Generalized Mem2Reg for MLIR and how to use it

Théo Degioanni

What is Mem2Reg?

What is Mem2Reg?

Mem2Reg in LLVM, also known as SSA construction.

Convert non-SSA memory locations into SSA values.

…when memory locations do not escape the scope.

What is Mem2Reg?

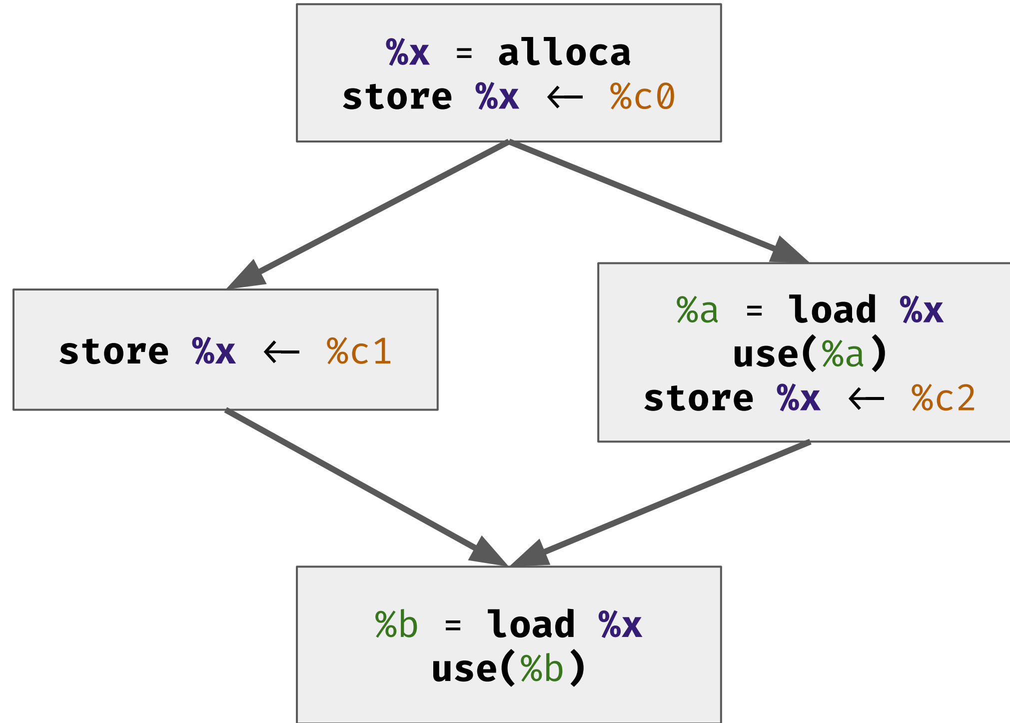
Mem2Reg in LLVM, also known as SSA construction.

Convert non-SSA memory locations into SSA values.

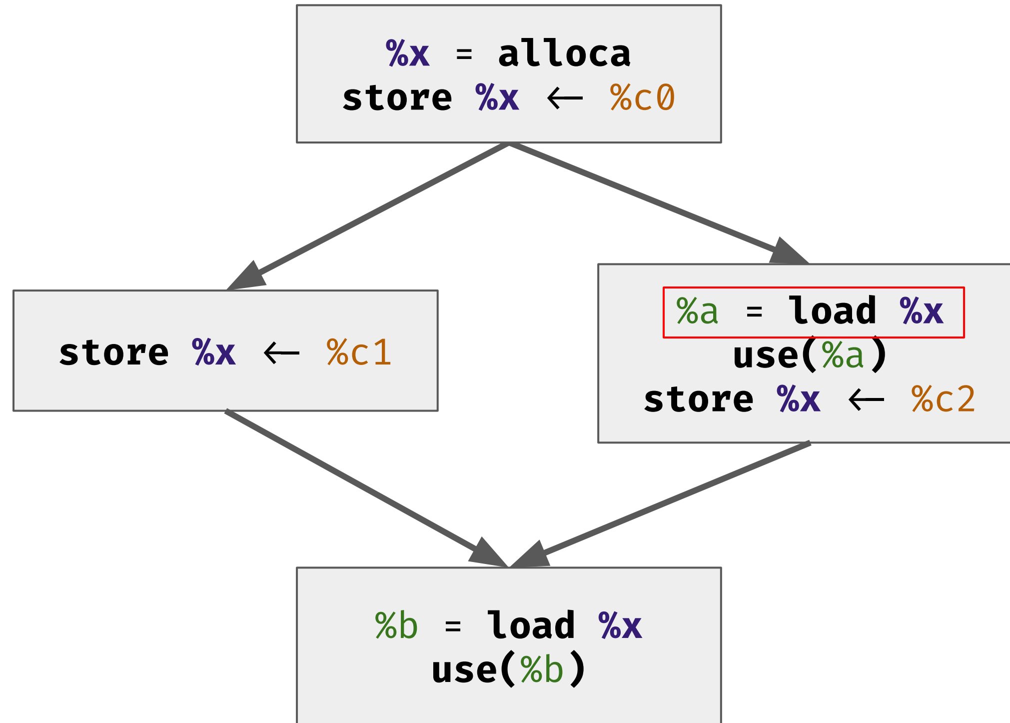
…when memory locations do not escape the scope.

In LLVM: convert stack alloca to SSA values.

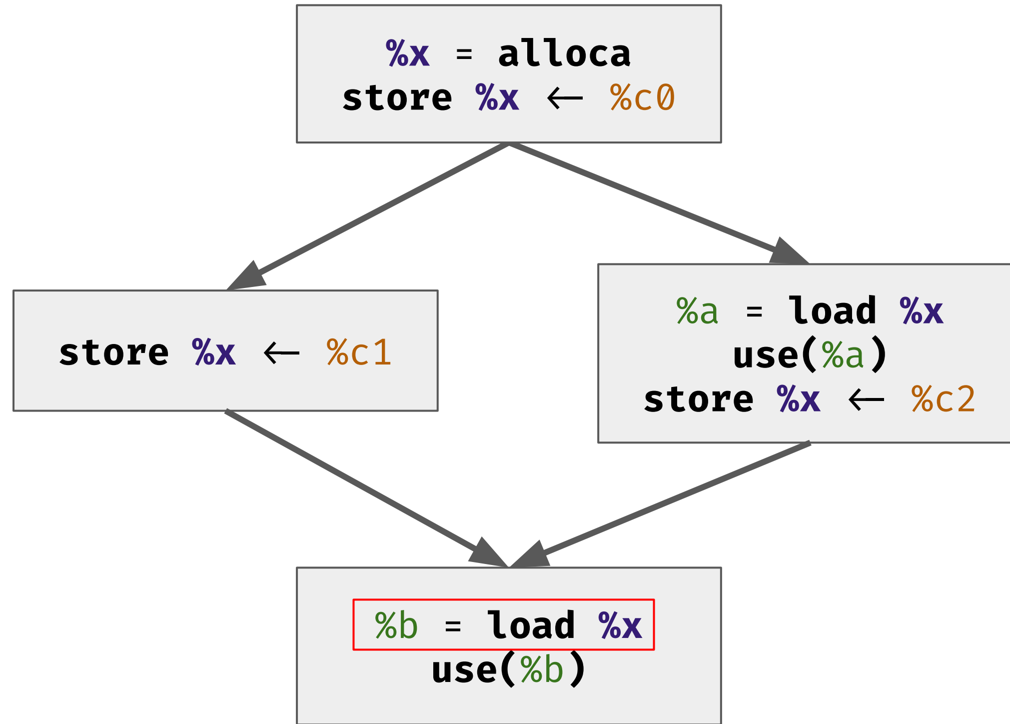
What is Mem2Reg?



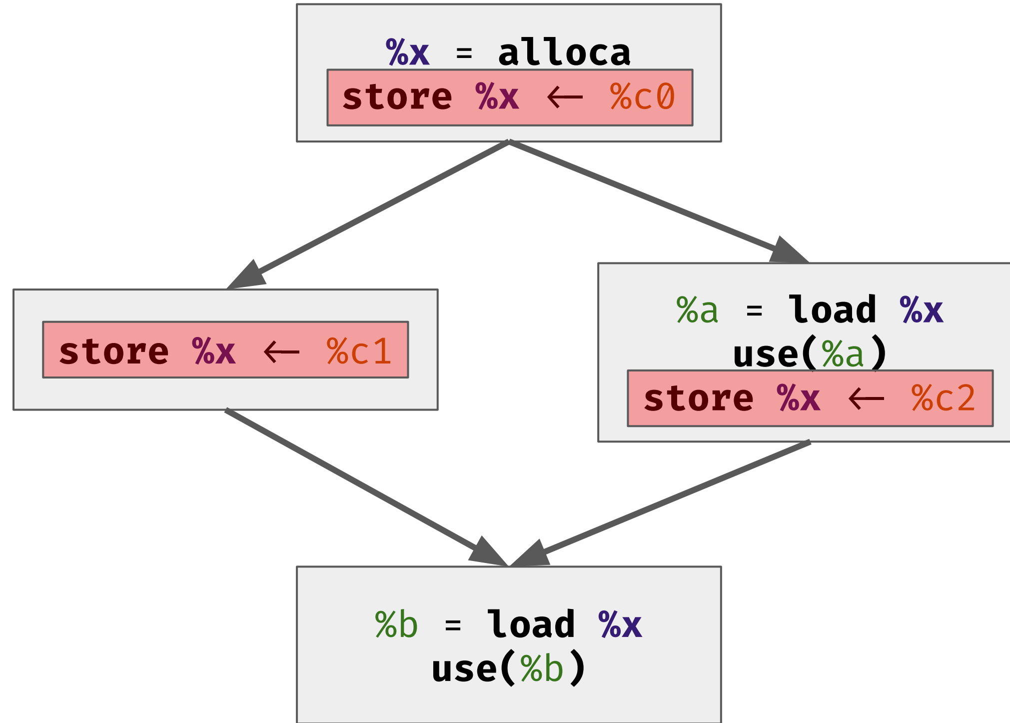
What is Mem2Reg?



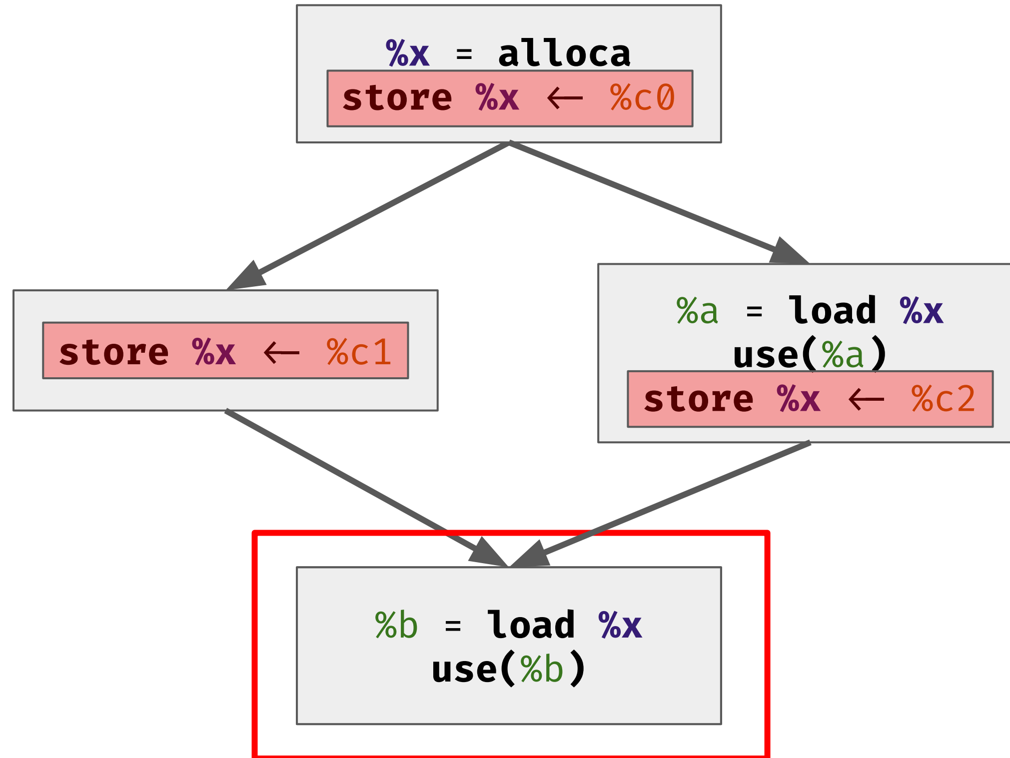
What is Mem2Reg?



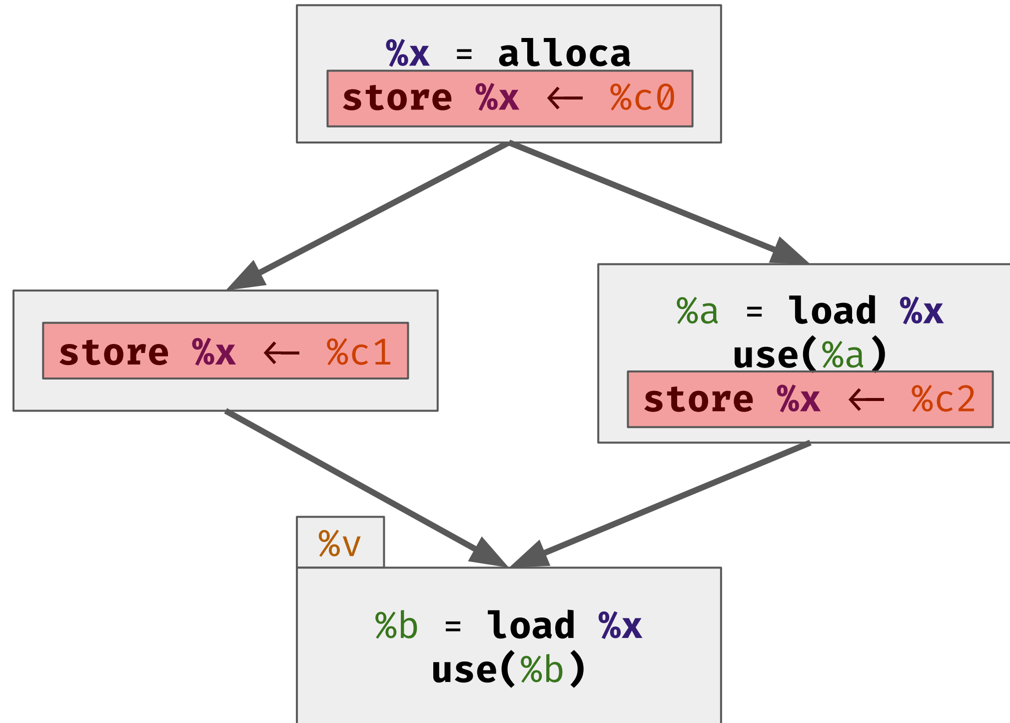
What is Mem2Reg?



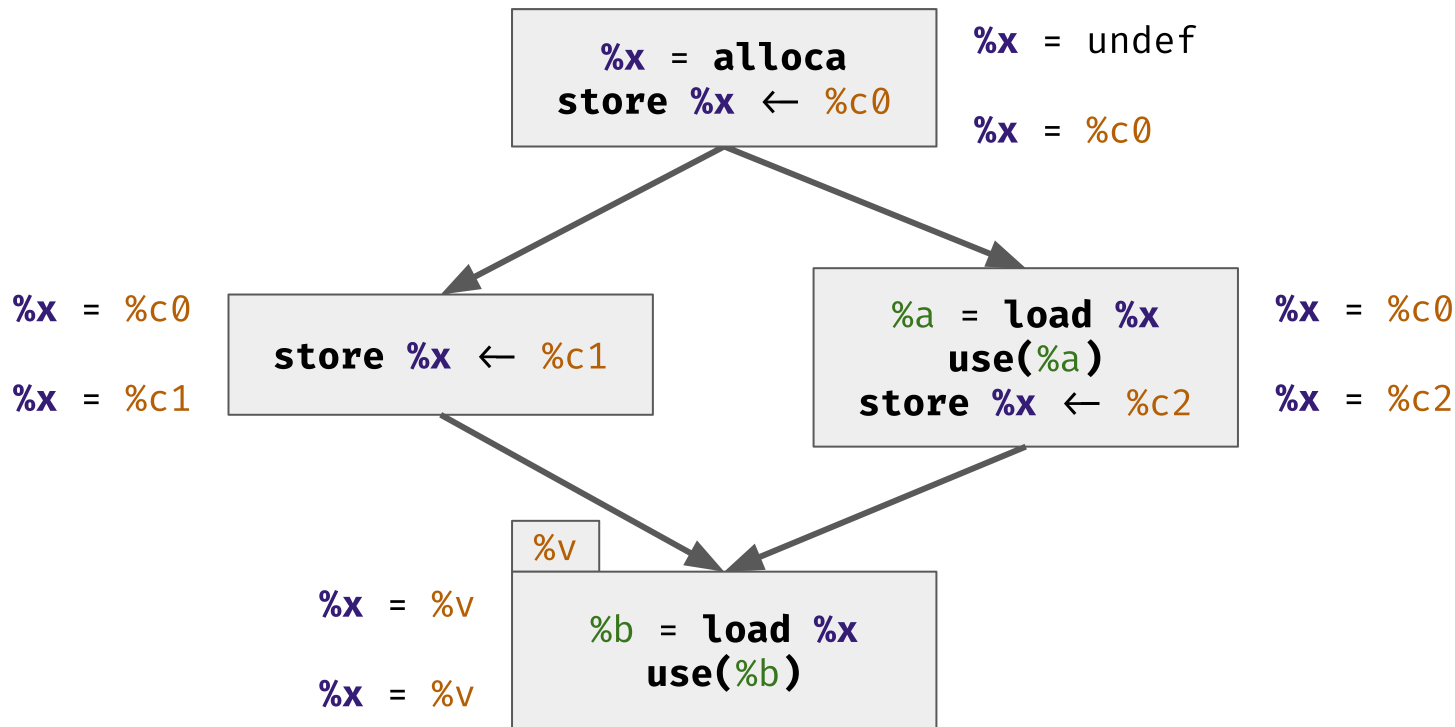
What is Mem2Reg?



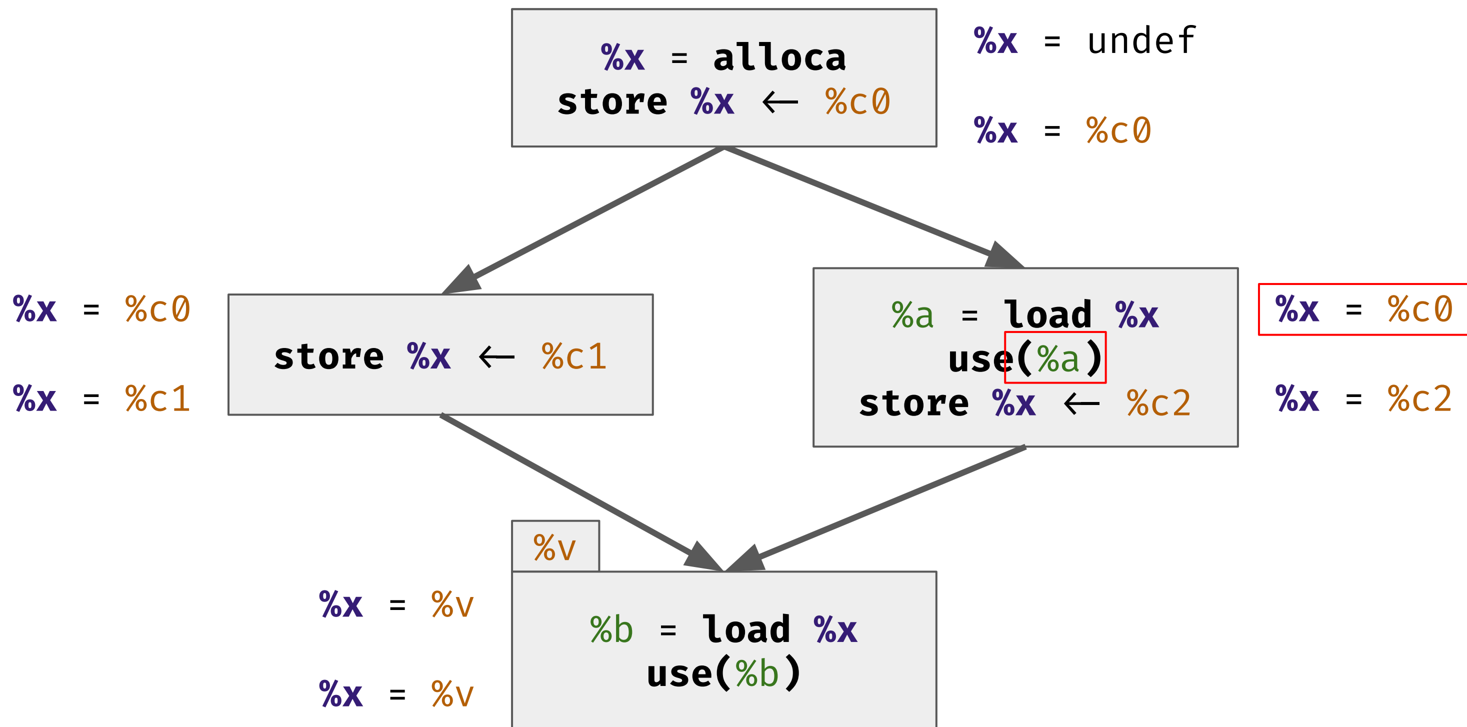
What is Mem2Reg?



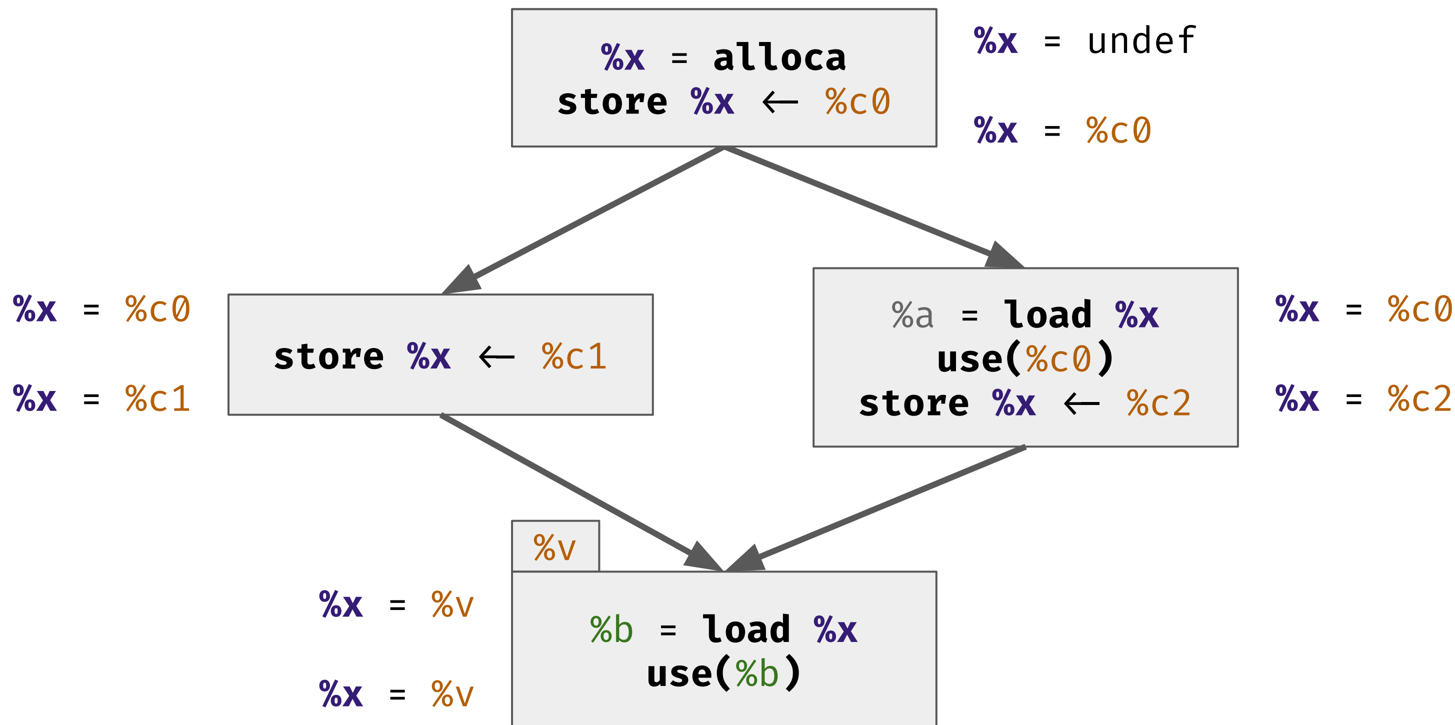
What is Mem2Reg?



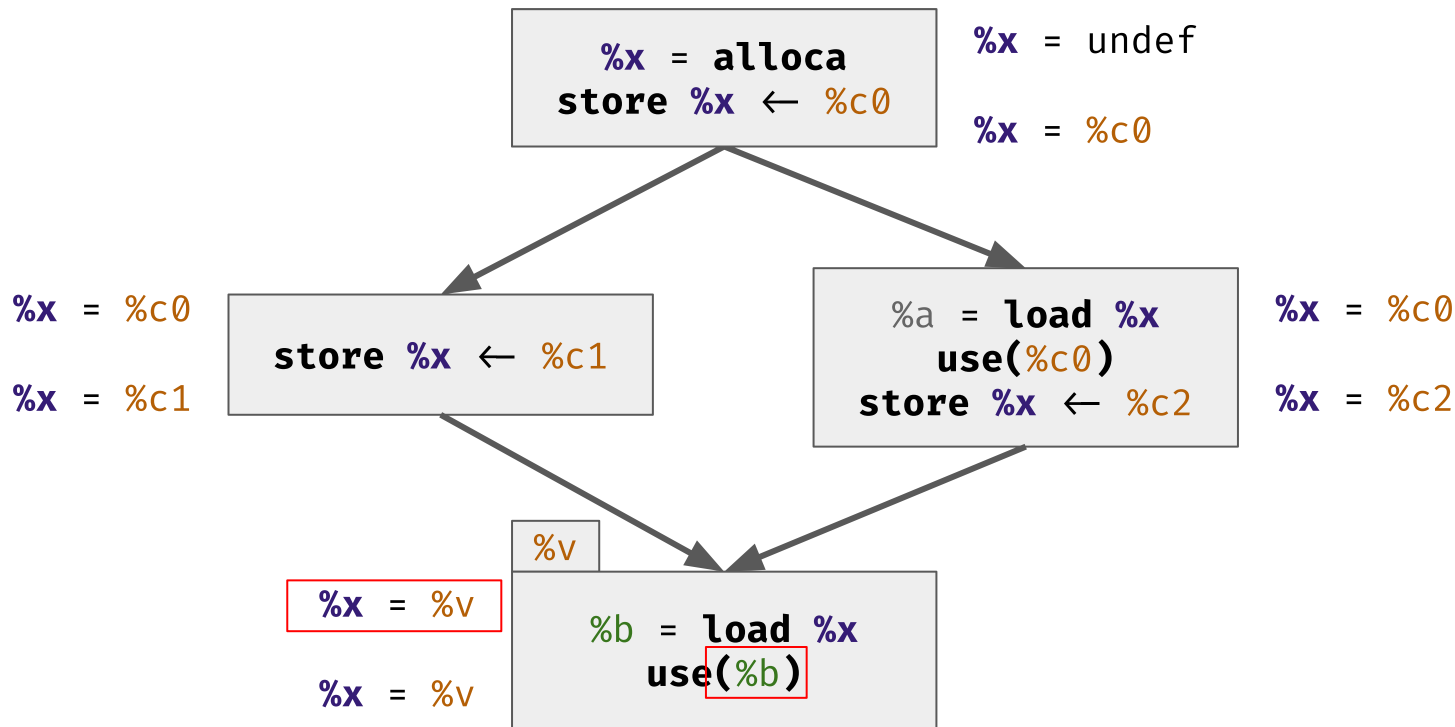
What is Mem2Reg?



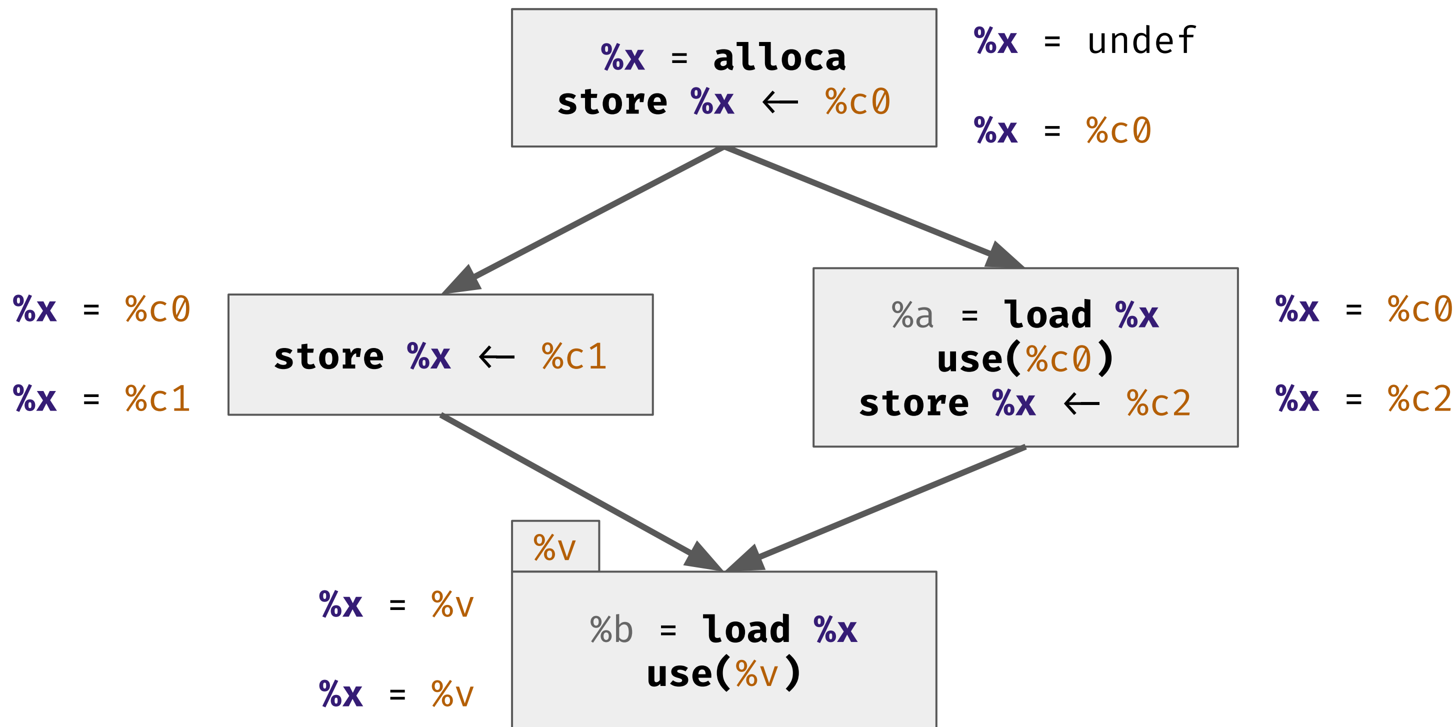
What is Mem2Reg?



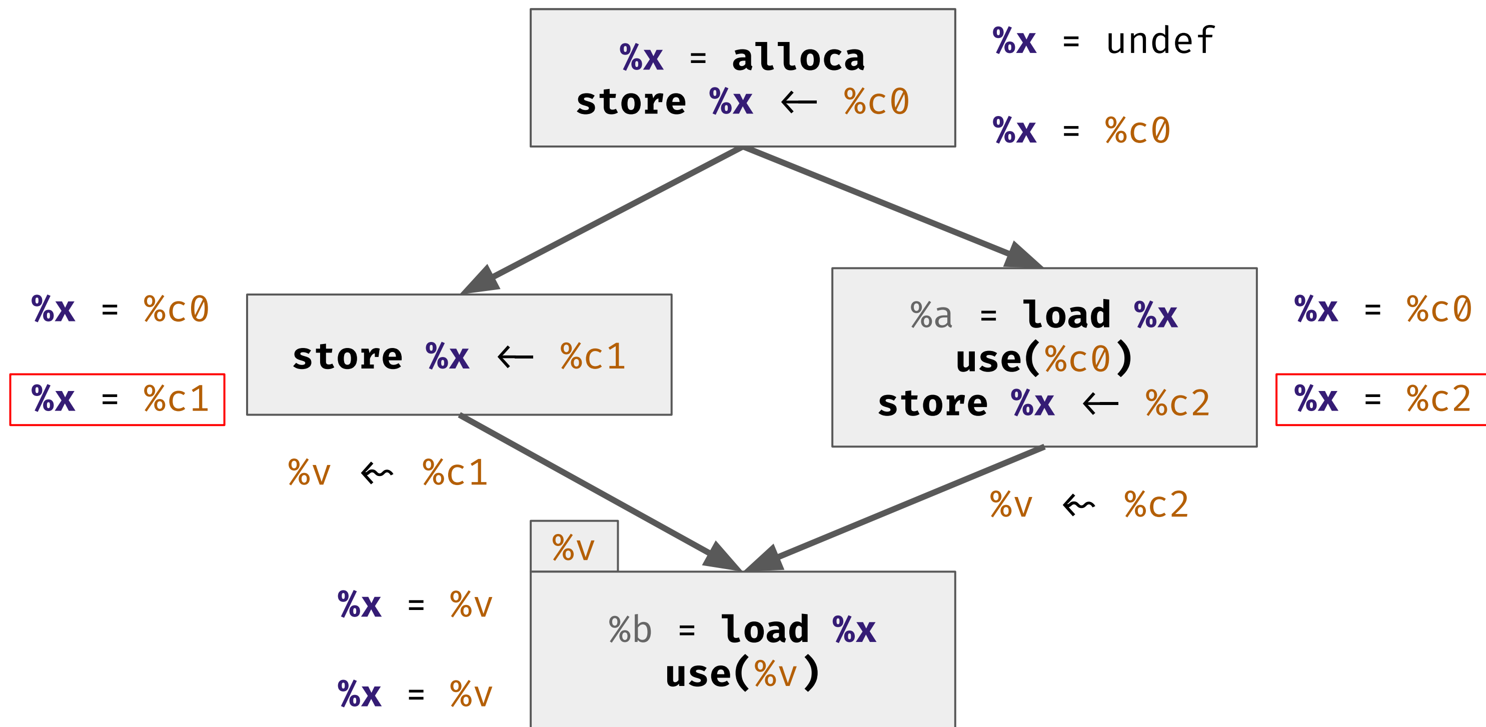
What is Mem2Reg?



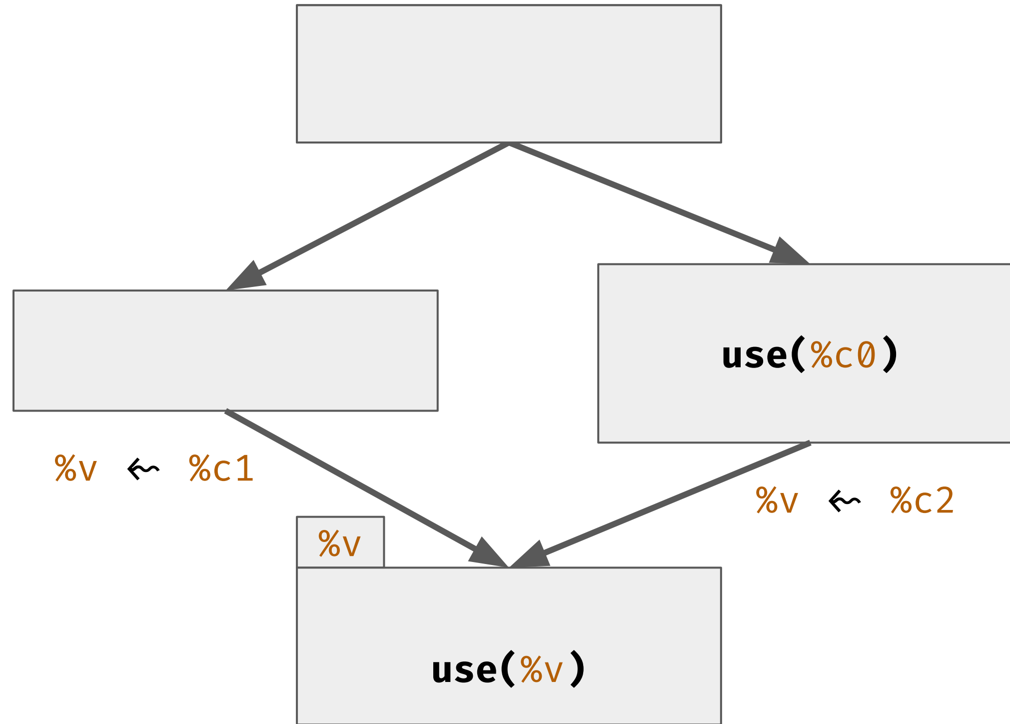
What is Mem2Reg?



What is Mem2Reg?



What is Mem2Reg?



Why Mem2Reg?

- Write programs without caring about SSA
- Remove costly memory usage
- Simplify program structures for analysis and optimization

Mem2Reg in MLIR

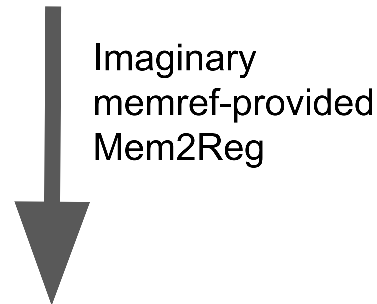
- No standard interfaces
- No standard implementation
- Implementation must be done downstream, without coordination

Unfortunate lack of coordination

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    memref.store %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    return %l : i32
}
```

Unfortunate lack of coordination

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    memref.store %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    return %l : i32
}
```



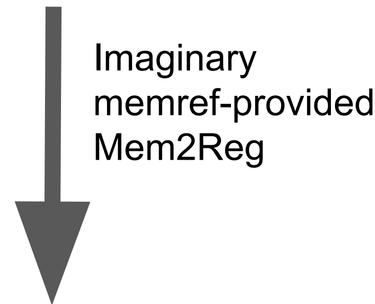

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    return %const : i32
}
```

Unfortunate lack of coordination

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    return %l : i32
}
```

Unfortunate lack of coordination

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    return %l : i32
}
```



?

Interfaces are well suited!

- Operate on any dialect that *behaves in some specified way*

- No need to define it in advance

Interfaces are well suited!

- Operate on any dialect that *behaves in some specified way*
- No need to de

How does one encode Mem2Reg semantics in interfaces?

What is a “good” “memory” location?

What is a “good” “memory” location?

Location must contain
a single value

What is a “good” “memory” location?

Location must contain a single value

Location type must be consistent

What is a “good” “memory” location?

Location must contain a single value

Location type must be consistent

Location uses must not escape a given scope/alias

What is a “good” “memory” location?

Location must contain a single value

Location type must be consistent

Location uses must not escape a given scope/alias

Location must be considered in a void

MemorySlot

```
struct MemorySlot {
  /// Pointer to the memory slot.
  Value ptr;
  /// Type of the value contained in the slot.
  Type elemType;
};
```

MemorySlot

```
struct MemorySlot {
  /// Pointer to the memory slot.
  Value ptr;
  /// Type of the value contained in the slot.
  Type elemType;
};
```

Contains a single value of a given consistent type without aliasing

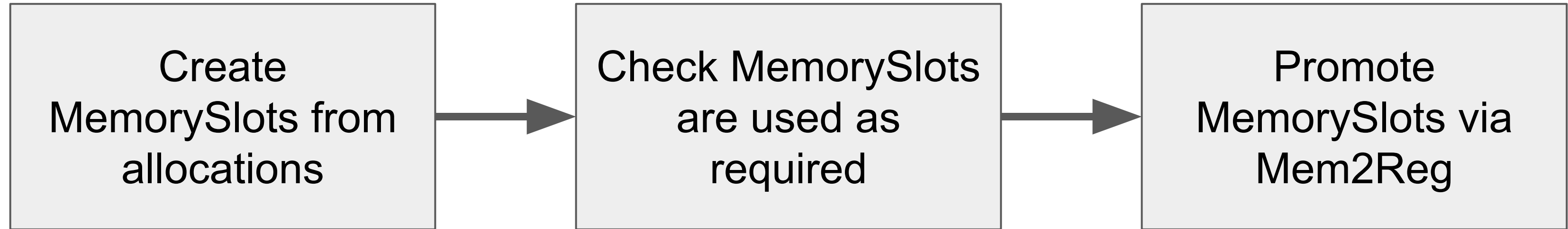
MemorySlot

```
struct MemorySlot {
  /// Pointer to the memory slot.
  Value ptr;
  /// Type of the value contained in the slot.
  Type elemType;
};
```

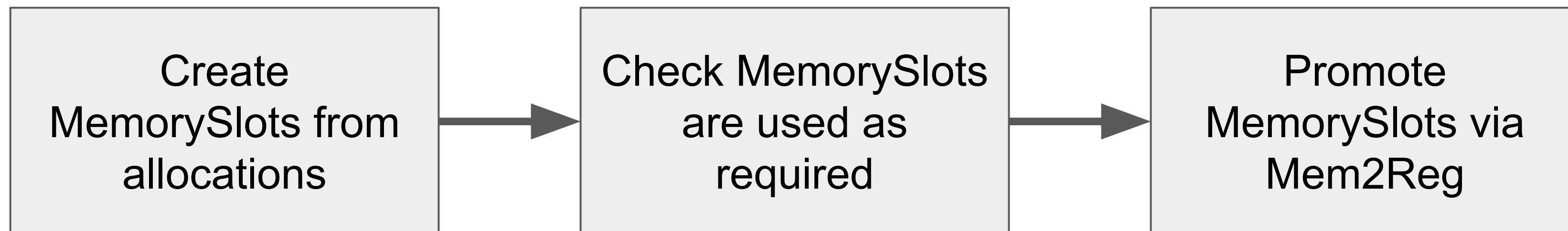
Contains a single value of a given consistent type without aliasing

Pointer must be used to lookup a value of the type and nothing else

MemorySlot



MemorySlot



Let’s implement this for the previous example!

Create MemorySlots

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Create MemorySlots from allocations

Create MemorySlots

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Create MemorySlots from allocations

Create MemorySlots

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Create MemorySlots from allocations

Create MemorySlots

PromotableAllocationOpInterface for `memref.alloca`

```
SmallVec<MemorySlot> getPromotableSlots();

Value getDefaultValue(MemorySlot &slot,
                      RewriterBase &rewriter);

// …
```

Create MemorySlots from allocations

Create MemorySlots

PromotableAllocationOpInterface for `memref.alloca`

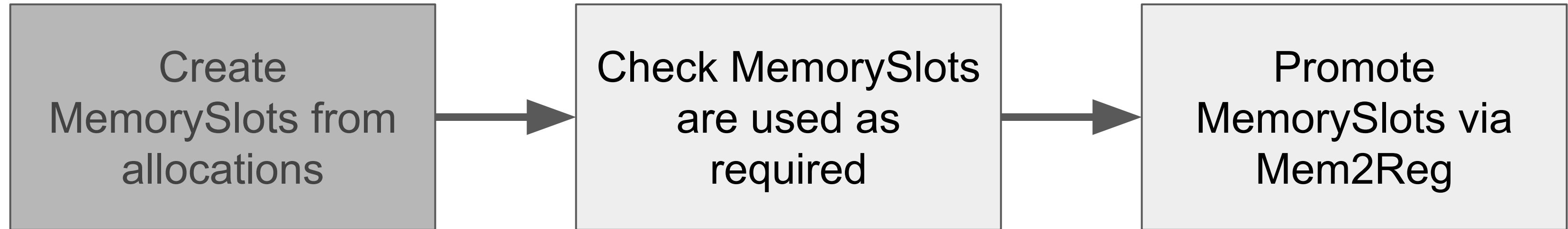
```
SmallVec<MemorySlot> getPromotableSlots();

Value getDefaultValue(MemorySlot &slot,
                      RewriterBase &rewriter);

// …
```

```
%mem = memref.alloca() : memref<i32>
```

```
MemorySlot {
  .ptr = %mem,
  .elemType = i32;
};
```

Create MemorySlots from allocations

MemorySlot



Check usage

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Check MemorySlots are used as required

Check usage

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

“blocking” uses

Check MemorySlots are used as required

Check usage

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Check MemorySlots are used as required

Check usage

PromotableOpInterface for `atomic_memref.metadata`

```
bool canUsesBeRemoved(const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter);
```

Check MemorySlots are used as required

Check usage

PromotableOpInterface for `atomic_memref.metadata`

```
bool canUsesBeRemoved(const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter);
```

- Can uses be removed? Always.
- How to remove them?
  We can just delete the op by returning
  `DeletionKind::Delete`

Check MemorySlots are used as required

Check usage

PromotableOpInterface for `atomic_memref.metadata`

```
bool canUsesBeRemoved(const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter);
```

- Can uses be removed? Always.
- How to remove them?
  We can just delete the op by returning
  `DeletionKind::Delete`

Check MemorySlots are used as required

Check usage

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Check MemorySlots are used as required

Check usage

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Check MemorySlots are used as required

Check usage

PromotableMemOpInterface for `atomic_memref.astore`

```
bool canUsesBeRemoved(const MemorySlot &slot,
                      const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const MemorySlot &slot,
                                const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter,
                                Value reachingDefinition);
```

Check MemorySlots are used as required

Check usage

PromotableMemOpInterface for `atomic_memref.astore`

```
bool canUsesBeRemoved(const MemorySlot &slot,
                      const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const MemorySlot &slot,
                                const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter,
                                Value reachingDefinition);
```

Check MemorySlots are used as required

Check usage

PromotableMemOpInterface for `atomic_memref.astore`

```
bool canUsesBeRemoved(const MemorySlot &slot,
                      const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const MemorySlot &slot,
                                const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter,
                                Value reachingDefinition);
```

Check MemorySlots are used as required

Check usage

PromotableMemOpInterface for `atomic_memref.astore`

```
bool canUsesBeRemoved(const MemorySlot &slot,
                      const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const MemorySlot &slot,
                                const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter,
                                Value reachingDefinition);
```

- Can uses be removed? As long as the types are consistent.
- How to remove them?
  We can just delete the op by returning
  `DeletionKind::Delete`

Check MemorySlots are used as required

Check usage

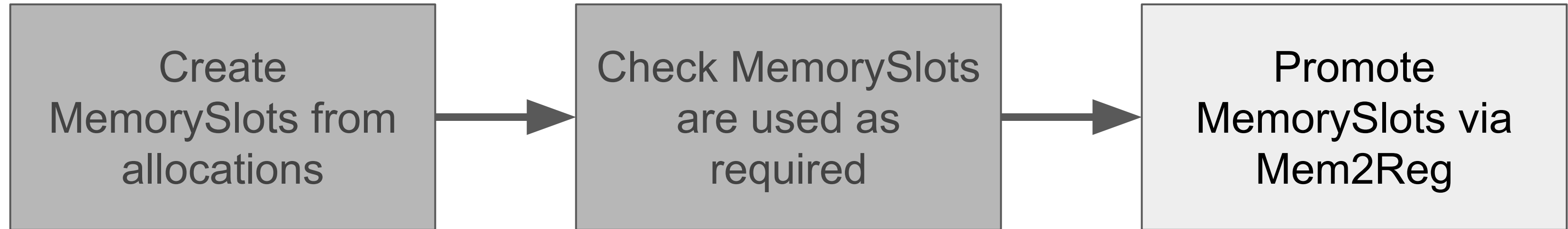
PromotableMemOpInterface for `memref.load`

```
bool canUsesBeRemoved(const MemorySlot &slot,
                      const Set<OpOperand *> &blockingUses,
                      Set<OpOperand *> &newBlockingUses);

DeletionKind removeBlockingUses(const MemorySlot &slot,
                                const Set<OpOperand *> &blockingUses,
                                RewriterBase &rewriter,
                                Value reachingDefinition);
```

- Can uses be removed? As long as the types are consistent.
- How to remove them?
  We need to replace uses then return
  `DeletionKind::Delete`

Check MemorySlots are used as required

MemorySlot



Analyze behavior for promotion

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Promote MemorySlots via Mem2Reg

Analyze behavior for promotion

```
func @demo() -> i32 {
    %const = arith.constant 12 : i32
    %mem = memref.alloca() : memref<i32>

    atomic_memref.astore %const, %mem : memref<i32>
    %l = memref.load %mem : memref<i32>

    atomic_memref.metadata %mem : memref<i32>
    return %l : i32
}
```

Promote MemorySlots via Mem2Reg

Analyze behavior for promotion

PromotableMemOpInterface

```
bool loadsFrom(const MemorySlot &slot);

bool storesTo(const MemorySlot &slot);

Value getStored(const MemorySlot &slot,
                RewriterBase &rewriter);
```

Promote MemorySlots via Mem2Reg

Analyze behavior for promotion

PromotableMemOpInterface

```
bool loadsFrom(const MemorySlot &slot);

bool storesTo(const MemorySlot &slot);

Value getStored(const MemorySlot &slot,
                RewriterBase &rewriter);
```

We can now apply the Mem2Reg rewrite pattern!

Promote MemorySlots via Mem2Reg

Mem2Reg for any dialect

- Interfaces to define how operations interact with Mem2Reg
- Upstream rewrite pattern or pass to apply the transformation
- Out-of-the-box coordination between dialects

Basic SROA also available

- Allows breaking aggregate-like types into their fields

- Achieved by breaking allocators of large MemorySlots into allocators of their fields

- Interfaces to prove slots are used correctly
  - `DestructurableAllocationOpInterface`
  - `SafeMemorySlotAccessOpInterface`
  - `DestructurableAccessorOpInterface`

What implementations are currently upstream

LLVM Dialect

- Support for `alloca` stack slots
- Support for debuginfo and markers
- Support for memory intrinsics
- Basic SROA on structs and arrays

MemRef Dialect

- Support for MemRef `alloca`
- Support for Mem2Reg of scalar MemRefs
- Basic SROA of small higher rank MemRefs

Still lots of things to be done!

- More interface design needed to support structured control flow
- Terminators must be branch-like control-flow
- More public support for open dialects that need it

Thank you!