MLIR Testing Guide

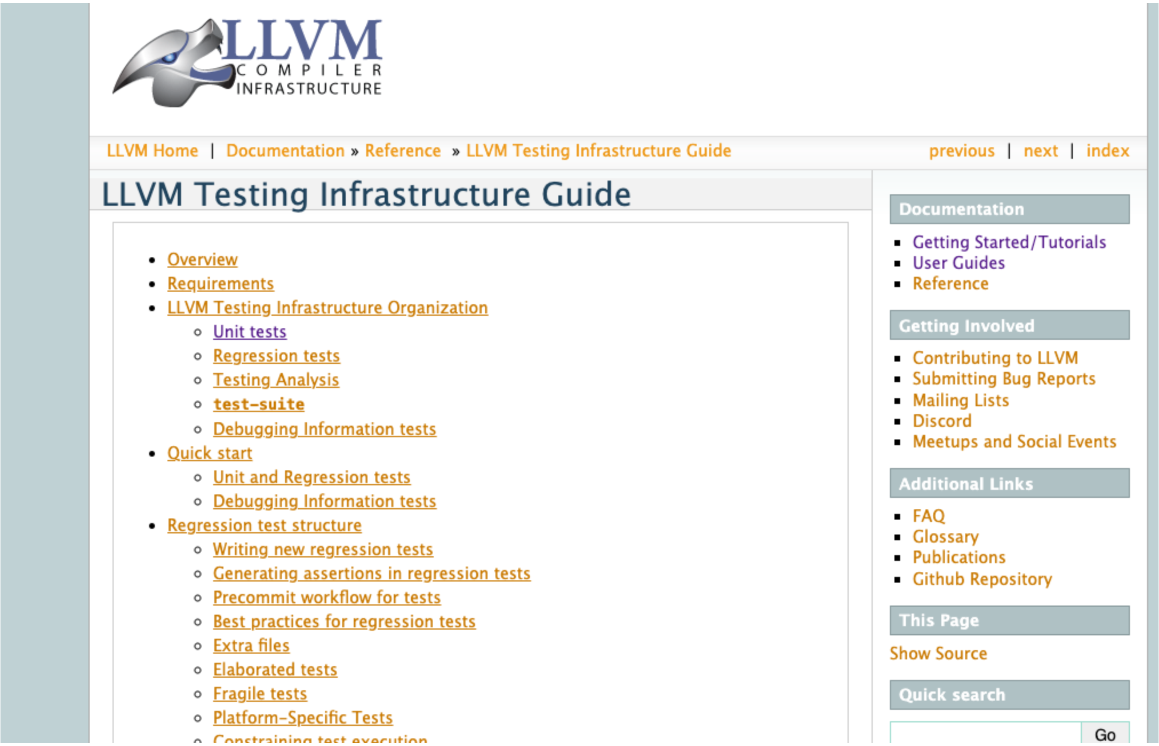
What and Why?

Andrzej Warzyński

October 29th, 2025

Testing Guide: **LLVM** + **MLIR**

https://llvm.org/docs/TestingGuide.html



LLVM Home | Documentation » Reference » LLVM Testing Infrastructure Guide previous | next | index

LLVM Testing Infrastructure Guide

- Overview
- Requirements
- LLVM Testing Infrastructure Organization
  - Unit tests
  - Regression tests
  - Testing Analysis
  - `test-suite`
  - Debugging Information tests
- Quick start
  - Unit and Regression tests
  - Debugging Information tests
- Regression test structure
  - Writing new regression tests
  - Generating assertions in regression tests
  - Precommit workflow for tests
  - Best practices for regression tests
  - Extra files
  - Elaborated tests
  - Fragile tests
  - Platform-Specific Tests
  - Constraining test execution

Documentation

- Getting Started/Tutorials
- User Guides
- Reference

Getting Involved

- Contributing to LLVM
- Submitting Bug Reports
- Mailing Lists
- Discord
- Meetups and Social Events

Additional Links

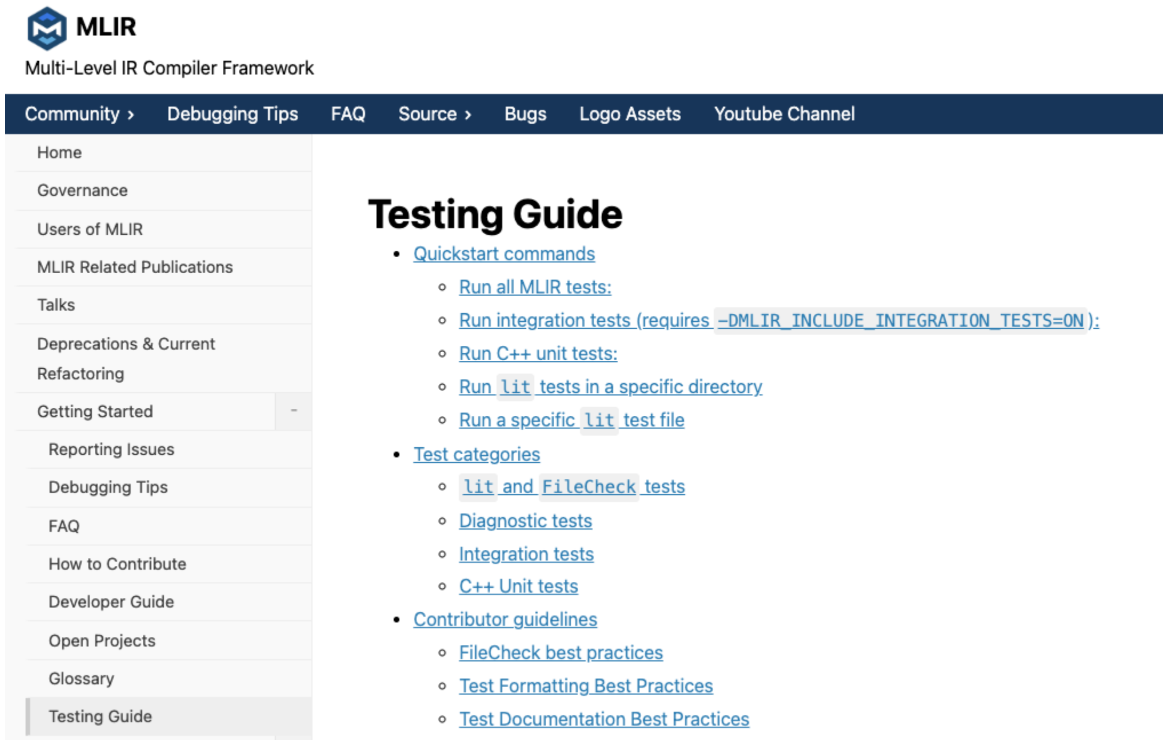
- FAQ
- Glossary
- Publications
- Github Repository

This Page

Show Source

Quick search

https://mlir.llvm.org/getting_started/TestingGuide/



MLIR
Multi-Level IR Compiler Framework
Community › Debugging Tips FAQ Source › Bugs Logo Assets Youtube Channel
Home
Governance
Users of MLIR
MLIR Related Publications
Talks
Deprecations & Current Refactoring
Getting Started
Reporting Issues
Debugging Tips
FAQ
How to Contribute
Developer Guide
Open Projects
Glossary
Testing Guide

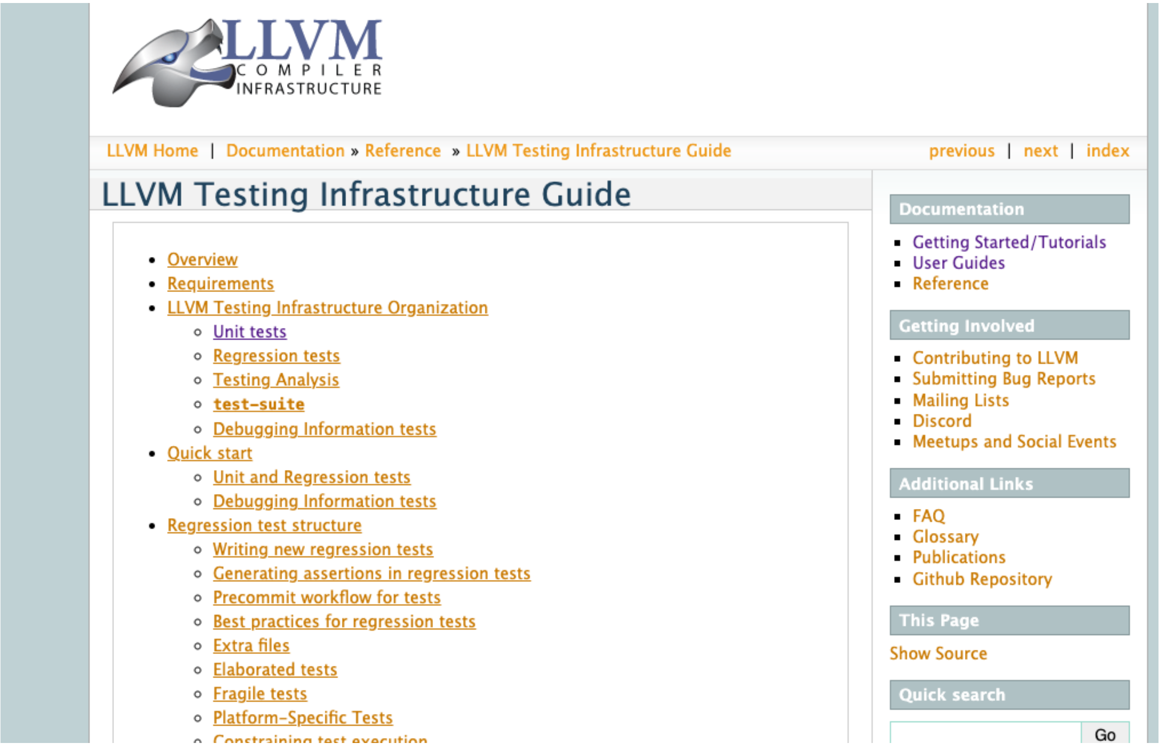
Testing Guide

- Quickstart commands
  - Run all MLIR tests:
  - Run integration tests (requires `-DMLIR_INCLUDE_INTEGRATION_TESTS=ON`):
  - Run C++ unit tests:
  - Run `lit` tests in a specific directory
  - Run a specific `lit` test file
- Test categories
  - `lit` and `FileCheck` tests
  - Diagnostic tests
  - Integration tests
  - C++ Unit tests
- Contributor guidelines
  - FileCheck best practices
  - Test Formatting Best Practices
  - Test Documentation Best Practices

Testing Guide: **LLVM + MLIR**

This presentation!

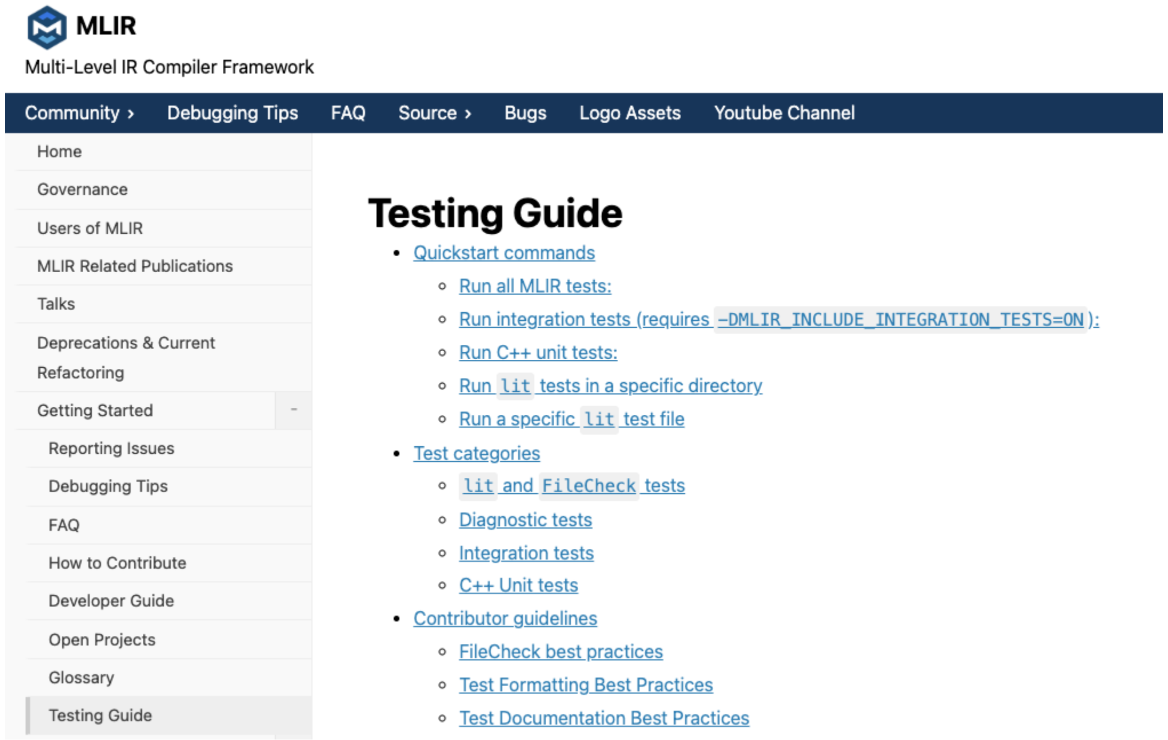
https://llvm.org/docs/TestingGuide.html



LLVM Testing Infrastructure Guide

- Overview
- Requirements
- LLVM Testing Infrastructure Organization
  - Unit tests
  - Regression tests
  - Testing Analysis
  - `test-suite`
  - Debugging Information tests
- Quick start
  - Unit and Regression tests
  - Debugging Information tests
- Regression test structure
  - Writing new regression tests
  - Generating assertions in regression tests
  - Precommit workflow for tests
  - Best practices for regression tests
  - Extra files
  - Elaborated tests
  - Fragile tests
  - Platform-Specific Tests
  - Constraining test execution

https://mlir.llvm.org/getting_started/TestingGuide/



Testing Guide

- Quickstart commands
  - Run all MLIR tests:
  - Run integration tests (requires `-DMLIR_INCLUDE_INTEGRATION_TESTS=ON`):
  - Run C++ unit tests:
  - Run `lit` tests in a specific directory
  - Run a specific `lit` test file
- Test categories
  - `lit` and `FileCheck` tests
  - Diagnostic tests
  - Integration tests
  - C++ Unit tests
- Contributor guidelines
  - FileCheck best practices
  - Test Formatting Best Practices
  - Test Documentation Best Practices

The remaining slides: MLIR Testing Guide overview with **LIT** + **FileCheck** examples ☺

Why?

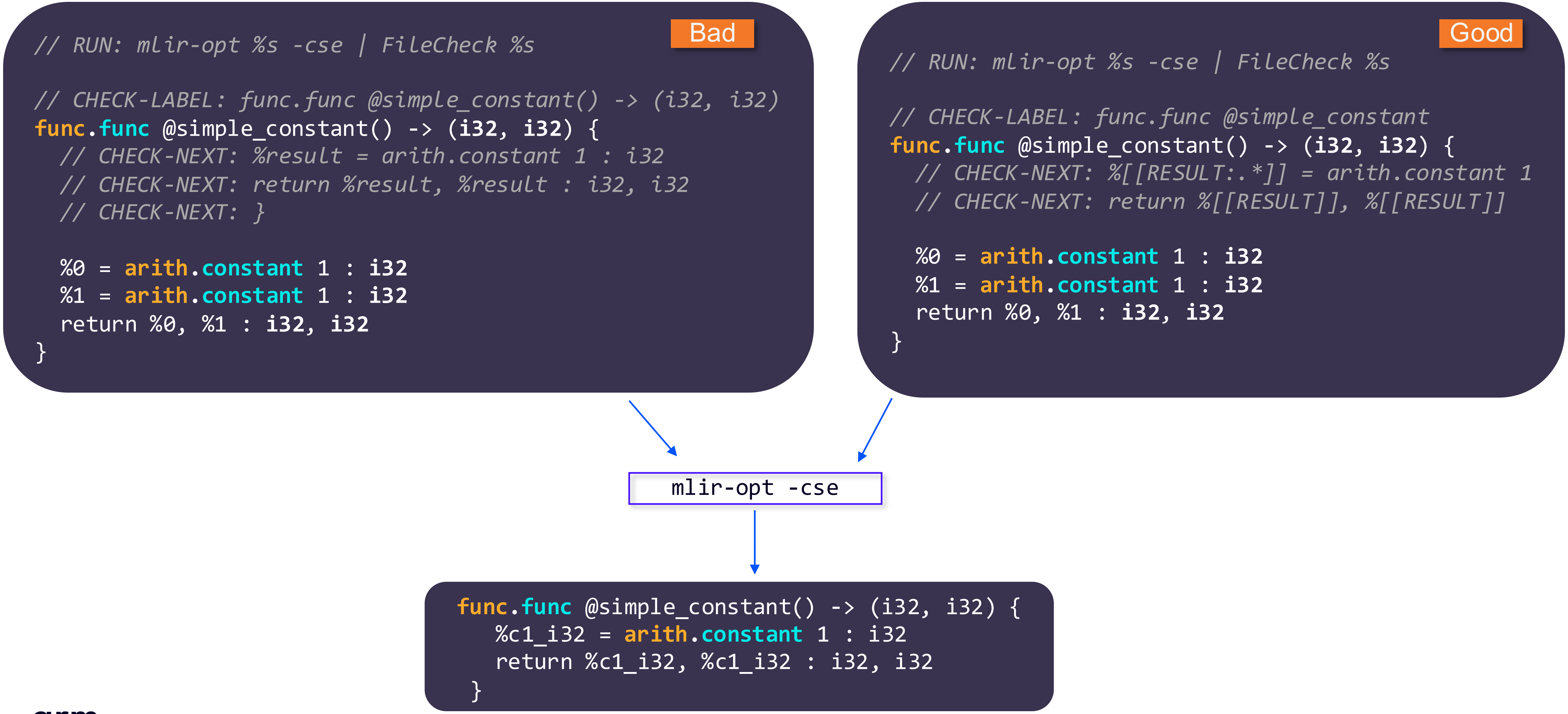
Tests document our code – write it as such.

- **Tests are key** - If you want to understand the code, look at the tests.
- **Test discoverability** – Well-documented tests make it easier to pair tests with patterns and understand their purpose.
- **Test consistency** – Consistent documentation and naming lowers cognitive load and helps avoid duplication.

Anything else?

- To avoid a situation where we end up with tests that we don’t understand!
- To make it easier to maintain MLIR!

Only test what’s required!



Bad

```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant() -> (i32, i32)
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %result = arith.constant 1 : i32
  // CHECK-NEXT: return %result, %result : i32, i32
  // CHECK-NEXT: }

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

Good

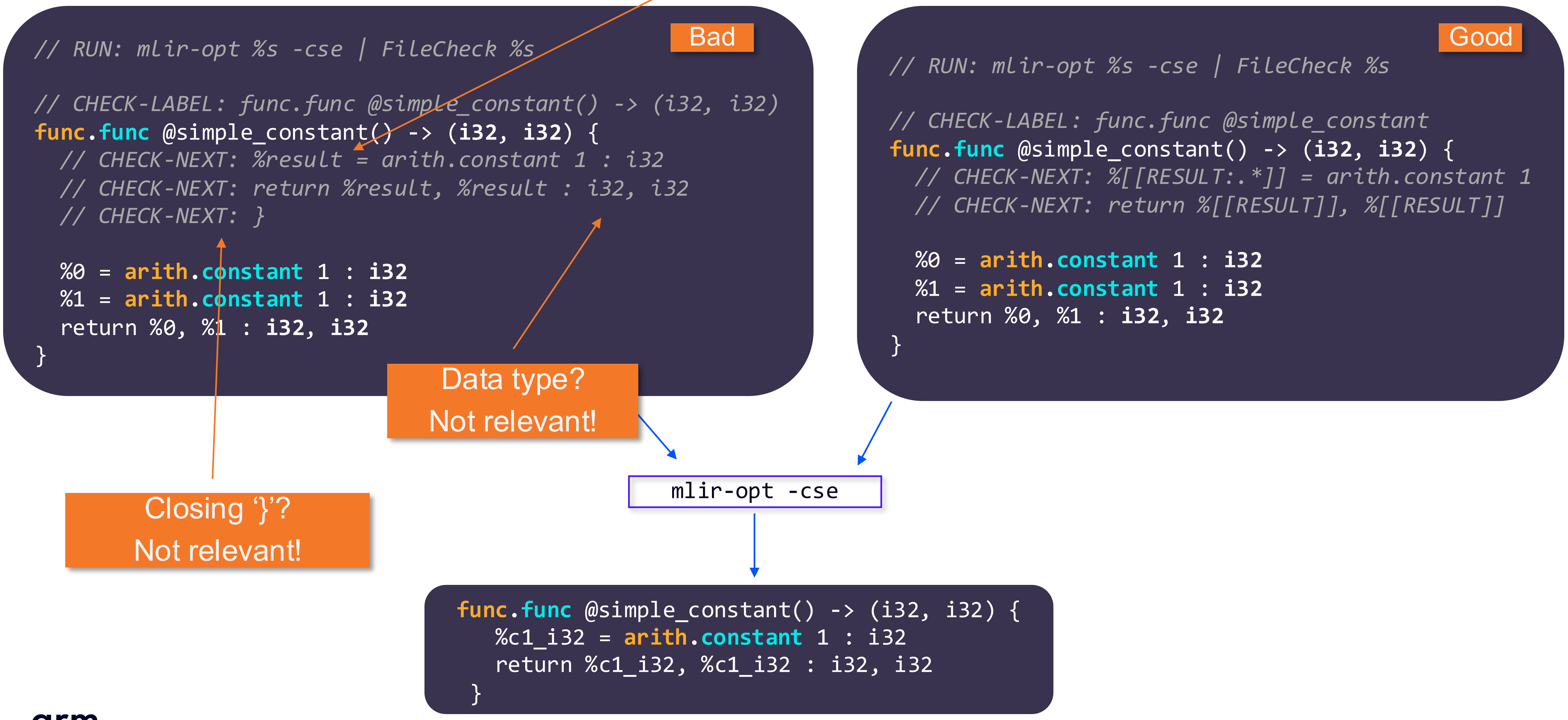
```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %[[RESULT:.*]] = arith.constant 1
  // CHECK-NEXT: return %[[RESULT]], %[[RESULT]]

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

```
mlir-opt -cse
```

```
func.func @simple_constant() -> (i32, i32) {
   %c1_i32 = arith.constant 1 : i32
   return %c1_i32, %c1_i32 : i32, i32
 }
```

Only test what's required!

Use FileCheck variables!



Bad

```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant() -> (i32, i32)
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %result = arith.constant 1 : i32
  // CHECK-NEXT: return %result, %result : i32, i32
  // CHECK-NEXT: }

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

Good

```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %[[RESULT:.*]] = arith.constant 1
  // CHECK-NEXT: return %[[RESULT]], %[[RESULT]]

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

Data type?
Not relevant!

Closing '}'?
Not relevant!

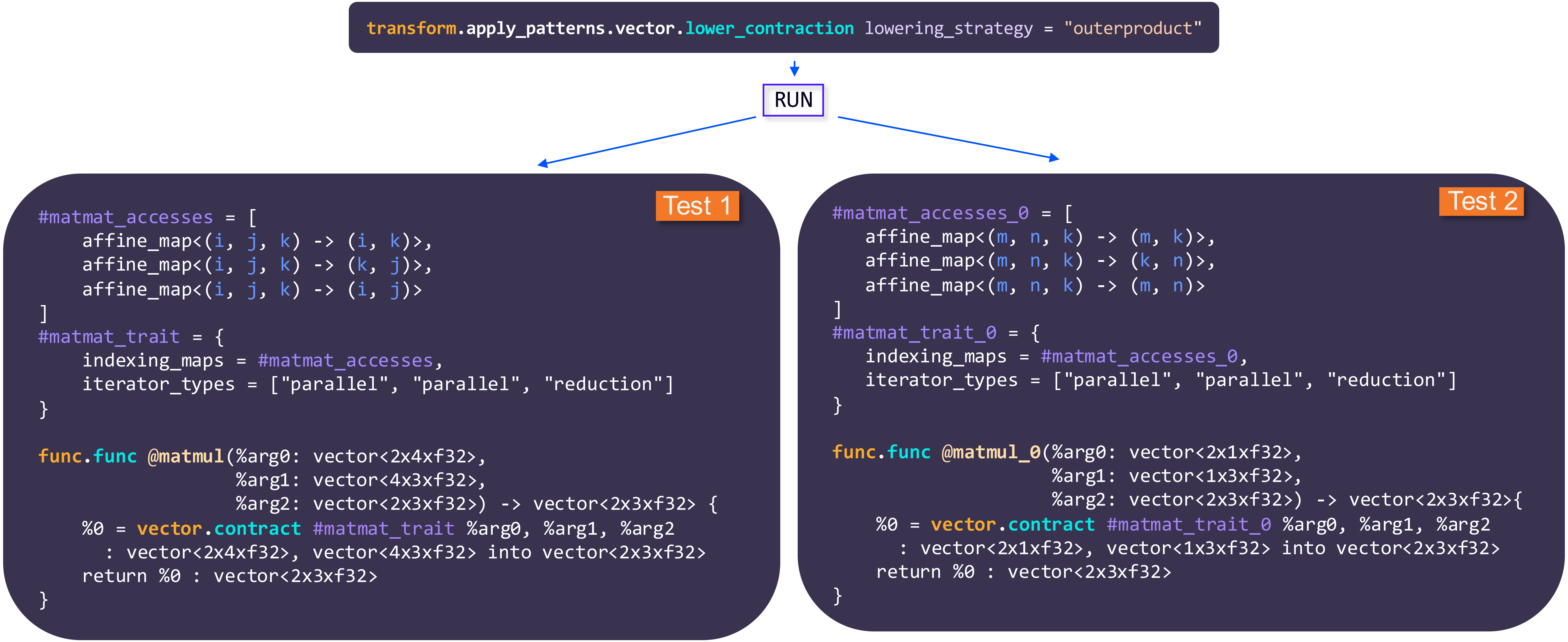
```
mlir-opt -cse
```

```
func.func @simple_constant() -> (i32, i32) {
  %c1_i32 = arith.constant 1 : i32
  return %c1_i32, %c1_i32 : i32, i32
}
```

arm

Duplicate tests in one file

Tests for `vector.contract` → `vector.outerproduct` lowering



```
transform.apply_patterns.vector.lower_contraction lowering_strategy = "outerproduct"
```

RUN

Test 1

```
#matmat_accesses = [
    affine_map<(i, j, k) -> (i, k)>,
    affine_map<(i, j, k) -> (k, j)>,
    affine_map<(i, j, k) -> (i, j)>
]
#matmat_trait = {
    indexing_maps = #matmat_accesses,
    iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul(%arg0: vector<2x4xf32>,
                  %arg1: vector<4x3xf32>,
                  %arg2: vector<2x3xf32>) -> vector<2x3xf32> {
    %0 = vector.contract #matmat_trait %arg0, %arg1, %arg2
      : vector<2x4xf32>, vector<4x3xf32> into vector<2x3xf32>
    return %0 : vector<2x3xf32>
}
```

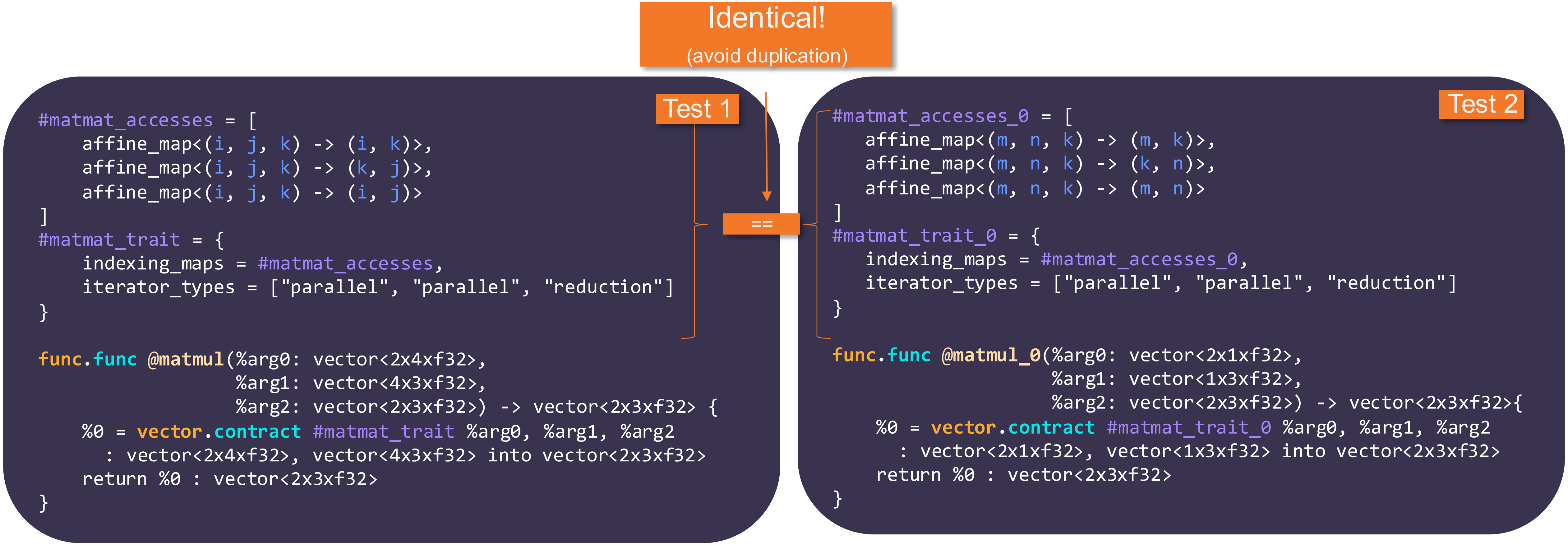
Test 2

```
#matmat_accesses_0 = [
   affine_map<(m, n, k) -> (m, k)>,
   affine_map<(m, n, k) -> (k, n)>,
   affine_map<(m, n, k) -> (m, n)>
]
#matmat_trait_0 = {
   indexing_maps = #matmat_accesses_0,
   iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul_0(%arg0: vector<2x1xf32>,
                    %arg1: vector<1x3xf32>,
                    %arg2: vector<2x3xf32>) -> vector<2x3xf32>{
    %0 = vector.contract #matmat_trait_0 %arg0, %arg1, %arg2
      : vector<2x1xf32>, vector<1x3xf32> into vector<2x3xf32>
    return %0 : vector<2x3xf32>
}
```

Duplicate tests in one file

Tests for `vector.contract` → `vector.outerproduct` lowering

- If the traits are identical, what is the difference between **@matmul** and **@matmul_0**?
  - Not a problem with 1-2 tests, but there is 100s!



Identical!

(avoid duplication)

==

Test 1

```
#matmat_accesses = [
    affine_map<(i, j, k) -> (i, k)>,
    affine_map<(i, j, k) -> (k, j)>,
    affine_map<(i, j, k) -> (i, j)>
]
#matmat_trait = {
    indexing_maps = #matmat_accesses,
    iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul(%arg0: vector<2x4xf32>,
                  %arg1: vector<4x3xf32>,
                  %arg2: vector<2x3xf32>) -> vector<2x3xf32> {
    %0 = vector.contract #matmat_trait %arg0, %arg1, %arg2
      : vector<2x4xf32>, vector<4x3xf32> into vector<2x3xf32>
    return %0 : vector<2x3xf32>
}
```

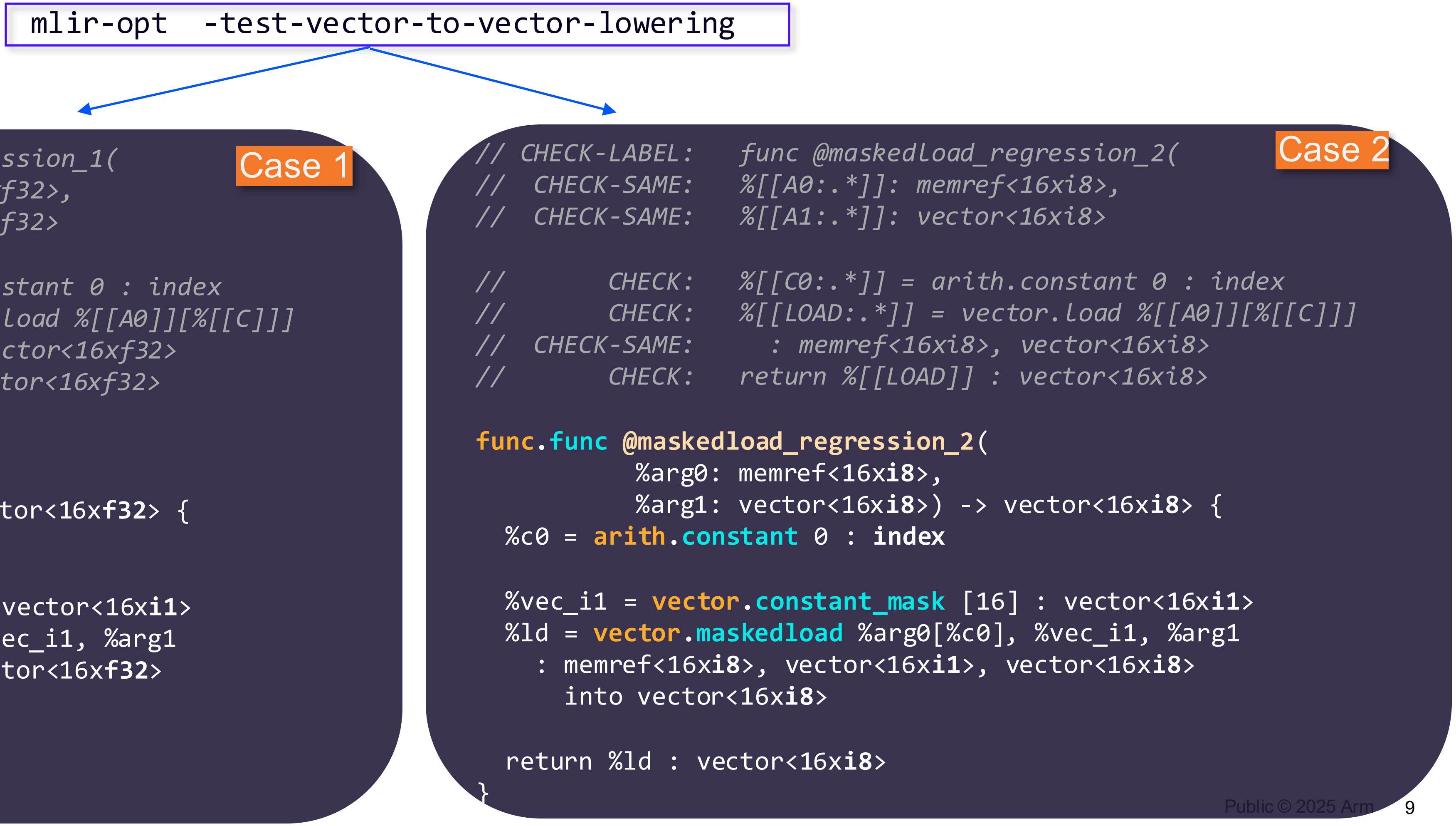
Test 2

```
#matmat_accesses_0 = [
   affine_map<(m, n, k) -> (m, k)>,
   affine_map<(m, n, k) -> (k, n)>,
   affine_map<(m, n, k) -> (m, n)>
]
#matmat_trait_0 = {
   indexing_maps = #matmat_accesses_0,
   iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul_0(%arg0: vector<2x1xf32>,
                    %arg1: vector<1x3xf32>,
                    %arg2: vector<2x3xf32>) -> vector<2x3xf32>{
    %0 = vector.contract #matmat_trait_0 %arg0, %arg1, %arg2
      : vector<2x1xf32>, vector<1x3xf32> into vector<2x3xf32>
    return %0 : vector<2x3xf32>
}
```

Seemingly “similar” tests

Missed opportunities to encode helpful info in tests.

- Leverage test function names + variable names.



```
mlir-opt  -test-vector-to-vector-lowering
```

Case 1

```
// CHECK-LABEL:   func @maskedload_regression_1(
//  CHECK-SAME:   %[[A0:.*]]: memref<16xf32>,
//  CHECK-SAME:   %[[A1:.*]]: vector<16xf32>

//       CHECK:   %[[C0:.*]] = arith.constant 0 : index
//       CHECK:   %[[LOAD:.*]] = vector.load %[[A0]][%[[C]]]
//  CHECK-SAME:     : memref<16xf32>, vector<16xf32>
//       CHECK:   return %[[LOAD]] : vector<16xf32>

func.func @maskedload_regression_1(
           %arg0: memref<16xf32>,
           %arg1: vector<16xf32>) -> vector<16xf32> {
  %c0 = arith.constant 0 : index

  %vec_i1 = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %arg0[%c0], %vec_i1, %arg1
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
}
```

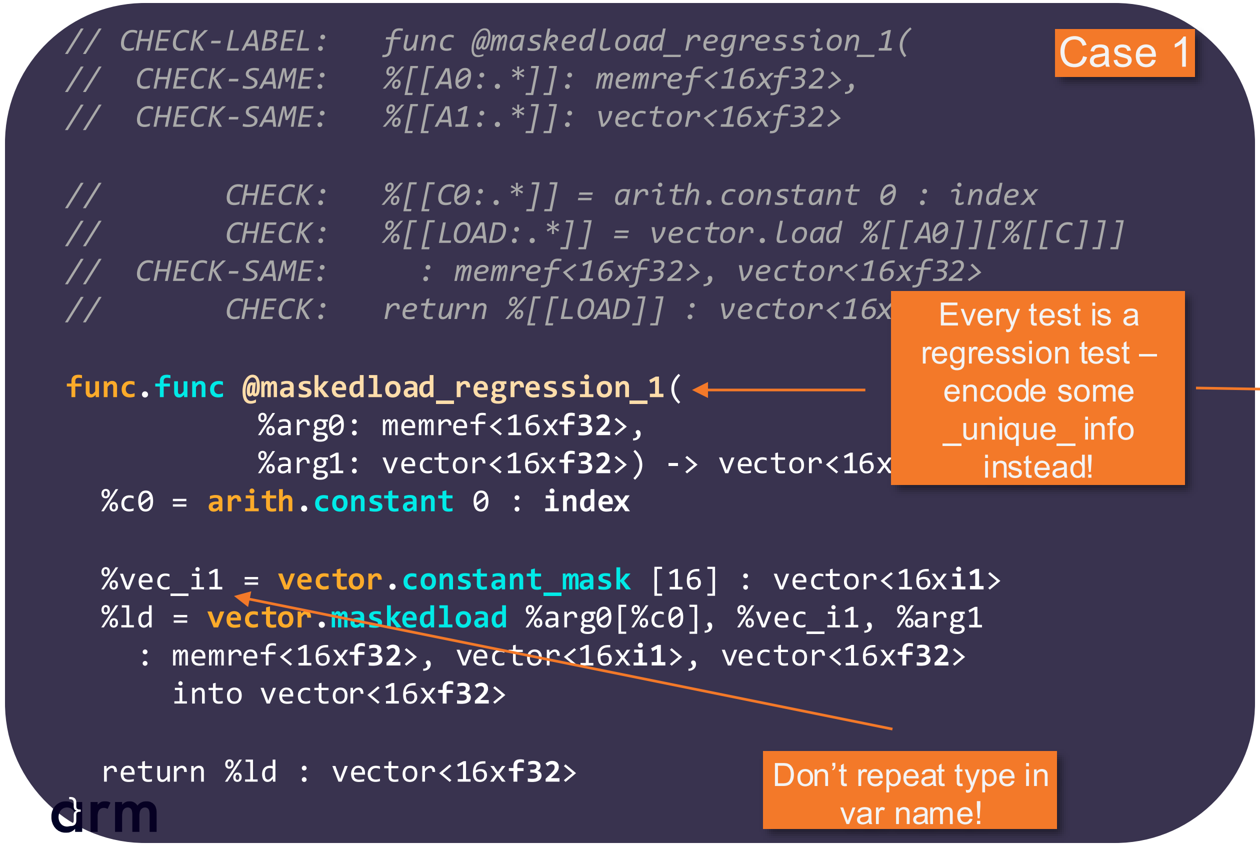
Case 2

```
// CHECK-LABEL:   func @maskedload_regression_2(
//  CHECK-SAME:   %[[A0:.*]]: memref<16xi8>,
//  CHECK-SAME:   %[[A1:.*]]: vector<16xi8>

//       CHECK:   %[[C0:.*]] = arith.constant 0 : index
//       CHECK:   %[[LOAD:.*]] = vector.load %[[A0]][%[[C]]]
//  CHECK-SAME:     : memref<16xi8>, vector<16xi8>
//       CHECK:   return %[[LOAD]] : vector<16xi8>

func.func @maskedload_regression_2(
           %arg0: memref<16xi8>,
           %arg1: vector<16xi8>) -> vector<16xi8> {
  %c0 = arith.constant 0 : index

  %vec_i1 = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %arg0[%c0], %vec_i1, %arg1
    : memref<16xi8>, vector<16xi1>, vector<16xi8>
      into vector<16xi8>

  return %ld : vector<16xi8>
}
```

Seemingly “similar” tests

Missed opportunities to encode helpful info in tests.

- Missed opportunities to encode helpful info in tests.
  - Why not leverage test function names + variable names?

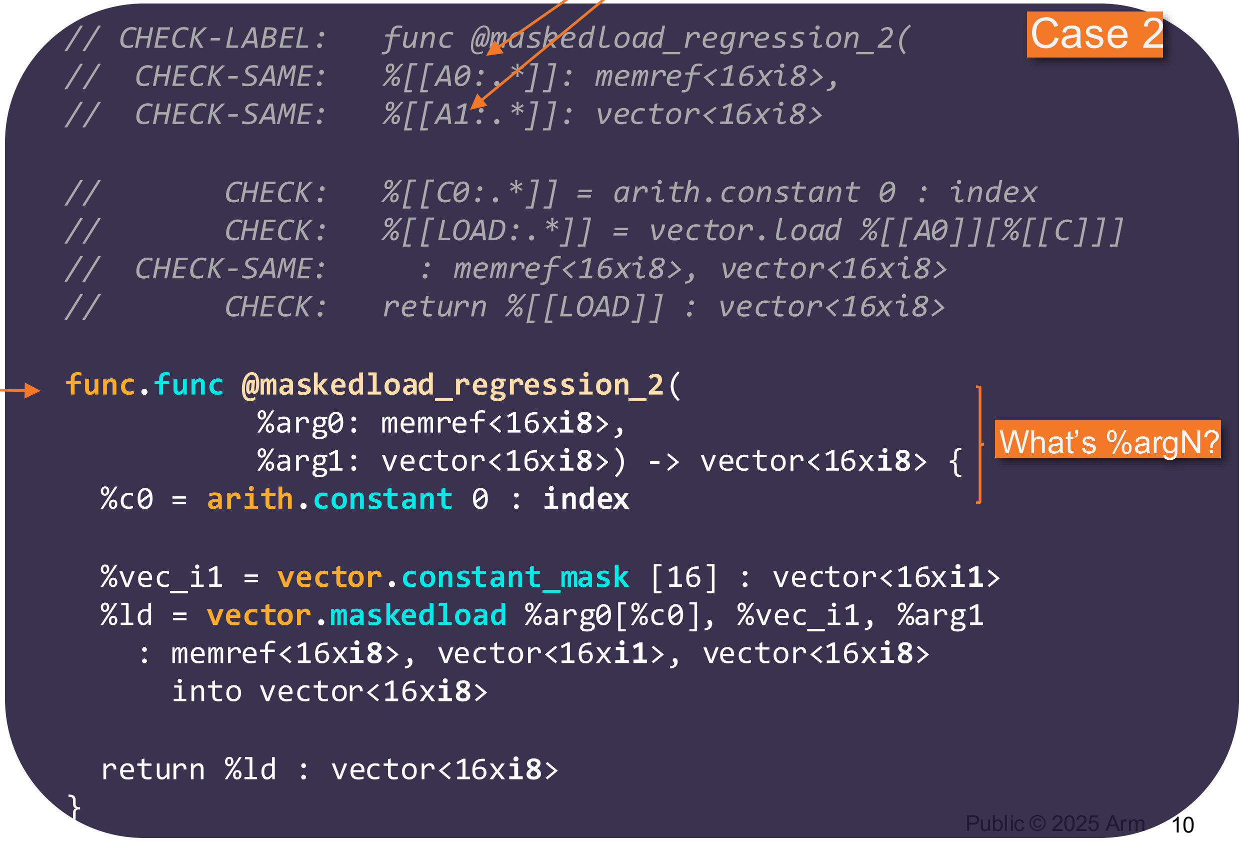


Case 1

```
// CHECK-LABEL:   func @maskedload_regression_1(
//  CHECK-SAME:   %[[A0:.*]]: memref<16xf32>,
//  CHECK-SAME:   %[[A1:.*]]: vector<16xf32>

//        CHECK:   %[[C0:.*]] = arith.constant 0 : index
//        CHECK:   %[[LOAD:.*]] = vector.load %[[A0]][%[[C]]]
//  CHECK-SAME:      : memref<16xf32>, vector<16xf32>
//        CHECK:   return %[[LOAD]] : vector<16x

func.func @maskedload_regression_1(
           %arg0: memref<16xf32>,
           %arg1: vector<16xf32>) -> vector<16x
  %c0 = arith.constant 0 : index

  %vec_i1 = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %arg0[%c0], %vec_i1, %arg1
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
}
```

Every test is a regression test – encode some _unique_ info instead!

Don’t repeat type in var name!



Case 2

```
// CHECK-LABEL:   func @maskedload_regression_2(
//  CHECK-SAME:   %[[A0:.*]]: memref<16xi8>,
//  CHECK-SAME:   %[[A1:.*]]: vector<16xi8>

//        CHECK:   %[[C0:.*]] = arith.constant 0 : index
//        CHECK:   %[[LOAD:.*]] = vector.load %[[A0]][%[[C]]]
//  CHECK-SAME:      : memref<16xi8>, vector<16xi8>
//        CHECK:   return %[[LOAD]] : vector<16xi8>

func.func @maskedload_regression_2(
           %arg0: memref<16xi8>,
           %arg1: vector<16xi8>) -> vector<16xi8> {
  %c0 = arith.constant 0 : index

  %vec_i1 = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %arg0[%c0], %vec_i1, %arg1
    : memref<16xi8>, vector<16xi1>, vector<16xi8>
      into vector<16xi8>

  return %ld : vector<16xi8>
}
```

Are these %arg0 and %arg1?

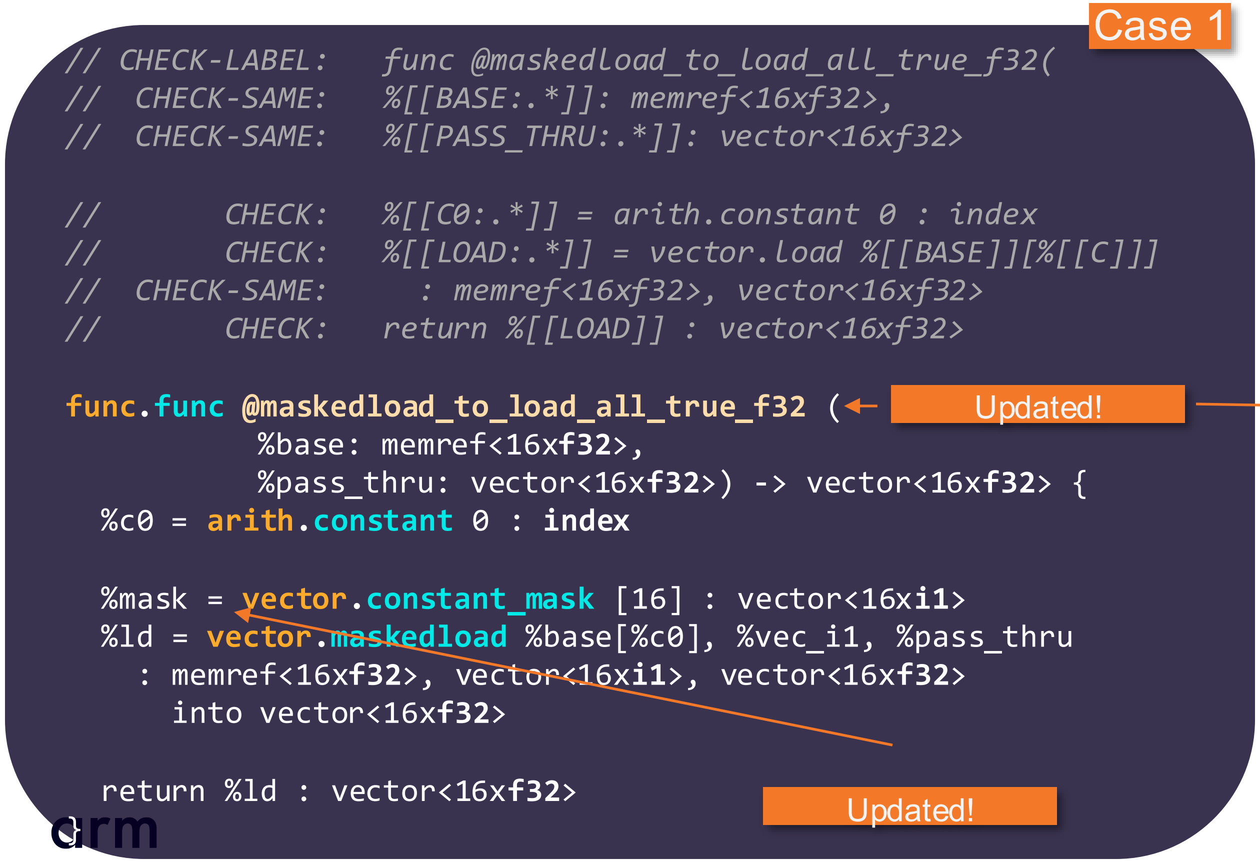
What’s %argN?

Seemingly “similar” tests

Missed opportunities to encode helpful info in tests.

- With new, ***consistent*** names, the ***<u>intent</u>*** becomes clear.
  - Testing `vector.maskedload` → `vector.load` folding when the mask is **ALL_TRUE**.
  - The only difference is **f32** vs **i8**.

Updated!

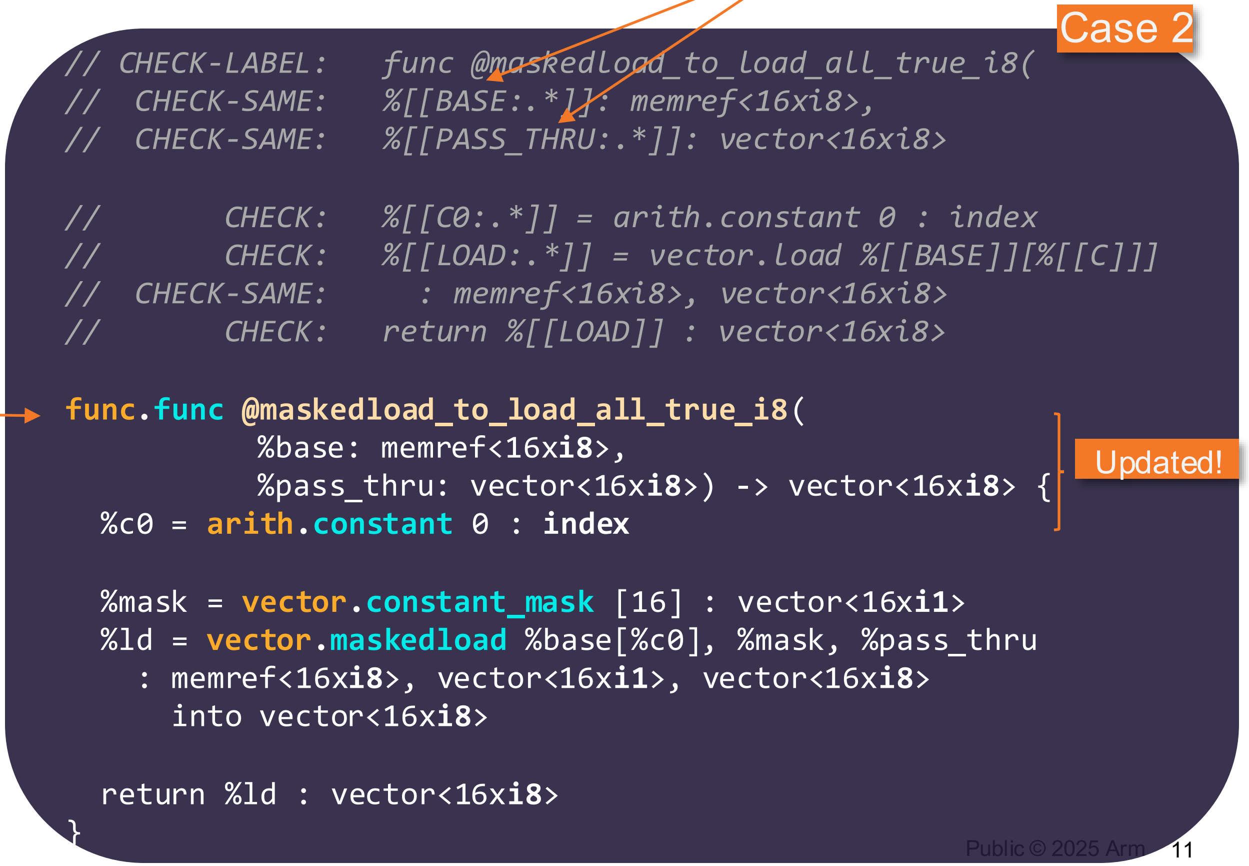


Case 1

```
// CHECK-LABEL:   func @maskedload_to_load_all_true_f32(
//  CHECK-SAME:   %[[BASE:.*]]: memref<16xf32>,
//  CHECK-SAME:   %[[PASS_THRU:.*]]: vector<16xf32>

//       CHECK:   %[[C0:.*]] = arith.constant 0 : index
//       CHECK:   %[[LOAD:.*]] = vector.load %[[BASE]][%[[C]]]
//  CHECK-SAME:     : memref<16xf32>, vector<16xf32>
//       CHECK:   return %[[LOAD]] : vector<16xf32>

func.func @maskedload_to_load_all_true_f32 (
          %base: memref<16xf32>,
          %pass_thru: vector<16xf32>) -> vector<16xf32> {
  %c0 = arith.constant 0 : index

  %mask = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %base[%c0], %vec_i1, %pass_thru
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
}
```

Updated!

Updated!



Case 2

```
// CHECK-LABEL:   func @maskedload_to_load_all_true_i8(
//  CHECK-SAME:   %[[BASE:.*]]: memref<16xi8>,
//  CHECK-SAME:   %[[PASS_THRU:.*]]: vector<16xi8>

//       CHECK:   %[[C0:.*]] = arith.constant 0 : index
//       CHECK:   %[[LOAD:.*]] = vector.load %[[BASE]][%[[C]]]
//  CHECK-SAME:     : memref<16xi8>, vector<16xi8>
//       CHECK:   return %[[LOAD]] : vector<16xi8>

func.func @maskedload_to_load_all_true_i8(
          %base: memref<16xi8>,
          %pass_thru: vector<16xi8>) -> vector<16xi8> {
  %c0 = arith.constant 0 : index

  %mask = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %base[%c0], %mask, %pass_thru
    : memref<16xi8>, vector<16xi1>, vector<16xi8>
      into vector<16xi8>

  return %ld : vector<16xi8>
}
```

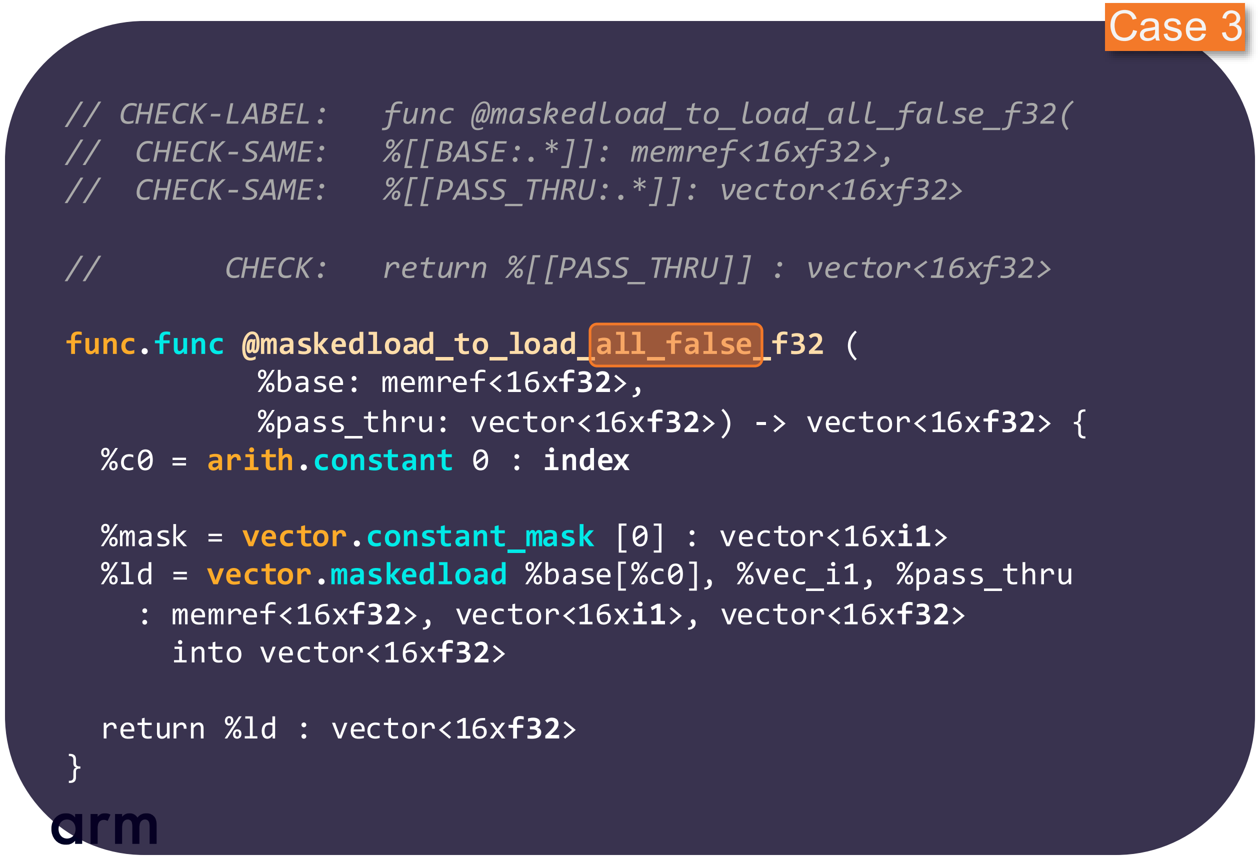
Updated!

Seemingly “similar” tests

Missed opportunities to encode helpful info in tests.

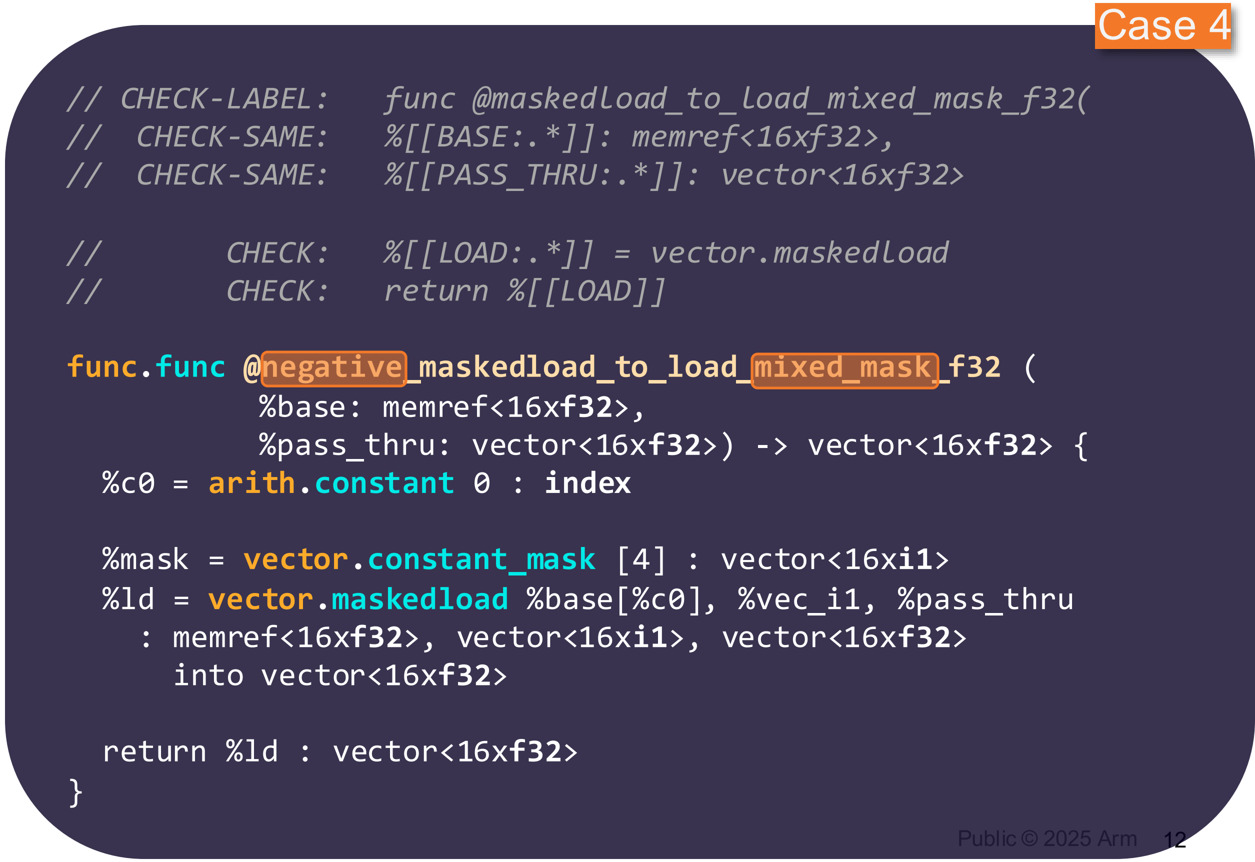
- With new, ***consistent*** names, the ***missing cases*** are obvious.
  - \`vector.maskedload\` → \`vector.load\` folding when the mask is **ALL_FALSE** or **MIXED**

Case 3



```
// CHECK-LABEL:   func @maskedload_to_load_all_false_f32(
//  CHECK-SAME:   %[[BASE:.*]]: memref<16xf32>,
//  CHECK-SAME:   %[[PASS_THRU:.*]]: vector<16xf32>

//       CHECK:   return %[[PASS_THRU]] : vector<16xf32>

func.func @maskedload_to_load_all_false_f32 (
            %base: memref<16xf32>,
            %pass_thru: vector<16xf32>) -> vector<16xf32> {
  %c0 = arith.constant 0 : index

  %mask = vector.constant_mask [0] : vector<16xi1>
  %ld = vector.maskedload %base[%c0], %vec_i1, %pass_thru
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
}
```

Case 4



```
// CHECK-LABEL:   func @maskedload_to_load_mixed_mask_f32(
//  CHECK-SAME:   %[[BASE:.*]]: memref<16xf32>,
//  CHECK-SAME:   %[[PASS_THRU:.*]]: vector<16xf32>

//       CHECK:   %[[LOAD:.*]] = vector.maskedload
//       CHECK:   return %[[LOAD]]

func.func @negative_maskedload_to_load_mixed_mask_f32 (
            %base: memref<16xf32>,
            %pass_thru: vector<16xf32>) -> vector<16xf32> {
  %c0 = arith.constant 0 : index

  %mask = vector.constant_mask [4] : vector<16xi1>
  %ld = vector.maskedload %base[%c0], %vec_i1, %pass_thru
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
}
```

Automation is available!

- Use **`generate-test-checks.py`,** but remember:
  - It is **not** authoritative about what constitutes a good test!
  - It won’t fix your input IR.
  - The generated CHECK lines usually require reviewing.

Case 1

```
func.func @maskedload_to_load_all_true_f32 (
          %base: memref<16xf32>,
          %pass_thru: vector<16xf32>) -> vector<16xf32> {
  %c0 = arith.constant 0 : index

  %mask = vector.constant_mask [16] : vector<16xi1>
  %ld = vector.maskedload %base[%c0], %vec_i1, %pass_thru
    : memref<16xf32>, vector<16xi1>, vector<16xf32>
      into vector<16xf32>

  return %ld : vector<16xf32>
```

```
mlir-opt -test-vector-to-vector-lowering
            | generate-test-checks.py
```

```
// CHECK-LABEL:   func.func @maskedload_to_load_all_true_f32(
// CHECK-SAME:      %[[ARG0:.*]]: memref<?xf32>,
// CHECK-SAME:      %[[ARG1:.*]]: vector<16xf32>) -> vector<16xf32> {
// CHECK:           %[[CONSTANT_0:.*]] = arith.constant 0 : index
// CHECK:           %[[LOAD_0:.*]] = vector.load %[[ARG0]]{{\[}}%[[CONSTANT_0]]] : memref<?xf32>, vector<16xf32>
// CHECK:           return %[[LOAD_0]] : vector<16xf32>
// CHECK:         }
```

arm

Call for action!

Leverage MLIR’s ability to capture high-level context when writing tests!

1. Follow the guideline – both when **contributing** and **reviewing** PRs.
2. Prioritize **“quality” over “quantity”** when writing tests!
3. Send PRs to improve existing tests!