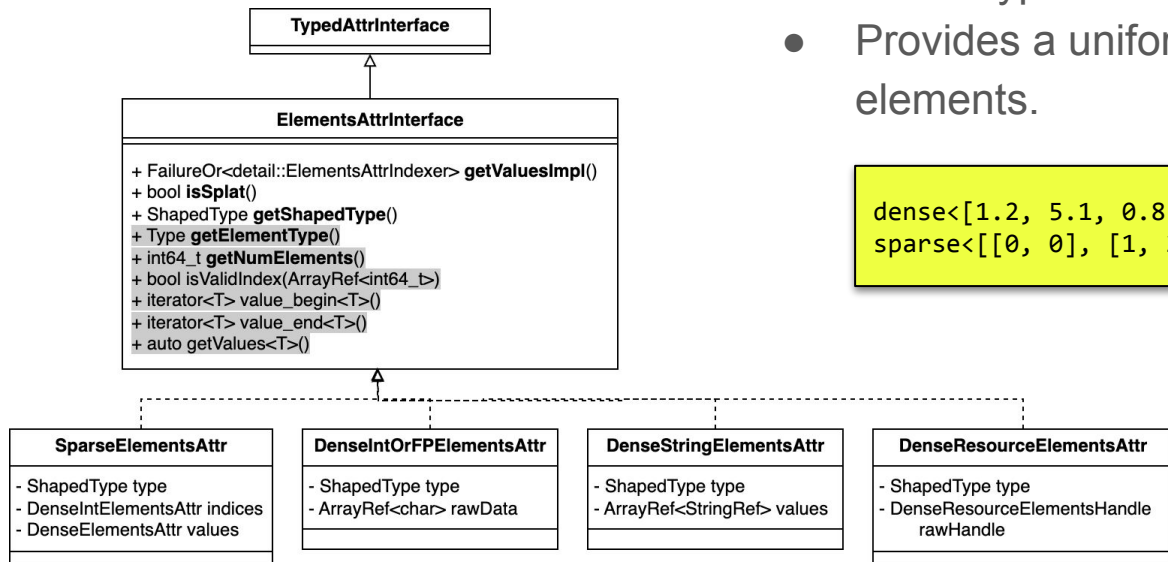
- Query the type of an attribute in a uniform way.
- Examples: FloatAttr, IntegerAttr

```
2.1 : f32
127 : i8
dense<[1.2, 5.1, 0.8]> : tensor<3xf16>
```



ElementsAttrInterface

- Abstracts over constant elements with a vector or tensor type.
- Provides a uniform C++ API to iterate / query elements.



```
dense<[1.2, 5.1, 0.8]> : tensor<3xf16>
sparse<[[0, 0], [1, 2]], [1, 5]> : tensor<3x4xi32>
```

Dialect Interfaces



BytecodeDialectInterface

| BytecodeDialectInterface |
|---|
| <ul style="list-style-type: none">+ Attribute readAttribute(
DialectBytecodeReader &)+ Type readType(
DialectBytecodeReader &)+ LogicalResult writeAttribute(
Attribute, DialectBytecodeWriter &)+ LogicalResult writeType(
Type, DialectBytecodeWriter &)+ void writeVersion(
DialectBytecodeWriter &)+ std::unique_ptr<DialectVersion> readVersion(
DialectBytecodeReader &)+ LogicalResult upgradeFromVersion(
Operation *, const DialectVersion &) |

- Bytecode is a more space-efficient storage of IR.
- Every dialect has a version.
- Dialects provide hooks to read/write attributes/types.
- Dialects provide hooks to upgrade bytecode of an older version to a newer version.

Open Questions / Discussion



Interface Promise breaks Modularity

- Dialect compilation unit must depend on all compilation units that contain implemented interfaces. Even those that are implemented via external model.
 - Example: `TensorDialect` must depend on `BufferizationDialect` to promise the interface.
- An interface defined outside of the MLIRIR cannot be promised for `builtin` operations / types / attributes / dialect and requires hard-coding the op.
 - Example: `builtin.module` cannot implement `DataLayoutOpInterface`.
 - Example: `builtin` dialect cannot implement `Inliner` dialect interface.
 - Example: `OpAsmOpInterface`, `SymbolOpInterface` cannot be defined in `MLIRInterfaces`.
- We are giving preferential treatment to upstream interfaces. Downstream interface do not have to be (and cannot be) promised.



Swappable Interface Implementations

- External interface registrations are tracked in the MLIR context (`OperationName::Impl::interfaceMap`).
- OSS folks occasionally ask for customizable op interfaces, i.e.:
 - A hook to unregister an interface: remove the interface from the interface map.
 - Allow registration of another external model.
 - Example: Customizable BufferizableOpInterface implementation.



Other Interfaces

CastOpInterface
SelectLikeOpInterface
ReturnLike
DataLayoutOpInterface
DialectFoldInterface
InferTypeOpInterface
InferShapedTypeOpInterface
BlobAttrInterface
DestinationStyleOpInterface
HasParallelRegion
ParallelCombiningOpInterface
RuntimeVerifiableOpInterface
VectorUnrollOpInterface
VectorTransferOpInterface
DerivedAttributeOpInterface

PromotableAllocationOpInterface
PromotableMemOpInterface
PromotableOpInterface
DestructurableAllocationOpInterface
Symbol
SymbolUserOpInterface
BytecodeOpInterface
VectorElementTypeInterface
IndexingMapOpInterface
MemRefElementTypeInterface
PtrLikeTypeInterface
BlobAttrInterface
...