MLIR QUERY TOOL FOR EASIER EXPLORATION OF THE IR

MLIR

Open Design Meeting

Devajith Valaparambil Sreeramaswamy

Agenda

- Introduction
- Capabilities of mlir-query
- Query example and implementation
- Function extraction and implementation
- Phabricator review
- Demo (if time permits)
- Suggestions/Questions

Introduction

- Interactive query tool for MLIR
- REPL interface for querying various properties of MLIR code
- Can assist in debugging and testing MLIR
- Standalone tool

Capabilities of mlir-query

- Find operations based on certain properties
- Find use-def up to N hops away
- Extract the matched operation/subgraph into a separate function

Capabilities of mlir-query

- mlir-query> match isConstant()
- Find use-def up to N hops away
- Extract the matched operation/subgraph into a separate function

Capabilities of mlir-query

- mlir-query> match isConstant()
- mlir-query> match getUses(isConstantOp(), 2)
- Extract the matched operation/subgraph into a separate function

Capabilities of mlir-query

- mlir-query> match isConstant()
- mlir-query> match getUses(isConstantOp(), 2)
- mlir-query> match getUses(isConstantOp(), 2).extract("foo")

Query: hasOpName

```
$ mlir-query basic-queries.mlir
mlir-query> m hasOpName("hello.japanese")
```

```
1 module {
2   func.func @basic_queries(%arg0: f32) -> f32 {
3     %c2_i32 = arith.constant 2 : i32
4     %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5     %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6     %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7     %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8     return %3 : f32
9   }
10 }
11
```

Query: hasOpName

```
$ mlir-query basic-queries.mlir
mlir-query> m hasOpName("hello.japanese")
```

```
1 module {
2   func.func @basic_queries(%arg0: f32) -> f32 {
3     %c2_i32 = arith.constant 2 : i32
4     %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5     %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6     %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7     %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8     return %3 : f32
9   }
10 }
11
```

Query: hasOpName

```
$ mlir-query basic-queries.mlir
mlir-query> m hasOpName("hello.japanese")

Match #1:

basic-queries.mlir:6:10: note: "root" binds here
%2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32

1 match.

mlir-query>
```

```
1  module {
2    func.func @basic_queries(%arg0: f32) -> f32 {
3      %c2_i32 = arith.constant 2 : i32
4      %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5      %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6      %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7      %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8      return %3 : f32
9    }
10 }
11
```

Matchers and mlir-query

- They are the building blocks for mlir-query.
- Can define custom matchers that can match any pattern.
- There’s a registry class that is responsible for storing and managing matchers.
- There’s a marshalling layer that wraps these matchers with different arguments and types (inspired from clang-query) for use by mlir-query.

Matcher.h

```
/// The matcher that matches operations that have the specified op name.
struct NameOpMatcher {
  NameOpMatcher(StringRef name) : name(name) {}
  bool match(Operation *op) const { return op->getName().getStringRef() == name; }

  StringRef name;
};

/// Matches a named operation.
NameOpMatcher m_Op(StringRef opName) {
  return NameOpMatcher(opName);
}
```

Registry.cpp

```
....
// Generate a registry map with all the known matchers.
RegistryMaps::RegistryMaps() {
  auto registerOpMatcher = [&](const std::string &name, auto matcher) {
    registerMatcher(name, internal::makeMatcherAutoMarshall(matcher, name));
  };

  registerOpMatcher("hasOpName", m_Op);
}
...
```

Registry.cpp

```
....
// Generate a registry map with all the known matchers.
RegistryMaps::RegistryMaps() {
  auto registerOpMatcher = [&](const std::string &name, auto matcher) {
    registerMatcher(name, internal::makeMatcherAutoMarshall(matcher, name));
  };

  registerOpMatcher("hasOpName", m_Op);
}
...
```

Query: isConstant

```
mlir-query> m isConstantOp()
```

```
1 module {
2   func.func @basic_queries(%arg0: f32) -> f32 {
3     %c2_i32 = arith.constant 2 : i32
4     %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5     %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6     %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7     %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8     return %3 : f32
9   }
10 }
11
```

Query: isConstant

```
mlir-query> m isConstantOp()
```

```
1 module {
2   func.func @basic_queries(%arg0: f32) -> f32 {
3     %c2_i32 = arith.constant 2 : i32
4     %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5     %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6     %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7     %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8     return %3 : f32
9   }
10 }
11
```

Query: isConstant

```
mlir-query> m isConstantOp()

Match #1:

basic-queries.mlir:3:15: note: "root" binds here
%c2_i32 = arith.constant 2 : i32

1 match.

mlir-query>
```

```
1 module {
2   func.func @basic_queries(%arg0: f32) -> f32 {
3     %c2_i32 = arith.constant 2 : i32
4     %0 = "hello.french"(%c2_i32) {bonjour = 1 : i32} : (i32) -> f32
5     %1 = "hello.english"(%c2_i32) {hello = 1 : i32} : (i32) -> f32
6     %2 = "hello.japanese"(%0, %1) {konnichiwa = 1 : i32} : (f32, f32) -> f32
7     %3 = "hello.spanish"(%1, %2) {hola = 1 : i32} : (f32, f32) -> f32
8     return %3 : f32
9   }
10 }
11
```

Matcher.h

```
/// The matcher that matches operations that have the `ConstantLike` trait.
struct ConstantOpMatcher {
  bool match(Operation *op) const { return op->hasTrait<OpTrait::ConstantLike>(); }
};

/// Matches a constant operation.
ConstantOpMatcher m_Constant() {
  return ConstantOpMatcher();
}
```

Registry.cpp

```
// Generate a registry map with all the known matchers.
RegistryMaps::RegistryMaps() {
  auto registerOpMatcher = [&](const std::string &name, auto matcher) {
    registerMatcher(name, internal::makeMatcherAutoMarshall(matcher, name));
  };

  registerOpMatcher("hasOpName", m_Op);
  registerOpMatcher("isConstant", m_Constant);
}
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

```
1 module {
2   func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3     %c1_i32 = arith.constant 1 : i32
4     "test.noop"() : () -> ()
5     %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6     %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7     %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8     %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9     %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11    %6 = "test.use_coo"(%5) : (i32) -> i32
12    return %6 : i32
13  }
14 }
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

```
1 module {
2   func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3     %c1_i32 = arith.constant 1 : i32
4     "test.noop"() : () -> ()
5     %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6     %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7     %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8     %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9     %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11    %6 = "test.use_coo"(%5) : (i32) -> i32
12    return %6 : i32
13  }
14 }
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

```
1  module {
2    func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3      %c1_i32 = arith.constant 1 : i32
4      "test.noop"() : () -> ()
5      %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6      %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7      %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8      %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9      %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10     %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11     %6 = "test.use_coo"(%5) : (i32) -> i32
12     return %6 : i32
13   }
14 }
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))

Match #1:

nested-queries.mlir:3:15: note: "root" binds here
%c1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
%0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
%4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

3 matches.

mlir-query>
```

```
1  module {
2    func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3      %c1_i32 = arith.constant 1 : i32
4      "test.noop"() : () -> ()
5      %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6      %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7      %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8      %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9      %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10     %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11     %6 = "test.use_coo"(%5) : (i32) -> i32
12     return %6 : i32
13   }
14 }
```

Matcher.h

```
struct DefinitionsMatcher {
  DefinitionsMatcher(Matcher innerMatcher) : innerMatcher(innerMatcher) {}

  bool match(Operation *op) const {
    return llvm::any_of(op->getUsers(), [&](Operation *childOp) {
      return innerMatcher.match(childOp);
    });
  }

  Matcher innerMatcher;
};

inline DefinitionsMatcher definedBy(Matcher innerMatcher) {
  return DefinitionsMatcher(innerMatcher);
}
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

```
1 module {
2   func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3     %c1_i32 = arith.constant 1 : i32
4     "test.noop"() : () -> ()
5     %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6     %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7     %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8     %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9     %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11    %6 = "test.use_coo"(%5) : (i32) -> i32
12    return %6 : i32
13  }
14 }
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))
```

```
1  module {
2    func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3      %c1_i32 = arith.constant 1 : i32
4      "test.noop"() : () -> ()
5      %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6      %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7      %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8      %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9      %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10     %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11     %6 = "test.use_coo"(%5) : (i32) -> i32
12     return %6 : i32
13   }
14 }
```

Query: definedBy

```
mlir-query> m definedBy(hasOpName("test.coo"))

Match #1:

nested-queries.mlir:3:15: note: "root" binds here
%c1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
%0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
%4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

3 matches.

mlir-query>
```

```
1  module {
2    func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
3      %c1_i32 = arith.constant 1 : i32
4      "test.noop"() : () -> ()
5      %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
6      %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
7      %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
8      %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
9      %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
10     %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
11     %6 = "test.use_coo"(%5) : (i32) -> i32
12     return %6 : i32
13   }
14 }
```

Potential solution

1) Memoization.
2) Pass extra arguments to match store resulting matches somewhere.

ASTMatchersInternal.h

```
...
template <typename T, typename DeclMatcherT>
class HasDeclarationMatcher : public MatcherInterface<T> {

  DynTypedMatcher InnerMatcher;

public:
  explicit HasDeclarationMatcher(const Matcher<Decl> &InnerMatcher)
      : InnerMatcher(InnerMatcher) {}

  bool matches(const T &Node, ASTMatchFinder *Finder,
               BoundNodesTreeBuilder *Builder) const override {
    return matchesSpecialized(Node, Finder, Builder);
  }
...
```

Registry.cpp

```
struct DefinitionsMatcher {
  DefinitionsMatcher(Matcher innerMatcher) : innerMatcher(innerMatcher) {}
  ...

  void matchBuilder(Operation *op, MatchFinder *finder, "#some_identifier") {
    if (innerMatcher.match(op)) {
      for (Value operand : op→getOperands()) {
        if (Operation *operandOp = operand.getDefiningOp()) {
          finder.addOperation(operandOp, "#some_identifier");
        }
      }
    }
  }

  Matcher innerMatcher;
};

inline DefinitionsMatcher definedBy(Matcher innerMatcher) {
  return DefinitionsMatcher(innerMatcher);
}
```

Registry.cpp

```
// Generate a registry map with all the known matchers.
RegistryMaps::RegistryMaps() {
  auto registerOpMatcher = [&](const std::string &name, auto matcher) {
    registerMatcher(name, internal::makeMatcherAutoMarshall(matcher, name));
  };

  registerOpMatcher("hasOpName", m_Op);
  registerOpMatcher("isConstant", m_Constant);
  registerOpMatcher("definedBy", definedBy);
}
```

Query: getAllDefinitions

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)
```

```
module {
  func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    "test.noop"() : () -> ()
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
    %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
    %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
    %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
    %6 = "test.use_coo"(%5) : (i32) -> i32
    return %6 : i32
  }
}
```

Query: getAllDefinitions

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
module {
  func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    "test.noop"() : () -> ()
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
    %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
    %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
    %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
    %6 = "test.use_coo"(%5) : (i32) -> i32
    return %6 : i32
  }
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2).extract("test")
```

```
module {
  func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    "test.noop"() : () -> ()
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
    %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
    %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
    %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
    %6 = "test.use_coo"(%5) : (i32) -> i32
    return %6 : i32
  }
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2).extract("test")


func.func @test(%arg0: i32, %arg1: i32, %arg2: i32, %arg3: i32) -> i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) -> i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) -> i32
  return %2 : i32
}


mlir-query>
```

```
module {
  func.func @foo(%arg0: i32, %arg1: i32, %arg2: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    "test.noop"() : () -> ()
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32
    %1:2 = "test.many_results"(%0) : (i32) -> (i32, i32)
    %2 = "test.unused_result"(%1#0, %1#1) : (i32, i32) -> i32
    %3 = "test.foo"(%c1_i32, %1#1) : (i32, i32) -> i32
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32
    %5 = "test.coo"(%4, %0, %c1_i32) : (i32, i32, i32) -> i32
    %6 = "test.use_coo"(%5) : (i32) -> i32
    return %6 : i32
  }
}
```

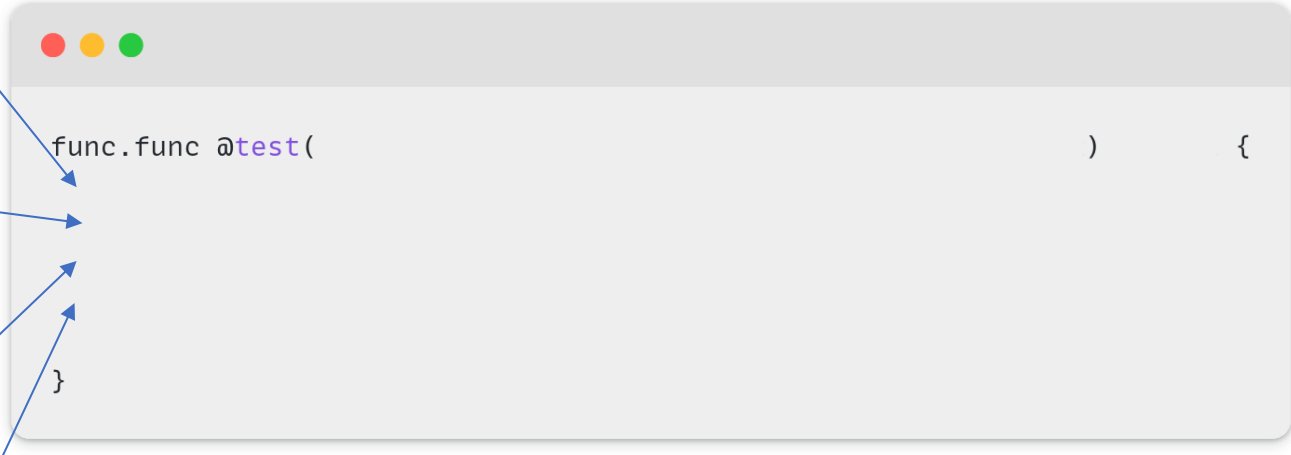
Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                                    )        {




}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```



Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                                    )        {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32

}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                                ) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                                ) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                        ) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(                                          ) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"                                    ->
i32

4 matches.
```

```
func.func @test(                                              ) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Function extraction

```
mlir-query> m getAllDefinitions(hasOpName("test.use_coo"), 2)

Match #1:

nested-queries.mlir:3:16: note: "root" binds here
    %cr1_i32 = arith.constant 1 : i32

Match #2:

nested-queries.mlir:5:10: note: "root" binds here
    %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) -> i32

Match #3:

nested-queries.mlir:9:10: note: "root" binds here
    %4 = "test.boo"(%1#0, %3) : (i32, i32) -> i32

Match #4:

nested-queries.mlir:10:10: note: "root" binds here
    %5 = "test.coo"(%4, %0, %cr1_i32) : (i32, i32, i32) ->
i32

4 matches.
```

```
func.func @test(%arg0: i32, %arg1: i32, %arg2: i32, %arg3: i32) → i32 {
  %c1_i32 = arith.constant 1 : i32
  %0 = "test.one_result"(%arg0, %arg1) : (i32, i32) → i32
  %1 = "test.boo"(%arg2, %arg3) : (i32, i32) → i32
  %2 = "test.coo"(%1, %0, %c1_i32) : (i32, i32, i32) → i32
  return %2 : i32
}
```

Initial Phabricator review

- Opened an initial review: https://reviews.llvm.org/D155127
- Parser, Diagnostics taking up a huge chunk of the diff and that won't be changing much
- Supports basic queries, more changes/refractoring of the matchers can happen at a later stage

QUESTIONS