Google Research

Computing Bounds of SSA Values in MLIR

Matthias Springer
Google Research, Switzerland

EuroLLVM 2024 – April 11, 2024

Introduction

- Compute LB/UB/EQ of index-typed SSA values or (dynamic) dimension sizes of shaped values (tensor/memref).
- Compare two index-typed SSA values or dimension sizes.
- Op interface driven: `ValueBoundsOpInterface`
- Built on top of the MLIR Presburger library.
- Use cases (examples):
  - *Allocation Hoisting*: Compute an upper bound for a dynamic memory allocation size.
  - *Enable Vectorization*: Compute an upper bound of a dynamically-shaped tensor computation.
  - *Subset-based Programming / Bufferization / etc*:
    Prove that two slices/subviews into the same tensor/memref are equivalent/non-overlapping.

Public API by Example

(still in review: #87980)

Compare Values / Dimensions

```cpp
#include "mlir/Interfaces/ValueBoundsOpInterface.h"

/// Return "true" if "lhs cmp rhs" was proven to hold. Return "false" if the
/// specified relation could not be proven. This could be because the
/// specified relation does in fact not hold or because there is not enough
/// information in the constraint set. In other words, if we do not know for
/// sure, this function returns "false".
static bool ValueBoundsConstraintSet::compare(
    const Variable &lhs, ComparisonOperator cmp, const Variable &rhs);
```

one of:

- index attribute
- index-typed SSA value
- shaped value + dim
- affine map + operands (vars)

Example: Index-typed Values

```
func.func @test_case(%arg0: index, %arg1: index) {
  %0 = arith.addi %arg0, %arg1 : index
  %1 = arith.addi %arg1, %arg0 : index
  return
}
```

lhs		rhs	
%0	==	%1	→ true
%0	>=	%1	→ true
%0	<	%1	→ false
%0	>=	%arg0	→ false

```
ValueBoundsConstraintSet::compare(lhs, ValueBoundsConstraintSet::EQ, rhs);
```

Example: Index-typed Values

```
func.func @scf_for(%lb: index, %ub: index, %s: index) {
  scf.for %iv = %lb to %ub step %s { }
  return
}
```

lhs		rhs	
%iv	>=	%lb	→ true
%iv	<	%ub	→ true

Example: Dimensions of Shaped Values

```
func.func @scf_for_tensor(%init: tensor<?xf32>) {
  %r = scf.for %iv = %a to %b step %c
           iter_args(%t = %init) -> tensor<?xf32> {
    %0 = tensor.insert ... into %t[...]
    scf.yield %0 : tensor<?xf32>
  }
  return
}
```

lhs		rhs	
dim(%r, 0)	==	dim(%init, 0)	→ true
dim(%r, 0)	==	dim(%t, 0)	→ true

```
ValueBoundsConstraintSet::compare(
    {lhs, /*dim=*/0}, ValueBoundsConstraintSet::EQ, {rhs, /*dim=*/0});
```

Compute Bounds

```
/// Compute a bound for the given index-typed value or shape dimension size.
/// The computed bound is stored in `resultMap`. The operands of the bound are
/// stored in `mapOperands`. An operand is either an index-type SSA value
/// or a shaped value and a dimension.
static LogicalResult ValueBoundsConstraintSet::computeBound(
    AffineMap &resultMap, ValueDimList &mapOperands,
    presburger::BoundType type, const Variable &value,
    StopConditionFn stopCondition, bool closedUB = false);
```

determines which SSA values are allowed to appear in the bound

no guarantees how tight the bound is

API Example

```
func.func @test_case(%arg0: tensor<?xf32>) {
  %0 = tensor.insert ... into %arg0[...] : tensor<?xf32>
  %1 = linalg.generic outs(%0 : tensor<?xf32>) ...
}
```

affine_map<()[s0] -> (s0)>

[(%arg0, 0)]

%1

```
AffineMap map;
ValueDimList operands;  // SmallVector<std::pair<Value, std::optional<int64_t>>>
LogicalResult status = ValueBoundsConstraintSet::computeBound(map, operands, BoundType::EQ, {val, /*dim=*/0},
    /*stopCondition=*/[](Value v, std::optional<int64_t> dim, ...) {
        auto bbArg = dyn_cast<BlockArgument>(v);
        if (!bbArg)
          return false;
        return isa<FunctionOpInterface>(bbArg.getOwner());
    });
```

Only func bbArg in the bound

API Example

```
func.func @test_case(%arg0: tensor<?xf32>) {
  %0 = tensor.insert ... into %arg0[...] : tensor<?xf32>
  %1 = linalg.generic outs(%0 : tensor<?xf32>) ...
}
```

```
AffineMap map;
ValueDimList operands; ... ivector<std::pair<Value, std::optional<int64_t>>>
LogicalResult status = ValueBoundsConstraintSet::computeBound(map, operands, BoundType::EQ, {val, /*dim=*/0},
    /*stopCondition=*/[](Value v, std::optional<int64_t> dim, ...) { return false; });
```

failure: could not compute bound

No SSA values in the bound
(constant bound)

Related: `ReifyRankedShapedTypeOpInterface`

```
// RUN: mlir-opt -resolve-ranked-shaped-type-result-dims %s

%0 = tensor.insert %f into %arg0[%idx1] : tensor<?xf32>
%1 = tensor.insert %f into %0[%idx2] : tensor<?xf32>
%dim = tensor.dim %1, %c0 : tensor<?xf32>
```

- Fixed-point iteration of rewrite pattern: rewrite tensor/memref op result dim in terms of operands.
- Materializes IR for every operation (in theory, less efficient).
- Op interface driven.

Related: `ReifyRankedShapedTypeOpInterface`

```
// RUN: mlir-opt -resolve-ranked-shaped-type-result-dims %s

%0 = tensor.insert %f into %arg0[%idx1] : tensor<?xf32>
%1 = tensor.insert %f into %0[%idx2] : tensor<?xf32>
%dim = tensor.dim %1, %c0 : tensor<?xf32>
→

%dim = tensor.dim %0, %c0 : tensor<?xf32>
```

- Fixed-point iteration of rewrite pattern: rewrite tensor/memref op result dim in terms of operands.
- Materializes IR for every operation (in theory, less efficient).
- Op interface driven.

Related: `ReifyRankedShapedTypeOpInterface`

```
// RUN: mlir-opt -resolve-ranked-shaped-type-result-dims %s

%0 = tensor.insert %f into %arg0[%idx1] : tensor<?xf32>
%1 = tensor.insert %f into %0[%idx2] : tensor<?xf32>
%dim = tensor.dim %1, %c0 : tensor<?xf32>
→

%dim = tensor.dim %0, %c0 : tensor<?xf32>
→

%dim = tensor.dim %arg0, %c0 : tensor<?xf32>
```

- Fixed-point iteration of rewrite pattern: rewrite tensor/memref op result dim in terms of operands.
- Materializes IR for every operation (in theory, less efficient).
- Op interface driven.

ValueBoundsOpInterface

Example: `arith.addi`

op result or block argument

```
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());
  }
};
```

get affine expr for SSA value

affine expression, constant or SSA value

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

Example: `arith.addi`

```
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");

   AffineExpr lhs = cstr.getExpr(addIOp.getLhs()), rhs = cstr.getExpr(addIOp.getLhs());
   cstr.bound(value) == lhs + rhs;
 }
};
```

getExpr has side effects

Example: `tensor.pad`

quite a bit simpler than
`ReifyRankedShapedTypeOpInterface`

```
struct PadOpInterface
   : public ValueBoundsOpInterface::ExternalModel<PadOpInterface, PadOp> {
 void populateBoundsForShapedValueDim(Operation *op, Value value, int64_t dim,
                                      ValueBoundsConstraintSet &cstr) const {
   auto padOp = cast<PadOp>(op);
   assert(value == padOp.getResult() && "invalid value");
   AffineExpr srcSize = cstr.getExpr(padOp.getSource(), dim);
   AffineExpr lowPad = cstr.getExpr(padOp.getMixedLowPad()[dim]);
   AffineExpr highPad = cstr.getExpr(padOp.getMixedHighPad()[dim]);
   cstr.bound(value)[dim] == srcSize + lowPad + highPad;
 }
};
```

add bound for dim size of shaped value

Implementation Details

Constraint Set: `FlatLinearConstraints`

$$Ax + b = 0$$

$$Ax + b \geq 0$$

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
  - compute LB/UB of a variable
  - check if constraint set is "empty"

Example: Constraints

computed bound should have only func args

```
// RUN: mlir-opt %s -test-affine-reify-value-bounds="reify-to-func-args"

func.func @test_case(%arg0: index, %arg1: index, %arg2: index) -> index {
  %0 = arith.addi %arg0, %arg1 : index        // %0 - %arg0 - %arg1 = 0
  %1 = arith.addi %0, %arg2 : index           // %1 - %0    - %arg2 = 0
  %r = "test.reify_bound"(%1) {type = "EQ"} : (index) -> (index)
  return %r : index
}
```

replace this op with the computed bound

```
Constraint set: 5 variables
(%1   %0    %arg2 %arg0 %arg1 const)
 1    -1    -1     0     0     0      = 0
 0     1     0    -1    -1     0      = 0
```

A b

Example: Constraints

computed bound should have only func args

```
// RUN: mlir-opt %s -test-affine-reify-value-bounds="reify-to-func-args"

func.func @test_case(%arg0: index, %arg1: index, %arg2: index) -> index {
  %0 = arith.addi %arg0, %arg1 : index        // %0 - %arg0 - %arg1 = 0
  %1 = arith.addi %0, %arg2 : index           // %1 - %0    - %arg2 = 0
  %r = "test.reify_bound"(%1) {type = "EQ"} : (index) -> (index)
  return %r : index
}
```

project out %0

replace this op with the computed bound

```
Constraint set: 4 variables
(%1  ~~%0~~  %arg2 %arg0 %arg1 const)
 1   ~~0~~   -1    -1    -1    0        = 0
```

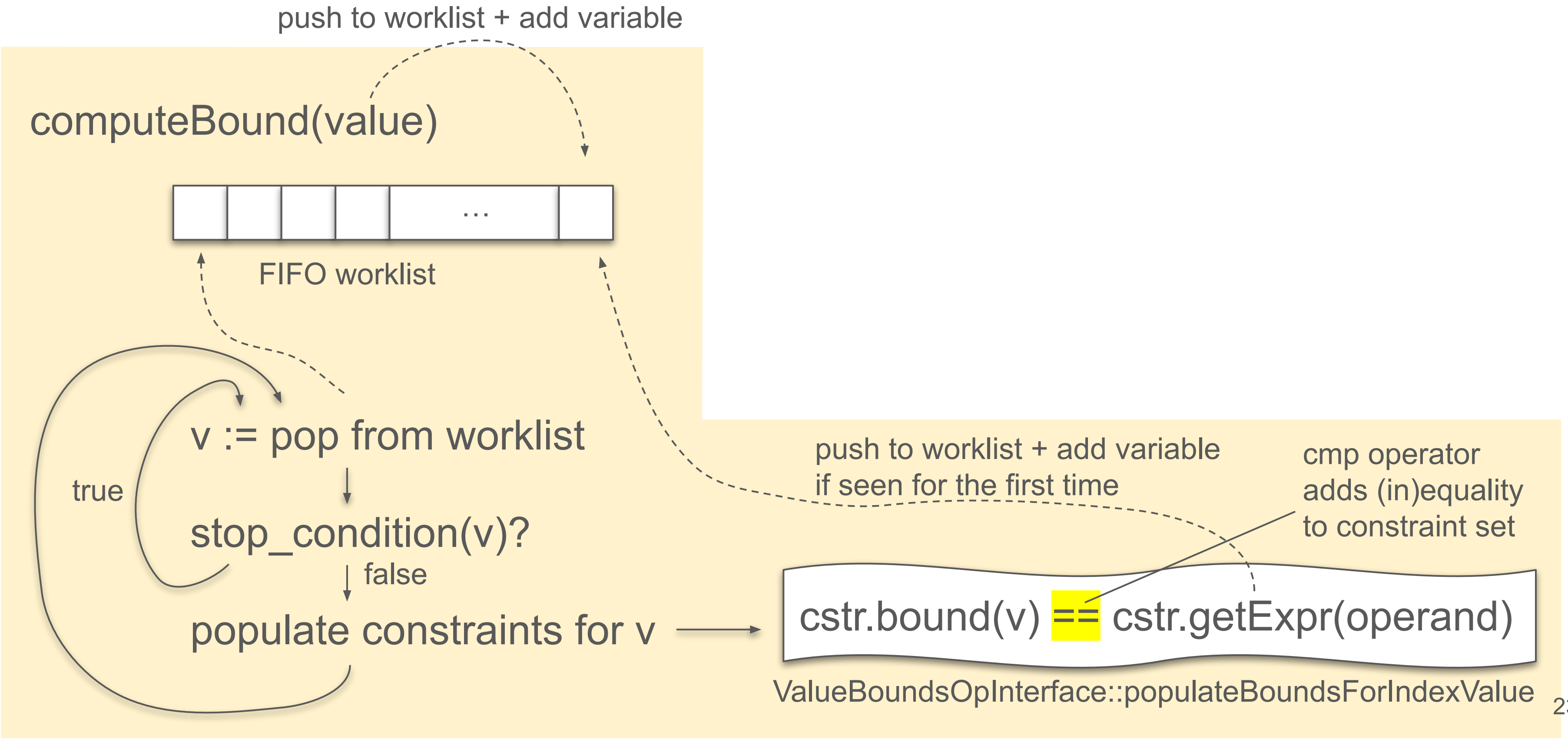
$\underbrace{\text{%1 %0 %arg2 %arg0 %arg1}}_{A}\ \underbrace{\text{const}}_{b}$

Example: Constraints

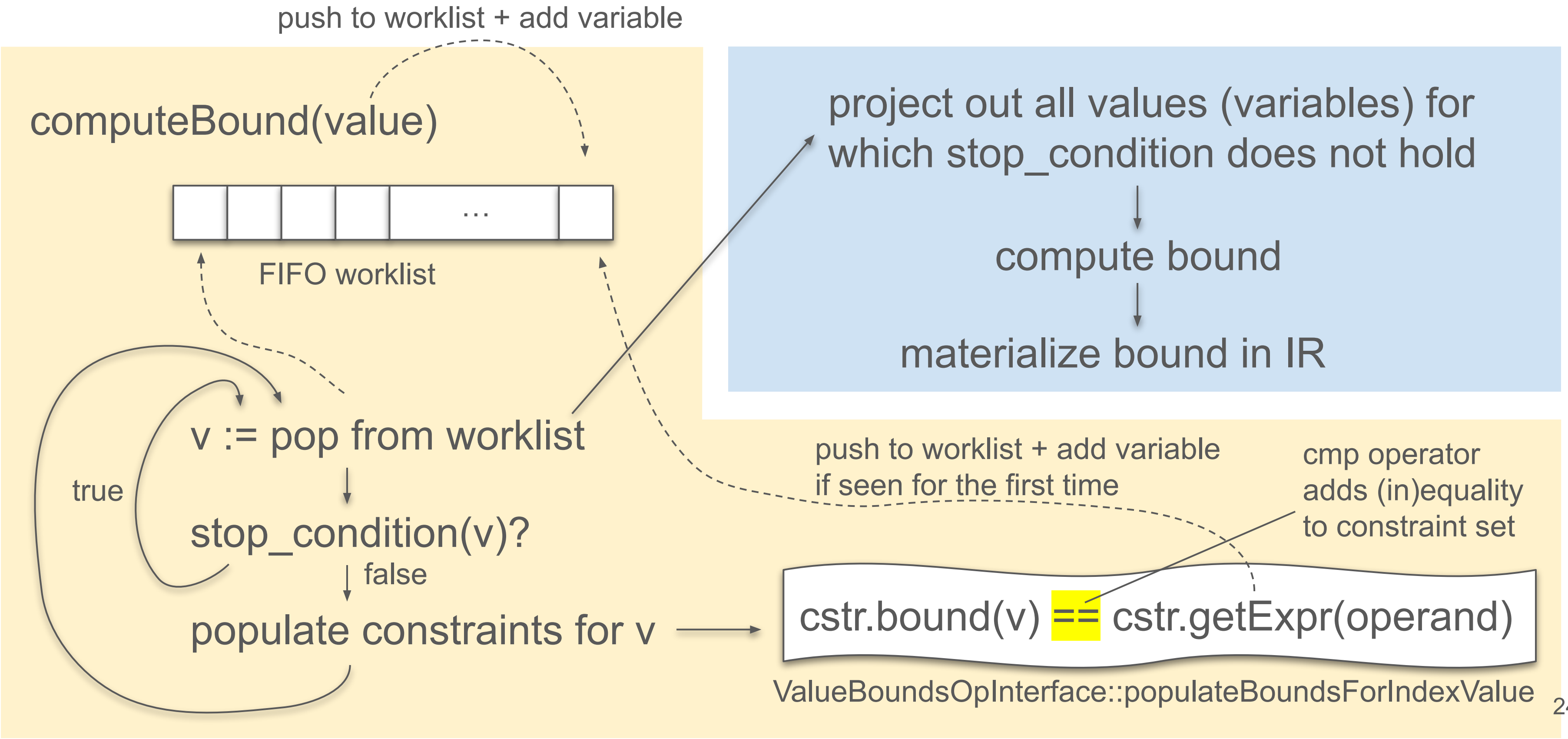
```
#map = affine_map<()[s0, s1, s2] -> (s0 + s1 + s2)>

func.func @test_case(%arg0: index, %arg1: index, %arg2: index) -> index {
  %0 = arith.addi %arg0, %arg1 : index
  %1 = arith.addi %0, %arg2 : index
  %r = affine.apply #map()[%arg2, %arg0, %arg1]
  return %r : index
}
```

can also be reified with arith dialect ops

Computing Bounds: Worklist-Driven IR Analysis



Computing Bounds: Worklist-Driven IR Analysis



Overview: Constraints for Various Operations

- `%r = arith.addi %a, %b : index`
    - %r == %a + %b
- `%r = arith.constant 5 : index`
    - %r == 5
- `%r = memref.subview %m[%offset0, %offset1] [%size0, %size1] [1, 1]`
  `: memref<?x?xf32> to memref<?x?xf32, strided<[?, 1], offset: ?>>`
    - dim(%r, 0) == %size0
    - dim(%r, 1) == %size1
- Destination-style Op: `%r = out(%t : tensor<?x?xf32>)`
    - dim(%r, 0) == dim(%t, 0)
    - dim(%r, 1) == dim(%t, 1)
- `%r = affine.max affine_map<()[s0] -> (s0, 2)>()[%a]`
    - %r <= 2
    - %r <= %a
- `%r = affine.apply affine_map<()[s0, s1] -> (s0 + s1 mod 8)>()[%a, %b]`
    - %r == expr(%a) + (expr(%b) mod 8)

Constraints for `scf.if`

```
%r = scf.if %c -> index {
  scf.yield %a : index
} else {
  scf.yield %b : index
}
```

%r >= min(%a, %b)

cannot be represented in the constraint set

%r <= max(%a, %b)

Constraints for `scf.if`

```
%r = scf.if %c -> index {
  scf.yield %a : index
} else {
  scf.yield %b : index
}
```

If %a <= %b:

- %r >= %a
- %r <= %b

If %b <= %a:

- %r >= %b
- %r <= %a

Constraints for `scf.if`

```
%r = scf.if %c -> index {
  scf.yield %a : index
} else {
  scf.yield %b : index
}
```

If %a <= %b:

- %r >= %a
- %r <= %b

If %b <= %a:

- %r >= %b
- %r <= %a

```
struct IfOpInterface
    : public ValueBoundsOpInterface::ExternalModel<IfOpInterface, IfOp> {

  void populateBoundsForIndexValue(Operation *op, Value value,
                                   ValueBoundsConstraintSet &cstr) const {
    unsigned int resultNum = cast<OpResult>(value).getResultNumber();
    Value thenValue = ifOp.thenYield().getResults()[resultNum];
    Value elseValue = ifOp.elseYield().getResults()[resultNum];

    if (cstr.populateAndCompare(thenValue, ComparisonOperator::LE, elseValue)) {
        cstr.bound(value) >= thenValue;
        cstr.bound(value) <= elseValue;
    }

   ...
```

Constraints for `scf.for`

```
%r = scf.for %iv = %lb to %ub step %s iter_args(%a = %t) -> tensor<?xf32>
{
  // ...
  scf.yield %0 : tensor<?xf32>
}
```

%iv >= %lb
%iv < %ub

If dim(%0, 0) == dim(%a, 0):

- dim(%r, 0) == dim(%t, 0)
- dim(%a, 0) == dim(%t, 0)

Comparing Values/Dimensions

Prove that %a >= %b. Prove by contradiction:

- Populate constraints for %a and %b.
- Assert that the constraint set is not empty (i.e., has a solution).
- Insert the inverse constraint: %a < %b
- If the constraint set is now empty, %a >= %b holds.

Stop condition: Keep traversing IR and populating constraints until the relation can be proven (or until we run out of IR to analyze).

Limitations and Future Work

Limitations and Future Work

- Non-flattenable expressions
  - Expressions that cannot be represented as linear combination of variables are not supported.
  - Example: `%r = arith.muli %a, %b : index`
- Performance: Still a lot of room for improvement
  - Stop function-based traversal may traverse more IR than necessary.
  - New constraint set is built for each computed bound (with new IR traversal).
- Cases where `FlatLinearConstraints` computes multiple bounds are not supported.
  - Example: `%r = affine.max affine_map<()[s0, s1] -> (s0, s1)>()[%a, %b]`
- Unify with `ReifyRankedShapedTypeOpInterface` and/or `InferIntRangeInterface`?

Questions?

- [ValueBoundsOpInterface]
- [ValueBoundsConstraintSet]
- [ReifyRankedShapedTypeOpInterface]
- [InferIntRangeInterface]
- [DestinationStyleOpInterface]
- [MLIR Presburger Library] (`FlatLinearConstraints`)
- Compare values/dimensions
- Compute LB/UB/EQ bound of value/dimension
- Worklist-Driven IR Analysis
- Stop condition
- Materialize bound with affine/arith dialect ops
- Branches (`scf.if`, `arith.select`)
- Loops (`scf.for`)
- Non-flattenable expressions (e.g., multiplications)
- Computing bounds for multiple values/dimensions

Appendix

API: Convenience Functions

- ```
  static FailureOr<int64_t> ValueBoundsConstraintSet::computeConstantBound(
       presburger::BoundType type, const Variable &var,
       StopConditionFn stopCondition = nullptr, bool closedUB = false);
  ```
  → Like `computeBound`, but stop condition is optional.
- ```
  static FailureOr<bool> areEqual(const Variable &var1, const Variable &var2);
  ```
  → Implemented in terms of `compare`:
  - var1 == var2: true
  - var1 < var2 or var2 > var1: false
  - otherwise: failure
- ```
  static FailureOr<bool> areEquivalentSlices(MLIRContext *ctx, HyperrectangularSlice slice1,
                                             HyperrectangularSlice slice2);
  ```
- ```
  static FailureOr<bool> areOverlappingSlices(MLIRContext *ctx, HyperrectangularSlice slice1,
                                              HyperrectangularSlice slice2);
  ```

Ex.: Matmul Tiling [4, 4, 4]

```
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 [4, 4, 4]
        : (!transform.any_op) -> (!transform.any_op, !transform.any_op, !transform.any_op, !transform.any_op)
    transform.yield
  }
}

func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>, %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {
  %0 = linalg.matmul ins(%arg0, %arg1: tensor<128x128xf32>, tensor<128x128xf32>)
                     outs(%arg2: tensor<128x128xf32>) -> tensor<128x128xf32>
  return %0 : tensor<128x128xf32>
}
```

Ex.: Matmul Tiling [4, 4, 4]

```
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 %c4 iter_args(%arg6 = %arg4) -> (tensor<128x128xf32>) {
     %2 = scf.for %arg7 = %c0 to %c128 step %c4 iter_args(%arg8 = %arg6) -> (tensor<128x128xf32>) {
       %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, 4] [1, 1] : tensor<128x128xf32> to tensor<4x4xf32>
       %extracted_slice_1 = tensor.extract_slice %arg8[%arg3, %arg5] [4, 4] [1, 1] : tensor<128x128xf32> to tensor<4x4xf32>
       %3 = linalg.matmul ins(%extracted_slice, %extracted_slice_0 : tensor<4x4xf32>, tensor<4x4xf32>)
                          outs(%extracted_slice_1 : tensor<4x4xf32>) -> tensor<4x4xf32>
       %inserted_slice = tensor.insert_slice %3 into %arg8[%arg3, %arg5] [4, 4] [1, 1] : tensor<4x4xf32> into tensor<128x128xf32>
       scf.yield %inserted_slice : tensor<128x128xf32>
     }
     scf.yield %2 : tensor<128x128xf32>
   }
   scf.yield %1 : tensor<128x128xf32>
 }
 return %0 : tensor<128x128xf32>
}
```

Ex.: Matmul Tiling [4, 9, 4]

```
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>
}
```

tile size does not divide tensor evenly

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<
       %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)