MLIR Properties

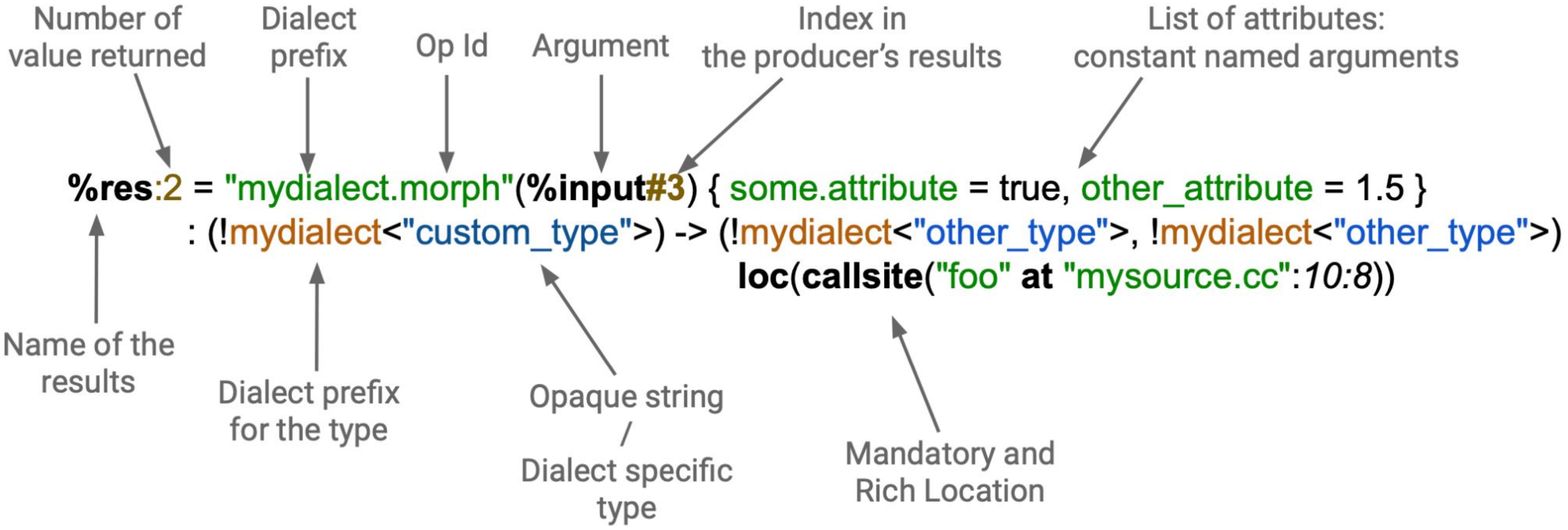
1. Operation Implementation Deep Dive
2. Attributes, Accessors, and ODS APIs
3. Properties

Mehdi Amini - 2/9/2023

Operation Implementation

Deep Dive

MLIR 101:



\+ A list of regions…

Operation storage

```
class alignas(8) Operation final
   : public llvm::ilist node with parent<Operation, Block>,
     private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                   BlockOperand, Region, OpOperand> {
```

Operation storage

```
class alignas(8) Operation final
    : public llvm::ilist_node_with_parent<Operation, Block>,
      private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                    BlockOperand, Region, OpOperand> {
```

```
      0 | class mlir::Operation [sizeof=64, dsize=64, align=8, nvsize=64, nvalign=8]
      0 |   class llvm::ilist_node_with_parent<class mlir::Operation, class mlir::Block> (base)
      0 |       class llvm::PointerIntPair<class llvm::ilist_node_base<true> *, 1> PrevAndSentinel
      8 |       class llvm::ilist_node_base<true> * Next
      0 |   class llvm::TrailingObjects<class mlir::Operation,...> (base) (empty)
     16 |   class mlir::Block * block
     24 |   class mlir::Location location
     32 |   unsigned int orderIndex
     36 |   const unsigned int numResults
     40 |   const unsigned int numSuccs
44:0-30 |   const unsigned int numRegions
 47:7-7 |   _Bool hasOperandStorage
     48 |   class mlir::OperationName name
     56 |   class mlir::DictionaryAttr attrs
```

Add *-XClang -fdump-record-layouts* to any clang invocation to get this information.

Operation storage

```
class alignas(8) Operation final
    : public llvm::ilist_node_with_parent<Operation, Block>,
      private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                    BlockOperand, Region, OpOperand> {
```

```
      0 | class mlir::Operation [sizeof=64, dsize=64, align=8, nvsize=64, nvalign=8]
      0 |   class llvm::ilist_node_with_parent<class mlir::Operation, class mlir::Block> (base)
      0 |         class llvm::PointerIntPair<class llvm::ilist_node_base<true> *, 1> PrevAndSentinel
      8 |         class llvm::ilist_node_base<true> * Next
      0 |   class llvm::TrailingObjects<class mlir::Operation,...> (base) (empty)
     16 |   class mlir::Block * block
     24 |   class mlir::Location location
     32 |   unsigned int orderIndex
     36 |   const unsigned int numResults
     40 |   const unsigned int numSuccs
44:0-30 |   const unsigned int numRegions
 47:7-7 |   _Bool hasOperandStorage
     48 |   class mlir::OperationName name
     56 |   class mlir::DictionaryAttr attrs
```

Operations are stored in a doubly-linked list, these are pointers to prev and next in the current bock.

Parent block (if any)

Order in the current block

Add *-XClang -fdump-record-layouts* to any clang invocation to get this information.

Operation storage

```
class alignas(8) Operation final
    : public llvm::ilist_node_with_parent<Operation, Block>,
      private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                    BlockOperand, Region, OpOperand> {
```

```
       0 | class mlir::Operation [sizeof=64, dsize=64, align=8, nvsize=64, nvalign=8]
       0 |   class llvm::ilist_node_with_parent<class mlir::Operation, class mlir::Block> (base)
       0 |         class llvm::PointerIntPair<class llvm::ilist_node_base<true> *, 1> PrevAndSentinel
       8 |         class llvm::ilist_node_base<true> * Next
       0 |   class llvm::TrailingObjects<class mlir::Operation,...> (base) (empty)
      16 |   class mlir::Block * block
      24 |   class mlir::Location location
      32 |   unsigned int orderIndex
      36 |   const unsigned int numResults
      40 |   const unsigned int numSuccs
 44:0-30 |   const unsigned int numRegions
  47:7-7 |   _Bool hasOperandStorage
      48 |   class mlir::OperationName name
      56 |   class mlir::DictionaryAttr attrs
```

Operations are stored in a doubly-linked list, these are pointers to prev and next in the current bock.

Parent block (if any)

Order in the current block

Where are the lists of Operands? Regions? Successor block operands?

Operation storage: TrailingObjects

```
class alignas(8) Operation final
    : public llvm::ilist_node_with_parent<Operation, Block>,
      private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                    BlockOperand, Region, OpOperand> {
```

```
        0 | class mlir::Operation [sizeof=64, dsize=64, align=8, nvsize=64, nvalign=8]
        0 |   class llvm::ilist_node_with_parent<class mlir::Operation, class mlir::Block> (base)
        0 |       class llvm::PointerIntPair<class llvm::ilist_node_base<true> *, 1> PrevAndSentinel
        8 |       class llvm::ilist_node_base<true> * Next
        0 |   class llvm::TrailingObjects<class mlir::Operation,...> (base) (empty)
       16 |   class mlir::Block * block
       24 |   class mlir::Location location
       32 |   unsigned int orderIndex
       36 |   const unsigned int numResults
       40 |   const unsigned int numSuccs
  44:0-30 |   const unsigned int numRegions
   47:7-7 |   _Bool hasOperandStorage
       48 |   class mlir::OperationName name
       56 |   class mlir::DictionaryAttr attrs
```

Where are the lists of Operands?
Regions? Successor block operands?

Operation storage: TrailingObjects

```
class alignas(8) Operation final
    : public llvm::ilist_node_with_parent<Operation, Block>,
      private llvm::TrailingObjects<Operation, detail::OperandStorage,
                                    BlockOperand, Region, OpOperand> {
```

Concept: malloc more than sizeof(Operation) to pack extra data in the same allocation.

Example: an operation with two regions.

```
        0 | class mlir::Operation
          |   …
       36 |   const unsigned int numResults = 0
       40 |   const unsigned int numSuccs = 0
  44:0-30 |   const unsigned int numRegions = 2
   47:7-7 |   _Bool hasOperandStorage = false
       48 |   class mlir::OperationName name
       56 |   class mlir::DictionaryAttr attrs
       64 |   Region [size=24]
       88 |   Region [size=24]
```

Malloc size = 112B

Operation storage: TrailingObjects

```
llvm::TrailingObjects<Operation, detail::OperandStorage, BlockOperand, Region, OpOperand> {
```

Example: an operation with two regions, two successors blocks, and 3 operands.

```
       0 | class mlir::Operation
         |   …
      36 |   const unsigned int numResults = 0
      40 |   const unsigned int numSuccs = 2
 44:0-30 |   const unsigned int numRegions = 2
  47:7-7 |   _Bool hasOperandStorage = true
      48 |   class mlir::OperationName name
      56 |   class mlir::DictionaryAttr attrs
      64 |   OperandStorage [size=16]
      80 |   BlockOperand [size=24]
     104 |   BlockOperand [size=24]
     128 |   Region [size=24]
     152 |   Region [size=24]
     176 |   OpOperand [size=16]
     192 |   OpOperand [size=16]
     208 |   OpOperand [size=16]
```

Malloc size = 224B

Operation storage: TrailingObjects

```
llvm::TrailingObjects<Operation, detail::OperandStorage, BlockOperand, Region, OpOperand> {
```

Example: an operation with two regions, two successors blocks, and 3 operands.

```
      0 | class mlir::Operation
        |   …
     36 |   const unsigned int numResults = 0
     40 |   const unsigned int numSuccs = 2
44:0-30 |   const unsigned int numRegions = 2
 47:7-7 |   _Bool hasOperandStorage = true
     48 |   class mlir::OperationName name
     56 |   class mlir::DictionaryAttr attrs
     64 |   OperandStorage [size=16]
     80 |   BlockOperand [size=24]
    104 |   BlockOperand [size=24]
    128 |   Region [size=24]
    152 |   Region [size=24]
    176 |   OpOperand [size=16]
    192 |   OpOperand [size=16]
    208 |   OpOperand [size=16]
```

The accessor:

```
Region *region = getRegion(1);
```

Operation storage: TrailingObjects

```
llvm::TrailingObjects<Operation, detail::OperandStorage, BlockOperand, Region, OpOperand> {
```

Example: an operation with two regions, two successors blocks, and 3 operands.

```
      0 | class mlir::Operation
        |   ...
     36 |   const unsigned int numResults = 0
     40 |   const unsigned int numSuccs = 2
44:0-30 |   const unsigned int numRegions = 2
 47:7-7 |   _Bool hasOperandStorage = true
     48 |   class mlir::OperationName name
     56 |   class mlir::DictionaryAttr attrs
     64 |   OperandStorage [size=16]
     80 |   BlockOperand [size=24]
    104 |   BlockOperand [size=24]
    128 |   Region [size=24]
    152 |   Region [size=24]
    176 |   OpOperand [size=16]
    192 |   OpOperand [size=16]
    208 |   OpOperand [size=16]
```

The accessor:

```
Region *region = getRegion(1);
```

Is implemented as:

```
auto *ptr = reinterpret_cast<char *>(this);
ptr += sizeof(Operation); // 64
ptr += sizeof(OperandStorage); // 16
ptr += 2 * sizeof(BlockOperand); // 2*24
auto *reg = reinterpret_cast<Region*>(ptr);
return &regions[1];
```

Operation storage: `OpOperands`

`OpOperandStorage` describes the storage of the operands: either tail-allocated or separated.

This allows for dynamic resizing of the operands “in-place”.

```
      0 | class mlir::Operation
        |   ...
     36 |   const unsigned int numResults = 0
     40 |   const unsigned int numSuccs = 2
44:0-30 |   const unsigned int numRegions = 2
 47:7-7 |   _Bool hasOperandStorage = true
     48 |   class mlir::OperationName name
     56 |   class mlir::DictionaryAttr attrs
     64 |   OperandStorage [size=16]
     80 |   BlockOperand [size=24]
    104 |   BlockOperand [size=24]
    128 |   Region [size=24]
    152 |   Region [size=24]
    176 |   OpOperand [size=16]
    192 |   OpOperand [size=16]
    208 |   OpOperand [size=16]
```

```cpp
class alignas(8) OperandStorage {
public:
  ...
private:
 /// Total capacity that the storage can hold.
 unsigned capacity : 31;
 /// Indicate if the storage was dyn-allocated
 /// as opposed to inlined into the operation.
 unsigned isStorageDynamic : 1;
 /// Number of operands within the storage.
 unsigned numOperands;
 /// A pointer to the operand storage.
 OpOperand *operandStorage;
```

Initial “capacity”, like in `SmallVector<OpOperand, 3>`

Operation storage: `Results`

They don't appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

```
    -144 |   OutOfLineOpResult [size=24]
    -120 |   OutOfLineOpResult [size=24]
     -96 |   InlineOpResult [size=16]
     -80 |   InlineOpResult [size=16]
     -64 |   InlineOpResult [size=16]
     -48 |   InlineOpResult [size=16]
     -32 |   InlineOpResult [size=16]
     -16 |   InlineOpResult [size=16]
       0 | class mlir::Operation
         |   …
      36 |   const unsigned int numResults = 8
      40 |   const unsigned int numSuccs = 0
 44:0-30 |   const unsigned int numRegions = 2
  47:7-7 |   _Bool hasOperandStorage = true
      48 |   class mlir::OperationName name
```

This is why you can't add/remove results, regions, and block successors to an *Operation*: you must create a new one!

Operation storage: `Results`

They don't appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

```
    -144 |   OutOfLineOpResult [size=24]
    -120 |   OutOfLineOpResult [size=24]
     -96 |   InlineOpResult [size=16]
     -80 |   InlineOpResult [size=16]
     -64 |   InlineOpResult [size=16]
     -48 |   InlineOpResult [size=16]
     -32 |   InlineOpResult [size=16]
     -16 |   InlineOpResult [size=16]
       0 | class mlir::Operation
         |   ...
      36 |   const unsigned int numResults = 8
      40 |   const unsigned int numSuccs = 0
 44:0-30 |   const unsigned int numRegions = 2
  47:7-7 |   _Bool hasOperandStorage = true
      48 |   class mlir::OperationName name
```

```
class OutOfLineOpResult {
 detail::IROperandBase *firstUse;
 class llvm::PointerIntPair<mlir::Type, 3,
       detail::ValueImpl::Kind> typeAndKind;
 int64_t outOfLineIndex;
};
```

```
class InlineOpResult {
 detail::IROperandBase *firstUse;
 llvm::PointerIntPair<mlir::Type, 3,
       detail::ValueImpl::Kind> typeAndKind;
};
```

Operation storage: `Results`

They don’t appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

```
    -144 |   OutOfLineOpResult [size=24]
    -120 |   OutOfLineOpResult [size=24]
     -96 |   InlineOpResult [size=16]
     -80 |   InlineOpResult [size=16]
     -64 |   InlineOpResult [size=16]
     -48 |   InlineOpResult [size=16]
     -32 |   InlineOpResult [size=16]
     -16 |   InlineOpResult [size=16]
       0 | class mlir::Operation
         |   …
      36 |   const unsigned int numResults = 8
      40 |   const unsigned int numSuccs = 0
 44:0-30 |   const unsigned int numRegions = 2
  47:7-7 |   _Bool hasOperandStorage = true
      48 |   class mlir::OperationName name
```

```
class OutOfLineOpResult {
 detail::IROperandBase *firstUse;
 class llvm::PointerIntPair<mlir::Type, 3,
        detail::ValueImpl::Kind> typeAndKind;
 int64_t outOfLineIndex;
};
```

Get back to the Operation* pointer from the result itself

```
class InlineOpResult {
 detail::IROperandBase *firstUse;
 llvm::PointerIntPair<mlir::Type, 3,
        detail::ValueImpl::Kind> typeAndKind;
};
```

3 bits stolen from the Type, enough to count to 6!

Operation storage: `Results`

They don't appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

```
 -144 |   OutOfLineOpResult [size=24]
 -120 |   OutOfLineOpResult [size=24]
  -96 |   InlineOpResult [size=16]
  -80 |   InlineOpResult [size=16]
  -64 |   InlineOpResult [size=16]
  -48 |   InlineOpResult [size=16]
  -32 |   InlineOpResult [size=16]
  -16 |   InlineOpResult [size=16]
    0 | class mlir::Operation
      |   ...
   36 |   const unsigned int numResults = 8
   40 |   const unsigned int numSuccs = 0
44:0-30 | const unsigned int numRegions = 2
47:7-7 |  _Bool hasOperandStorage = true
   48 |   class mlir::OperationName name
```

```
class OutOfLineOpResult {
 detail::IROperandBase *firstUse;
 class llvm::PointerIntPair<mlir::Type, 3,
        detail::ValueImpl::Kind> typeAndKind;
 int64_t outOfLineIndex;
};
```

Get back to the Operation* pointer from the result itself

```
class InlineOpResult {
 detail::IROperandBase *firstUse;
 llvm::PointerIntPair<mlir::Type, 3,
        detail::ValueImpl::Kind> typeAndKind;
};
```

3 bits stolen from the Type, enough to count to 6!

Operation storage: `Results`

They don't appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

-144 | OutOfLineOpResult **[size=2**
-120 | OutOfLineOpResult **[size=2**
-96 | InlineOpResult **[size=16]**
-80 | InlineOpResult **[size=16]**
-64 | InlineOpResult **[size=16]**
-48 | InlineOpResult **[size=16]**
-32 | InlineOpResult **[size=16]**
-16 | InlineOpResult **[size=16]**
0 | class mlir::Operation
| …
36 | const unsigned int numResu
40 | const unsigned int numSucc
44:0-30 | const unsigned int numRegi
47:7-7 | _Bool hasOperandStorage =
48 | class mlir::OperationName n

```
OpResult getResult(unsigned idx) {
  const int maxInlineResults= 6;
  auto *inlinePtr =
    reinterpret_cast<InlineOpResult *>(this);
  if (idx < maxInlineResults) {
    inlinePtr -= idx + 1;
    return OpResult(inlinePtr);
  }
  inlinePtr -= maxInlineResults;
  idx -= maxInlineResults;
  auto *outOfLinePtr =
    reinterpret_cast<OutOfLineOpResult *>(inlinePtr);
  outOfLinePtr -= idx + 1;
  return OpResult(outOfLinePtr);
}
```

Operation storage: `Results`

They don't appear in the `llvm::TrailingObjects` list: we allocate them **before** the `Operation`!

Example: Operation with 8 results:

```
-144 |   OutOfLineOpResult [size=2
-120 |   OutOfLineOpResult [size=2
 -96 |   InlineOpResult [size=16]
 -80 |   InlineOpResult [size=16]
 -64 |   InlineOpResult [size=16]
 -48 |   InlineOpResult [size=16]
 -32 |   InlineOpResult [size=16]
 -16 |   InlineOpResult [size=16]
   0 | class mlir::Operation
     |   ...
  36 |   const unsigned int numResu
  40 |   const unsigned int numSucc
44:0-30 | const unsigned int numRegi
47:7-7 | _Bool hasOperandStorage =
  48 |   class mlir::OperationName n
```

```
OpResult getResult(unsigned idx) {
  const int maxInlineResults= 6;
  auto *inlinePtr =
    reinterpret_cast<InlineOpResult *>(this);
  if (idx < maxInlineResults) {
    inlinePtr -= idx + 1;
    return OpResult(inlinePtr);
  }
  inlinePtr -= maxInlineResults;
  idx -= maxInlineResults;
  auto *outOfLinePtr =
    reinterpret_cast<OutOfLineOpResult *>(inlinePtr);
  outOfLinePtr -= idx + 1;
  return OpResult(outOfLinePtr);
}
```

Negative offset from Operation*

Attributes, Operation Accessors,
and ODS APIs

Attributes: recap

From language reference:

The top-level **attribute dictionary attached to an operation** has special semantics. The attribute entries are considered to be of two different kinds based on whether their dictionary key has a dialect prefix:

- ***inherent attributes*** are inherent to the definition of an operation's semantics. The operation itself is expected to verify the consistency of these attributes. An example is the `predicate` attribute of the `arith.cmpi` op. These attributes must have names that do not start with a dialect prefix.
- ***discardable attributes*** have semantics defined externally to the operation itself, but must be compatible with the operation's semantics. These attributes must have names that start with a dialect prefix. The dialect indicated by the dialect prefix is expected to verify these attributes. An example is the `gpu.container_module` attribute.

Attributes

```
def Arith_CmpIOp
 : Arith_CompareOpOfAnyRank<"cmpi"> {
 let summary = "integer comparison operation";
 let arguments = (ins Arith_CmpIPredicateAttr:$predicate,
                      SignlessIntegerLikeOfAnyRank:$lhs,
                      SignlessIntegerLikeOfAnyRank:$rhs);
}
// Custom form of scalar "signed less than" comparison.
%x = arith.cmpi slt, %lhs, %rhs : i32
// Generic form of the same operation.
%x = "arith.cmpi"(%lhs, %rhs) {predicate = 2 : i64} : (i32, i32) -> i1

module attributes {
 gpu.container_module,
 spirv.target_env = #spirv.target_env<#spirv.vce<v1.0, [Kernel, Addresses]>,
                                      #spirv.resource_limits<>>
```

Inherent Attribute

Discardable Attributes

Attributes

Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```

```
array<i64: 1, 2, 3, 4>
```

Attributes

Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```

```
array<i64: 1, 2, 3, 4>
```

```
0 | class mlir::detail::DenseArrayAttrImpl<int64_t>
0 |   class mlir::DenseArrayAttr (base)
0 |     class mlir::detail::StorageUserBase<class mlir::DenseArrayAttr,
0 |                                 class mlir::Attribute, struct mlir::detail::DenseArrayAttrStorage,
0 |                                 class mlir::detail::AttributeUniquer> (base)
0 |       class mlir::Attribute (base)
0 |         mlir::Attribute::ImplType * impl
  | [sizeof=8, dsize=8, align=8, nvsize=8, nvalign=8]
```

An *Attribute* (like an instance of `DenseI64ArrayAttr`) contains just a pointer. It should be treated as a pointer!

But a pointer to what?

Attributes

Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```

```
array<i64: 1, 2, 3, 4>
```

```
0 | class mlir::detail::DenseArrayAttrImpl<int64_t>
0 |   class mlir::DenseArrayAttr (base)
0 |     class mlir::detail::StorageUserBase<class mlir::DenseArrayAttr,
0 |                                         class mlir::Attribute, struct mlir::detail::DenseArrayAttrStorage,
0 |                                         class mlir::detail::AttributeUniquer> (base)
0 |       class mlir::Attribute (base)
0 |         mlir::Attribute::ImplType * impl
  | [sizeof=8, dsize=8, align=8, nvsize=8, nvalign=8]
```

An *Attribute* (like an instance of `DenseI64ArrayAttr`) contains just a pointer. It should be treated as a pointer!

But a pointer to what?

Attributes

Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```

```
struct DenseArrayAttrStorage : public ::mlir::AttributeStorage {
 using KeyTy = std::tuple<Type, int64_t, ::llvm::ArrayRef<char>>;
 DenseArrayAttrStorage(Type elementType, int64_t size,
                       ::llvm::ArrayRef<char> rawData)
    : elementType(elementType), size(size), rawData(rawData) {}
 Type elementType; /// Type of the element, for example here `i64`
 int64_t size;     /// Number of elements
 ::llvm::ArrayRef<char> rawData; /// Content of the array
 ///  ...
};
```

Attributes are pointing to a corresponding “Storage” object
(and wrapping this with a “nice” API)

Attributes

```
class MLIRContextImpl {
 // Attribute uniquing
 DenseMap<TypeID, AbstractAttribute *> registeredAttributes;
 StorageUniquer attributeUniquer;
 …
```

Informations about an Attribute class, like access to AttributeInterfaces for example.

Storage for all Attributes!

Attributes

```
class MLIRContextImpl {
 // Attribute uniquing
 DenseMap<TypeID, AbstractAttribute *> registeredAttributes;
 StorageUniquer attributeUniquer;
 …

/// This is the implementation of the StorageUniquer class.
struct StorageUniquerImpl {
 ...
 /// Map of TypeIDs to the storage uniquer to use for registered objects.
 DenseMap<TypeID, std::unique_ptr<ParametricStorageUniquer>>
     parametricUniquers;
```

Informations about an Attribute class, like access to AttributeInterfaces for example.

Storage for all Attributes!

Unique ID for storage for classes like DenseArrayAttr, StringAttr, IntegerAttr, YourCustomAttr, ….

When loading a dialect in the context, this map is populated with the “uniquer” for each attribute class.

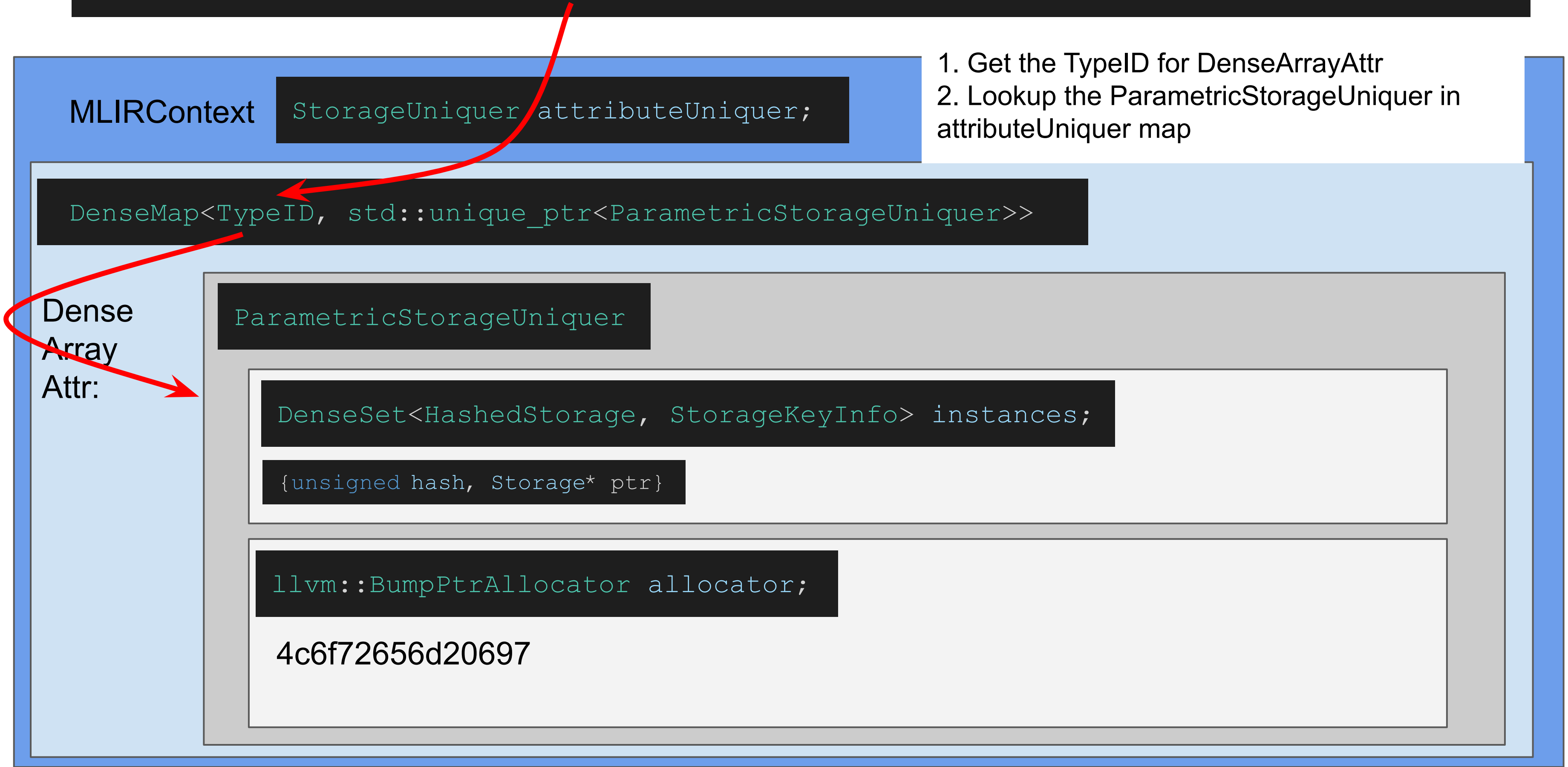
```
class ParametricStorageUniquer {
 /// Simplified view below

   /// The set containing the allocated storage instances.
   DenseSet<HashedStorage, StorageKeyInfo> instances;
   /// Allocator to use when constructing derived instances.
   StorageAllocator allocator;
```

```
 /// Utility allocator to allocate memory for instances of attributes
 class StorageAllocator {
   template <typename T>
   ArrayRef<T> copyInto(ArrayRef<T> elements);
   StringRef copyInto(StringRef str);
   template <typename T> T *allocate();
   void *allocate(size_t size, size_t alignment);
   bool allocated(const void *ptr);
 private:
   llvm::BumpPtrAllocator allocator;
```


MLIRContext
StorageUniquer attributeUniquer;
DenseMap<TypeID, std::unique_ptr<ParametricStorageUniquer>>
StringAttr:
ParametricStorageUniquer
DenseSet<HashedStorage, StorageKeyInfo> instances;
{unsigned hash, Storage* ptr}
{unsigned hash, Storage* ptr}
{unsigned hash, ...
llvm::BumpPtrAllocator allocator;
4c6f72656d20697073756d20646f6c6f72207369742061 6d65742c20636f6e7365
63746574757220616469706973636696e6720656c69742e20496e206163…
IntegerAttr:
ParametricStorageUniquer

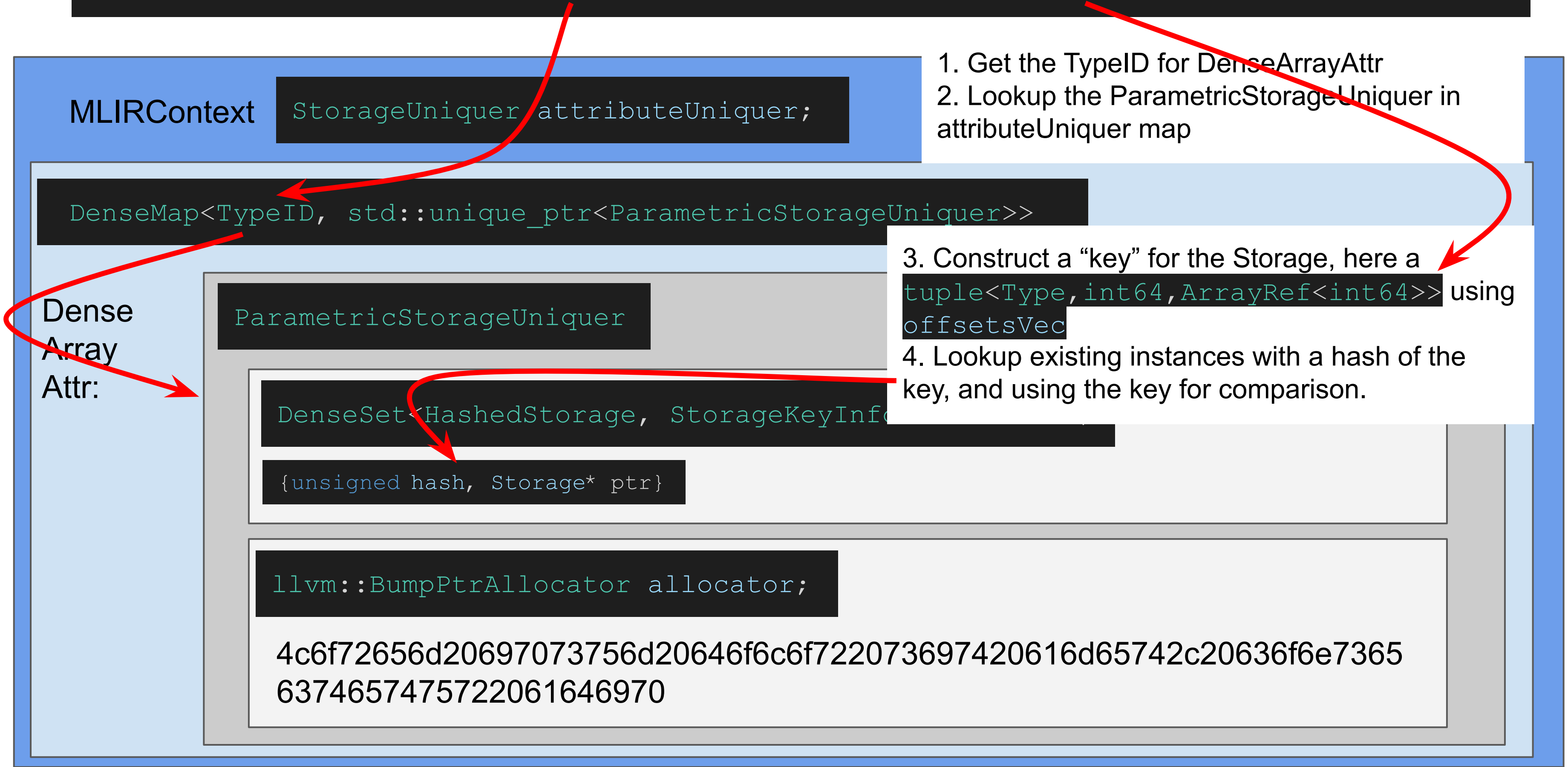
Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```



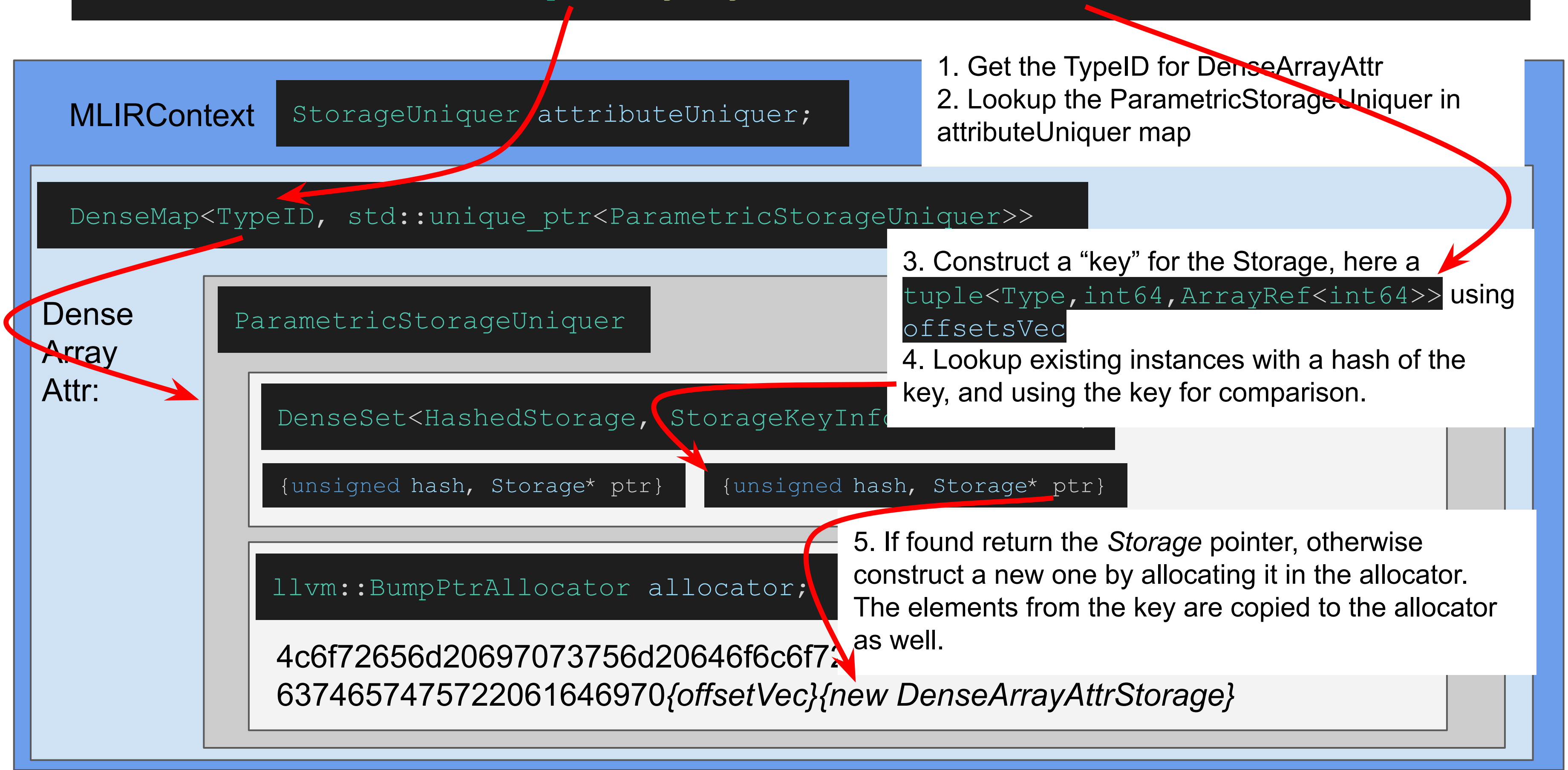
Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```



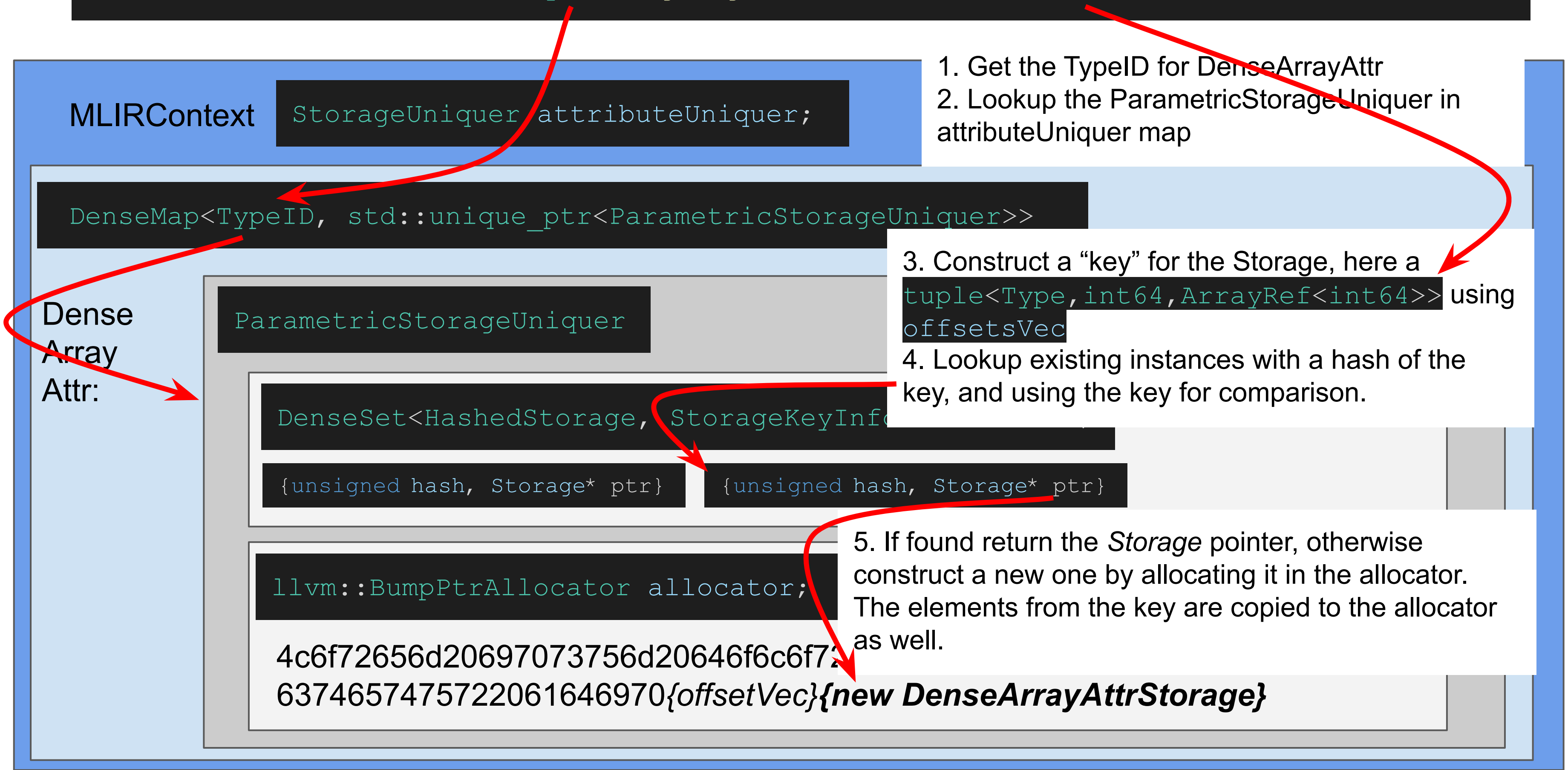
Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```



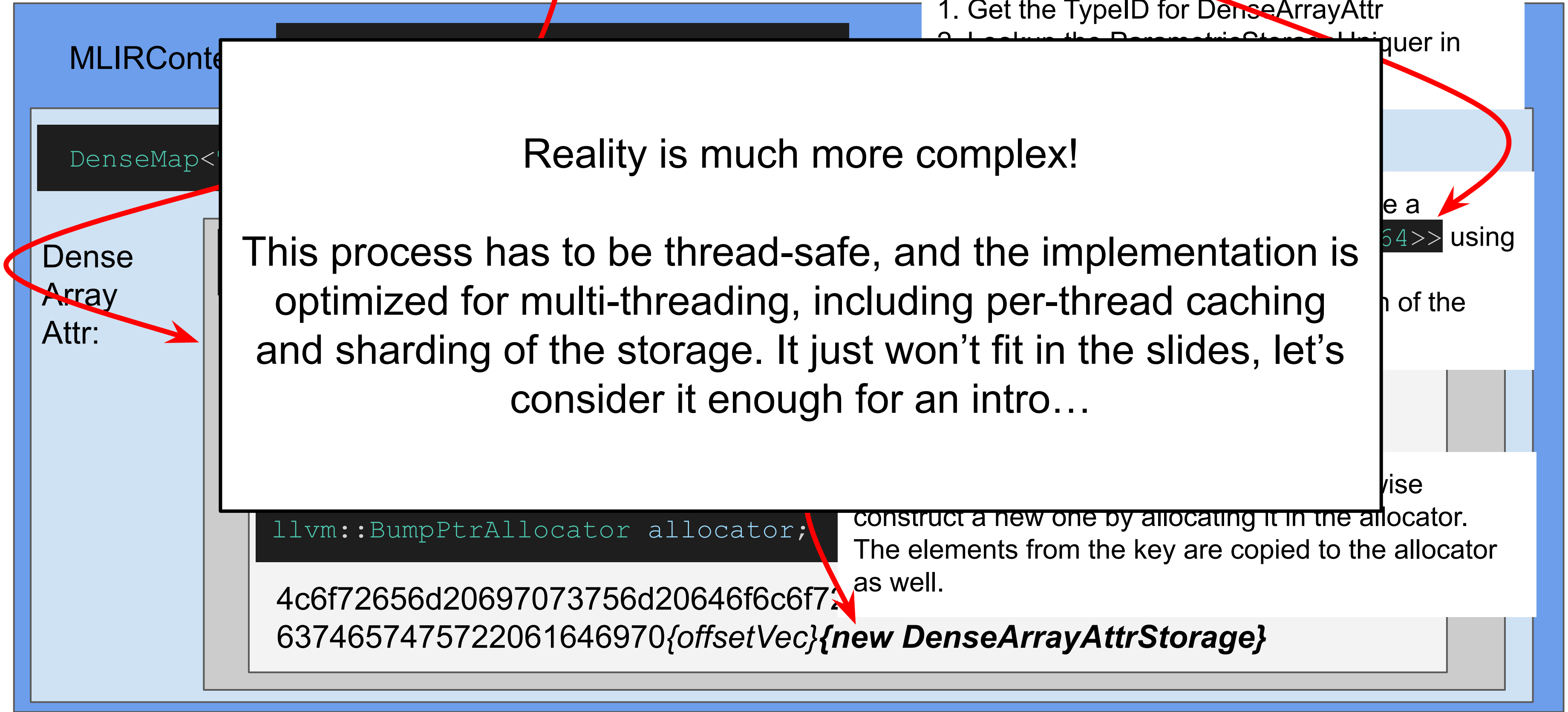
Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```



Have you ever wondered what happens when you do the following?

```
SmallVector<int64_t> offsetsVec = getOffsets();
auto offsets = DenseI64ArrayAttr::get(getContext(), offsetsVec);
```



Reality is much more complex!

This process has to be thread-safe, and the implementation is optimized for multi-threading, including per-thread caching and sharding of the storage. It just won't fit in the slides, let's consider it enough for an intro…

Attributes: recap

- Immutable objects
- “Get or create” access pattern to retrieve a unique pointer per *MLIRContext*
- Content-based hashing and comparison (on every *get*)
- Memory “leaks” into the context (bump ptr allocator)

But:

- Simple ownership model (tied to the *MLIRContext*)
- “Pointer Comparison” to check for equality between two attributes.

Operation Accessors

```
class Operation {
  Value getOperand(unsigned idx);
  void setOperand(unsigned idx, Value value);
  void eraseOperand(unsigned idx);
  unsigned getNumResults();
  OpResult getResult(unsigned idx);
  Region &getRegion(unsigned index) {

  DictionaryAttr getAttrDictionary();
  void setAttrs(DictionaryAttr newAttrs);
  Attribute getAttr(StringAttr name);
  void setAttr(StringAttr name, Attribute value);
  Attribute removeAttr(StringAttr name);
```

Operation Accessors

```
class Operation {
  Value getOperand(unsigned idx);
  void setOperand(unsigned idx, Value value);
  void eraseOperand(unsigned idx);
  unsigned getNumResults();
  OpResult getResult(unsigned idx);
  Region &getRegion(unsigned index) {

  DictionaryAttr getAttrDictionary();
  void setAttrs(DictionaryAttr newAttrs);
  Attribute getAttr(StringAttr name);
  void setAttr(StringAttr name, Attribute value);
  Attribute removeAttr(StringAttr name);
```

Direct member access and mutation

Direct member access and mutation, but uncommon API

Commonly used API, but hiding complex and inefficient behavior!

Operation Accessors

```
void setAttr(StringAttr name, Attribute value);
Attribute removeAttr(StringAttr name);
```

```
       0 | class mlir::Operation [sizeof=64, dsize=64, align=8, nvsize=64, nvalign=8]
       0 |   class llvm::ilist_node_with_parent<class mlir::Operation, class mlir::Block> (base)
       0 |           class llvm::PointerIntPair<class llvm::ilist_node_base<true> *, 1> PrevAndSentinel
       8 |           class llvm::ilist_node_base<true> * Next
       0 |   class llvm::TrailingObjects<class mlir::Operation,...> (base) (empty)
      16 |   class mlir::Block * block
      24 |   class mlir::Location location
      32 |   unsigned int orderIndex
      36 |   const unsigned int numResults
      40 |   const unsigned int numSuccs
 44:0-30 |   const unsigned int numRegions
  47:7-7 |   _Bool hasOperandStorage
      48 |   class mlir::OperationName name
      56 |   class mlir::DictionaryAttr attrs
```

DictionaryAttr

Conceptually: Map<String, Attribute>

Reality: a sorted *ArrayRef<Pair<StringAttr, Attribute>>*

Add Immutability and *MLIRContext* storage…

DictionaryAttr

Conceptually: Map<String, Attribute>

Reality: a sorted *ArrayRef<Pair<StringAttr, Attribute>>*

Add Immutability and *MLIRContext* storage…

```
void setAttr(StringAttr name, Attribute value) {
  NamedAttrList attributes(attrs);
  if (attributes.set(name, value) != value)
    attrs = attributes.getDictionary(getContext());
}
```

Copy the Dict into a vector

Mutate the vector in-place

“GetOrCreate” a new Dictionary in the context (including content hashing/copying)

ODS Accessors

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
  let arguments = (ins Arith_CmpIPredicateAttr:$predicate,
                       SignlessIntegerLikeOfAnyRank:$lhs,
                       SignlessIntegerLikeOfAnyRank:$rhs);
}
```

```
// Custom form of scalar "signed less than" comparison.
%x = arith.cmpi slt, %lhs, %rhs : i32
// Generic form of the same operation.
%x = "arith.cmpi"(%lhs, %rhs) {predicate = 2 : i64} : (i32, i32) -> i1
```

ODS Accessors

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
 let arguments = (ins Arith_CmpIPredicateAttr:$predicate,
                      SignlessIntegerLikeOfAnyRank:$lhs,
                      SignlessIntegerLikeOfAnyRank:$rhs);
}
```

```
void swapOperands(arith::CmpIOp op) {
 arith::CmpIOp op;
 Value lhs = op.getLhs();
 Value rhs = op.getRhs();
 getLhsMutable().assign(rhs);
 getRhsMutable().assign(lhs);
}
```

ODS-generated accessor, “thin” wrappers over direct member access

direct member access

ODS Accessors

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
  let arguments = (ins Arith_CmpIPredicateAttr:$predicate,
                       SignlessIntegerLikeOfAnyRank:$lhs,
                       SignlessIntegerLikeOfAnyRank:$rhs);
}
```

```
// Move constant operand to the right side and reverse the predicate.
if (adaptor.getLhs() && !adaptor.getRhs()) {
  CmpIPredicate origPred = getPredicate();
  setPredicate(getSwappedPredicate(origPred));
  swapOperands(*this);
  return getResult();
}
```

ODS-generated getter/setter, “thin” wrappers over *setAttr/getAttr* on *Operation*

ODS Accessors

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
  let arguments = (ins Arith_CmpIPredicateAttr:$predicate,
                       SignlessIntegerLikeOfAnyRank:$lhs,
                       SignlessIntegerLikeOfAnyRank:$rhs);
}
```

```
  // Mo
  if (adaptor.getLhs() && !adaptor.getRhs()) {
    CmpIPredicate origPred = getPredicate();
    setPredicate(getSwappedPredicate(origPred));
    swapOperands(*this);
    return getResult();
  }
```

"GetOrCreate" a new Dictionary in the context (including content hashing/copying)

ODS-generated getter/setter, "thin" wrappers over *setAttr/getAttr* on *Operation*

Operation Mutability

- Swapping, adding, removing operands: usual C++ direct member access
- Adding/Modifying attributes: complex and costly
  - Copy dictionary content to a vector
  - Edit vector in-place
  - Hash the content, lookup in the context.
  - Copy (and leak) the content in the context if not found.

Sequence of mutations of an operation will leak copies of the dictionary in the context, including the intermediate state!

Operation Mutability

```
op.setAttr("attr1", IntegerAttr::get(int32Ty, 42));
op.setAttr("attr2", IntegerAttr::get(int32Ty, 43));
op.setAttr("attr3", IntegerAttr::get(int32Ty, 44));
```

1) Find the *ParametricStorageUniquer* for *IntegerAttr*
2) Hash “42” and lookup an existing Storage, or allocate a new one and copy 42
3) Copy the current DictionaryAttr content for `op` into a vector,
4) Insert an entry for "attr1" and the new IntegerAttr in the vector
5) Find the *ParametricStorageUniquer* for DictionaryAttr
6) Hash the vector and lookup an existing Storage, or allocate memory for the vector and copy the content, before allocating a new DictionaryAttr Storage and returning it.

Repeat 3 times!

{*ptr1* = 42 : i32}   {*ptr2* = 43 : i32}   {*ptr3* = 44 : i32}

{ {“attr1” = *ptr1*}, {“attr1” = *ptr1*, “attr2” = *ptr2*}, {“attr1” = *ptr1*, “attr2” = *ptr2*, “attr3” = *ptr3*} }

Operation Mutability

```
op.setAttr("attr1", IntegerAttr::get(int32Ty, 42));
op.setAttr("attr2", IntegerAttr::get(int32Ty, 43));
op.setAttr("attr3", IntegerAttr::get(int32Ty, 44));
```

IntegerAttr ParametricStorageUniquer

ptr1= 42 : i32 ptr3= 43 : i32 ptr6= 44 : i32

StringAttr ParametricStorageUniquer

ptr2= “attr1” ptr4= “attr2” ptr7= “attr3”

DictionaryAttr ParametricStorageUniquer

ptr3= {<ptr2, ptr1>} ptr5= {<ptr2, ptr1>, <ptr3, ptr4>} ptr8= {<ptr2, ptr1>, <ptr3, ptr4>, <ptr6, ptr7>}

Operation Mutability

```
op.setAttr("attr1", IntegerAttr::get(int32Ty, 42));
op.setAttr("attr2", IntegerAttr::get(int32Ty, 43));
op.setAttr("attr3", IntegerAttr::get(int32Ty, 44));
```

IntegerAttr ParametricStorageUniquer

ptr1= 42 : i32 ptr3= 43 : i32 ptr6= 44 : i32

StringAttr ParametricStorageUniquer

ptr2= “attr1” ptr4= “attr2” ptr7= “attr3”

DictionaryAttr ParametricStorageUniquer

ptr3= {<ptr2, ptr1>} ptr5= {<ptr2, ptr1>, <ptr3, ptr4>} ptr8= {<ptr2, ptr1>, <ptr3, ptr4>, <ptr6, ptr7>}

These intermediary dictionary are “leaked” in the context unnecessarily

Operation Mutability: with ODS APIs

```
op.setAttr1(42);  // Still do the same thing under the hood behind ODS setters.
op.setAttr2(43);
op.setAttr3(44);
```

IntegerAttr ParametricStorageUniquer

ptr1= 42 : i32	ptr3= 43 : i32	ptr6= 44 : i32

StringAttr ParametricStorageUniquer

ptr2= “attr1”	ptr4= “attr2”	ptr7= “attr3”

DictionaryAttr ParametricStorageUniquer

ptr3= {<ptr2, ptr1>}	ptr5= {<ptr2, ptr1>, <ptr3, ptr4>}	ptr8= {<ptr2, ptr1>, <ptr3, ptr4>, <ptr6, ptr7>}

These intermediary dictionary are “leaked” in the context unnecessarily

Operation Mutability: with ODS APIs

```
op.setAttr1(42);  // Still do the same thing under the hood behind ODS setters.
op.setAttr2(43);
op.setAttr3(44);
```

We can save the intermediate DictionaryAttr, but at the cost of significant boilerplate!

```
int32_t newAttr1 = 42, newAttr2 = 43, newAttr3 = 44;

// Copy the dictionary into a vector of attributes, and mutate it in-place
NamedAttrList attrs(op.getAttrDictionary());
// Using a string key for example.
 attrs.set("axis1",  IntegerAttr::get(int32Ty, newAxis1));
// Or more efficiently using a precomputed keys exposed through an ODS accessors
attrs.set(op.getAxis2AttrName(),  IntegerAttr::get(int32Ty, newAxis2));
attrs.set(op.getAxis3AttrName(),  IntegerAttr::get(int32Ty, newAxis3));

// Build a new DictionaryAttr in the context.
DictionaryAttr dict = attrs.getDictionary(ctx);
// Update the operation in-place by swapping-in the new Dictionary.
op.setAttrs(dict);
```

Operation Mutability: with ODS APIs

```
  op.setAttr1(42);  // Still do the same thing under the hood behind ODS setters.
  op.setAttr2(43);
  op.setAttr3(44);
```

We can save the intermediate DictionaryAttr, but at the cost of significant boilerplate!

```
int32_t newAttr1 = 42, newAttr2 = 43, newAttr3 = 44;

// Copy the dictionary into a vector of attributes, and mutate it in-place
NamedAttrList attrs(op.getAttrDictionary());
// Using a string key for example.
 attrs.set("axis1",  IntegerAttr::get(int32Ty, newAxis1));
// Or more efficiently using a precomputed keys exposed through an ODS accessors
attrs.set(op.getAxis2AttrName(),  IntegerAttr::get(int32Ty, newAxis2));
attrs.set(op.getAxis3AttrName(),  IntegerAttr::get(int32Ty, newAxis3));

// Build a new DictionaryAttr in the context.
DictionaryAttr dict = attrs.getDictionary(ctx);
// Update the operation in-place by swapping-in the new Dictionary.
op.setAttrs(dict);
```

Still a significant traffic and uniquing in the MLIRContext!

Properties

Main goals

- Cleanly separate “inherent” and “discardable” attributes: separate concept deserve dedicated namespace. Two DictionaryAttr would be a solution.
- Align inherent attribute access with other Operation member (like operands), remove indirections.
- Mutability of Operation inherent attributes should be “free”: no complex hashing, locking, etc.
- Lifetime of the data should be tied to the Operation itself.

Goodbye “Attributes”, hello “Properties”!

Solution

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
 let arguments = (ins SignlessIntegerLikeOfAnyRank:$lhs,
                      SignlessIntegerLikeOfAnyRank:$rhs,
                      Arith_CmpIPredicateAttr:$predicate);
```

```
 0 | class mlir::Operation
   |   …
56 |   class mlir::DictionaryAttr attrs
64 |   OperandStorage [size=16]
80 |   OpOperand [size=16]
96 |   OpOperand [size=16]
```

Current Layout

"Predicate" attribute is stored in the DictionaryAttr.

Solution

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
 let arguments = (ins SignlessIntegerLikeOfAnyRank:$lhs,
                      SignlessIntegerLikeOfAnyRank:$rhs);
 let properties = (ins Property<"CmpIPredicate">:$predicate);
```

```
    0 | class mlir::Operation
      |   …
   56 |   class mlir::DictionaryAttr attrs
   64 |   OperandStorage [size=16]
   80 |   Properties [size = ?]
x+80 |   OpOperand [size=16]
x+96 |   OpOperand [size=16]
```

New Layout

"Predicate" is stored as an enum in the *Properties* allocation.

Solution

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
 let arguments = (ins SignlessIntegerLikeOfAnyRank:$lhs,
                      SignlessIntegerLikeOfAnyRank:$rhs);
 let properties = (ins Property<"CmpIPredicate">:$predicate);
```

Is roughly equivalent to:

```
let extraClassDeclaration = [{
   struct alignas(8) Properties {
     CmpIPredicate predicate;
     CmpIPredicate getPredicate() const { return predicate; }
     void setPredicate(CmpIPredicate predicate) { this->predicate = predicate; }
   };
   /// Return a mutable reference to the properties
   Properties &getProperties();
}];
```

Solution

```
def Arith_CmpIOp : Arith_CompareOpOfAnyRank<"cmpi"> {
  let arguments = (ins SignlessIntegerLikeOfAnyRank:$lhs,
                       SignlessIntegerLikeOfAnyRank:$rhs);
  let properties = (ins Property<"CmpIPredicate">:$predicate);
```

```
   0 | class mlir::Operation
     |   …
  56 |   class mlir::DictionaryAttr attrs
  64 |   OperandStorage [size=16]
  80 |   Properties [size = sizeof(arith::CmpIOp::Properties])]
x+80 |   OpOperand [size=16]
x+96 |   OpOperand [size=16]
```

Operation Mutability

```
op.setAttr("attr1", IntegerAttr::get(int32Ty, 42));
op.setAttr("attr2", IntegerAttr::get(int32Ty, 43));
op.setAttr("attr3", IntegerAttr::get(int32Ty, 44));
```

```
struct alignas(8) Properties {
  int attr1;
  int attr2;
  int attr3;
  int getAttr1() const { return attr1; }
  int setAttr1(int value) const { attr1 = value; }
  int getAttr2() const { return attr2; }
  int setAttr2(int value) const { attr2 = value; }
  int getAttr3() const { return attr3; }
  int setAttr3(int value) const { attr3 = value; }
};
```

```
 0 | class mlir::Operation
   |   …
64 |   OperandStorage [size=16]
80 |   Properties [size = 14]
80 |    { attr1,
84 |      attr2,
88 |      attr3} // + padding 4B
96 |   OpOperand [size=16]
```

```
auto &properties = op.properties(); /// mutable reference to the Operation* member
properties.setAttr1(42); /// Direct mutation
properties.attr2 = 43;   /// Data stored inline, no Context access!
properties.setAttr3(44);
```

Operation Mutability

```
op.setAttr("attr1", IntegerAttr::get(int32Ty, 42));
op.setAttr("attr2", IntegerAttr::get(int32Ty, 43));
op.setAttr("attr3", IntegerAttr::get(int32Ty, 44));
```

```
struct alignas(8) Properties {
  int attr1;
  int attr2;
  int attr3;
  int getAttr1()
  int setAttr1(in
  int getAttr2()
  int setAttr2(in
  int getAttr3()
  int setAttr3(in
};
```

0 | class mlir::Operation
  | ...
64 | OperandStorage [size=16]
rties [size = 14]
,
,
} // + padding 4B
erand [size=16]

Properties is the ability to add *any* C++ data member to an Operation, like a regular class.

```
auto &properties = op.properties(); /// mutable reference to the Operation* member
properties.setAttr1(42); /// Direct mutation
properties.attr2 = 43;   /// Data stored inline, no Context access!
properties.setAttr3(44);
```

Fancier Example

```
// A c++ struct with 3 members,
struct Properties { // [sizeof=48]
 int64_t a = -1; // Default value are honored
 std::vector<int64_t> array = {-33}; // Yes you can have std::vector!
 // A shared_ptr to a const object is safe: it is equivalent to a value-based
 // member. Here the label will be deallocated when the last operation
 // referring to it is destroyed.
 std::shared_ptr<const std::string> label;
 ~Properties(); // Destructor will be called when the operation is destroyed.
};
```

```
MyOp::Properties &prop = op.getProperties();
prop.array.push_back(42); // std::vector modified in-place!
// Example of pool-allocation in the dialect, with ref-counting lifetime.
auto &pool = cast<MyDialect>(op->getDialect()).getMyStringPool();
std::shared_ptr<const std::string> label = pool.getOrCreate("some string");
prop.label = std::move(label);
```

Some required boilerplate…

```
// Compute a hash for the structure: this is needed for
// computing OperationEquivalence, think about CSE.
llvm::hash_code computeHash(const MyOp::Properties &prop);

// Convert the structure to an attribute: this is used when printing
// an operation in the generic form.
Attribute getPropertiesAsAttribute(MLIRContext *ctx,
                                   const MyOp::Properties &prop);

// Convert the structure from an attribute: this is used when
// parsing an operation from the generic form.
LogicalResult setPropertiesFromAttribute(MyOp::Properties &prop,
                                         Attribute attr,
                                         InFlightDiagnostic *diagnostic);
```

But it will all be generated by TableGen/ODS!

Wrapping up

Drawbacks

- Memory footprint may increase: Operation allocations get larger than before (but allocation don't *leak* anymore!
  => *Properties* can still store a DictionaryAttr, which would scale identically to current attributes.
- Checking that two operations have the same Properties requires calling the Properties comparison operator.
- Extra runtime cost:
  - When creating an operation, we initialize the properties by calling its default constructor (through an indirect call) before calling the assignment operator.
  - When cloning an operation, we call the assignment operator and copy the properties.
  - When deleting an operation, we call the properties destructor.
  - OperationEquivalence (called by CSE for example) will hash the properties (through an indirect call).

TBD

- PDL and DRR integration
- Build methods generated by ODS
- Bindings auto-generation (C and Python)

Two paths to land this

1) Properties is an opt-in: we can migrate dialects and operation as we go, and mix and match:

```
def MyOp { // This operation defines one inherent attr and one property, both int64_t.
 let arguments = (ins I64Attr:$attr1);
 let properties = (ins Property<"int64">:$prop1);
}
```

2) Always use Properties, no mix-and-match (but likely a switch on the dialect):

```
def MyOp { // This operation defines one inherent attr and one property, both int64_t.
 let arguments = (ins I64Attr:$attr1,
                      Property<"int64">:$prop1);
```

```
struct Properties { // [sizeof=16]
 IntegerAttr attr1;
 int64_t prop1;
```