Distinct Attributes

Modeling LLVM's distinct metadata in MLIR
Motivating Use Case

```
define void @copy(ptr noalias %0, ptr noalias %1) {
    %2 = load float, ptr %0
    store float %2, ptr %1
    ret void
}
```

```
define void @main(ptr %0, ptr %1) {
    call void @copy(ptr %0, ptr %1)
    ret void
}
```

"restrict" pointers

Inlining creates aliasing metadata
Motivating Use Case

define void @main(ptr %0, ptr %1) {
  %tmp.i = load float, ptr %0, !alias.scope !0, !noalias !1
  store float %tmp.i, ptr %1, !alias.scope !1, !noalias !0
  ret void
}

inlined copy function

unique metadata

!0 = !{!3}
!1 = !{!4}

distinct metadata

!2 = distinct {!"copy"}
!3 = distinct {!2, !"copy: argument 0"}
!4 = distinct {!2, !"copy: argument 1"}

distinct metadata

domain

scope
define void @main(ptr %0, ptr %1) {
  %tmp.i = load float, ptr %0, !alias.scope !0, !noalias !1
  store float %tmp.i, ptr %1, !alias.scope !1, !noalias !0
  %tmp.i1 = load float, ptr %0, !alias.scope !2, !noalias !3
  store float %tmp.i1, ptr %1, !alias.scope !3, !noalias !2
  ret void
}

...
Modeling Metadata in MLIR

- Module-level metadata
  - Distinct and unique nodes
  - Structured data
  - E.g., Alias, AccessGroup, Loop, Debug metadata

- Parallel metadata manipulation
  - Thread-safe metadata creation
  - Deterministic metadata creation
  - E.g., during inlining or lowering from high-level dialects

attributes are a good match except for distinctness
**Op Based Representation**

```llvm
llvm.metadata __global_metadata {
  llvm.alias_scope_domain @domain {
    description = "copy"
  }
  llvm.alias_scope @scope {
    description = "copy: argument 0",
    domain = @domain
  }
  ...
}

llvm.func @main(%arg0: !llvm.ptr, %arg1: !llvm.ptr) {
  %0 = llvm.load %arg0 {
    alias_scopes = [@__global_metadata::@scope],
    ...
  }
  ...
  llvm.return
}
```

- **one global metadata operation**
- **ops carry metadata**
- **symbol references**
- **Debug and loop metadata uses attributes instead**
Limitations of the Op Based Representation

- Sequential creation of metadata operations
  - Inlining requires parallel metadata creation

- Symbol references are not "type-safe"
  - C++ verifiers
  - Limited composability with metadata attributes
Solution 1: Sequence Attribute

- Extend distinct attributes with a unique identifier to avoid uniquing
- Use a mutable attribute to generate identifiers in a thread-safe way
- Use a mutable attribute per function to avoid non-determinism

```cpp
#sequence = #llvm.distinct_sequence< scope = @main, state = 2>

#domain0 = #llvm.alias_domain<
    id = 0, elem_of = #sequence,
    description = "copy"
>
#domain1 = #llvm.alias_domain<
    id = 1, elem_of = #sequence,
    description = "copy"
>
```

- generator attribute with mutable state
- extra data to avoid uniquing
- actual data
Solution 1: Sequence Attribute

- Query the distinct sequence attribute for the next identifier
- Use the identifier and the sequence to prevent uniquing

```cpp
auto distinctSequence = DistinctSequenceAttr::get(
    SymbolRefAttr::get(StringAttr::get(context, funcName)));

auto aliasDomain = AliasDomainAttr::get(context,
    distinctSequence.getNextID(), distinctSequence,
    "copy");
```

get the sequence

query the next identifier and increment the sequence state
**Solution 1: Sequence Attribute**

Pros:
- Low implementation complexity

Cons:
- An LLVM Dialect-specific solution
- Leaks implementation details
- Unintuitive behavior if metadata is copied to another function

Neutral:
- Does not require MLIR core infrastructure changes
Solution 2: Distinct Attributes

- Add support for distinct attributes to MLIR core
- Update StorageUniquer to support distinct attributes based on a trait
- Use the pointer value instead of an identifier to model distinctness
- Print and parse a distinct identifiers according to program order

```llvm
#domain0 = distinct<0 = #llvm.alias_domain<description = "copy">>
#domain1 = distinct<1 = #llvm.alias_domain<description = "copy">>
```

distinct identifier generated during printing

distinct attributes compose with other attributes
Solution 2: Distinct Attributes

- Distinct attributes can be defined in `tablegen`
- A trait marks them as distinct

```python
def LLVM_AliasDomainAttr : LLVM_Attr<
    "AliasDomain", "alias_domain", [ IsDistinct]> {
    let parameters = (ins "StringAttr":$description);
    let assemblyFormat = "`<` struct(params) `>```; 
}
```
Solution 2: Distinct Attributes

- Use the existing StorageUniquer but overload isEqual to return false
- Use an atomic counter to avoid hash table collisions

```cpp
auto attr1 = AliasDomainAttr::get(context, "copy");
auto attr2 = AliasDomainAttr::get(context, "copy");
assert(attr1 != attr2);
```

distribute a new attribute every time
Solution 2: Distinct Attributes

- Print and parse a distinct ids according to program order

```cpp
#domain1 = distinct<0 = #llvm.alias_domain<description = "copy">>
#domain2 = distinct<1 = #llvm.alias_domain<description = "copy">>

#scope1 = distinct<2 = #llvm.alias_scope<
    description = "copy: argument 0", domain = #domain1>
>
```

composes with other attributes

```cpp
%0 = llvm.load %arg0 { alias_scopes = [#scope1], ... }
```
Solution 2: Distinct Attributes

- Print and parse a distinct ids according to program order

```c
%0 = llvm.load %arg0 { alias_scopes = [
  ...
  distinct<0 = #llvm.alias_scope<
    description = "copy: argument 0",
    domain = distinct<1 = <description = "copy">>
  >
  ...
]}`
**Solution 2: Distinct Attributes**

**Pros:**
- Storage efficient and concise representation of distinct attributes
- Generic solution not limited to LLVM dialect
- Tablegen works out-of-the box

**Cons:**
- Medium/high implementation complexity

**Neutral:**
- Requires MLIR core infrastructure changes
Thread-Safety

- Both solutions rely on attributes
  - Attributes are thread-safe
  - Attribute mutation is also thread-safe
  - Attributes have a global scope

- Solution 1 & Solution 2: Are thread-safe thanks to attributes
**Determinism**

- Parallel processing is non-deterministic by nature

- Solution 1: Generates unique identifiers per function
  - Scope the distinct identifier generation
  - Every function is processed sequentially

- Solution 2: Generates unique identifiers when printing
  - Sequentialize the distinct identifier generation
  - Use the position in the IR to generate the identifiers
Solution 3: Properties?

- Could properties be used to model distinct metadata
  - How can we ensure composability with other attributes?
  - Print an identifier based on program order?
  - Is there a way to print aliases at the beginning of the module?

```python
distinct<0 = #llvm.alias_scope<
  description = "copy: argument 0",
  domain = distinct<1 = <description = "copy">
>
```
Discussion

Requirements reminder:

- Module-level metadata
- Distinct and unique nodes
- Structured data / composable
- Thread-safe metadata creation
- Deterministic metadata creation