Mutable variables and Mem2Reg

Théo Degioanni

```
git checkout exercises/day-3-mem2reg
```

Reference-based variables

- Allocate a memory location

```
%ref = list.alloca : !list.ref<i32>
```

- Load SSA variables from the memory location

```
%loaded = list.load %ref : !list.ref<i32>
```

- Store SSA variables into the memory location

```
list.store %stored, %ref : !list.ref<i32>
```

Implement it!

- Create a parametric reference type that takes in either an integer or a list
- Create an alloca operation for allocation
- Create load and store operations
  - Printer and parser are already implemented for this operation.

```
%ref = list.alloca : !list.ref<i32>
%loaded = list.load %ref : !list.ref<i32>
list.store %stored, %ref : !list.ref<i32>
```

ListLang compiler

```
git checkout exercises/day-3-mem2reg
```

Drawbacks?

- More infrastructure
- **Harder to optimize!**

ListLang

```
let mut value = 1;
value = value + 1;
value
```

Generated IR

```
builtin.module {
  func.func @main() {
    %_c1 = arith.constant 1 : i32
    %value = list.alloca : !list.ref<i32>
    list.store %_c1, %value : !list.ref<i32>
    %_value_load = list.load %value : !list.ref<i32>
    %_c1_1 = arith.constant 1 : i32
    %_value_load_plus_c1 = arith.addi %_value_load, %_c1_1 : i32
    list.store %_value_load_plus_c1, %value : !list.ref<i32>
    %_value_load_1 = list.load %value : !list.ref<i32>
    vector.print %_value_load_1 : i32
    func.return
  }
}
```

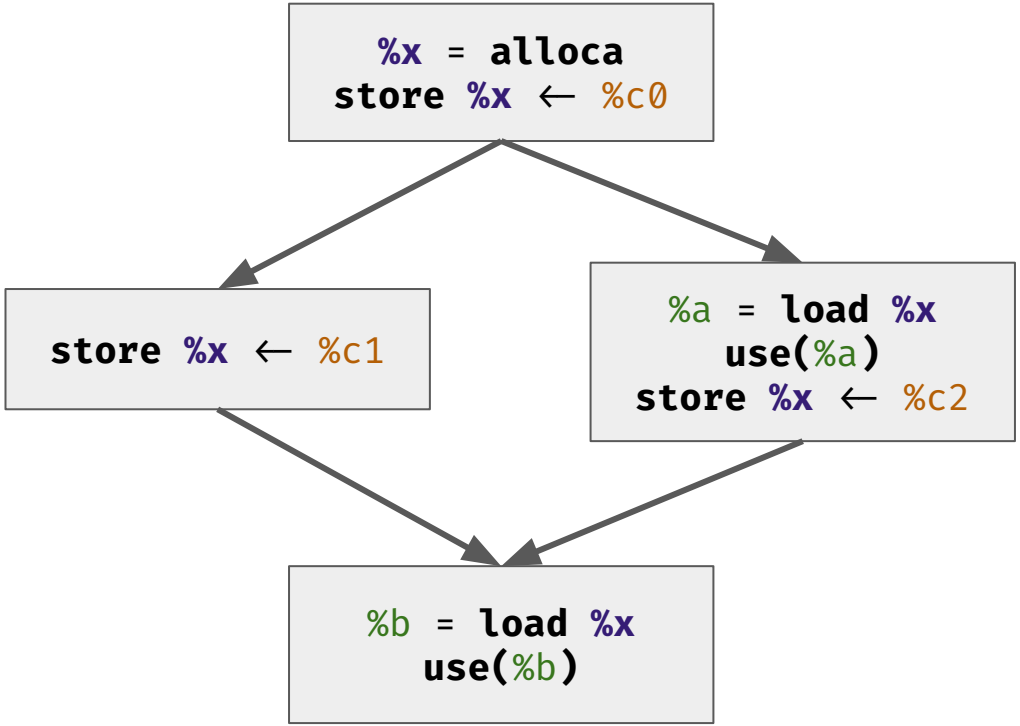
What is Mem2Reg?

Mem2Reg in LLVM, also known as SSA construction.

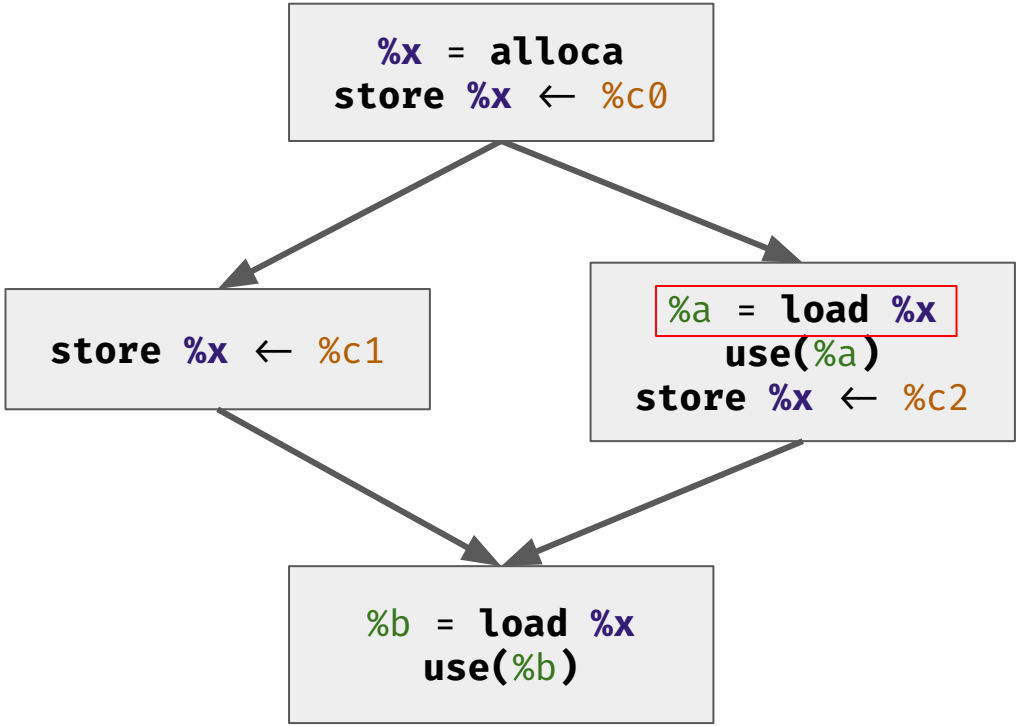
Convert non-SSA memory locations into SSA values.

…when memory locations do not escape the scope.

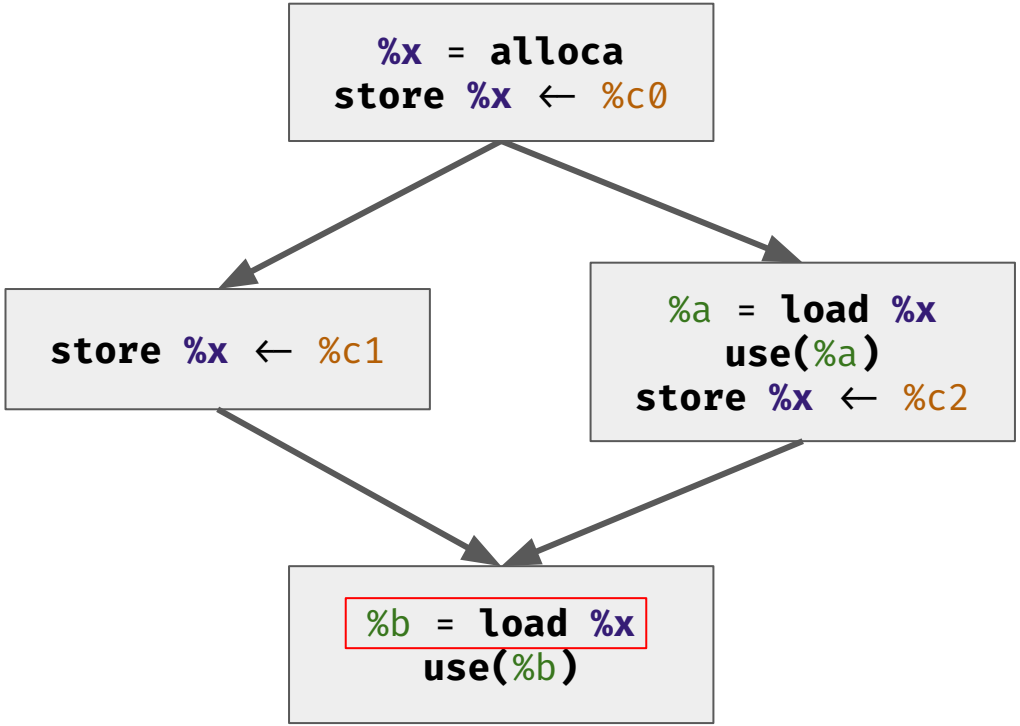
What is Mem2Reg?



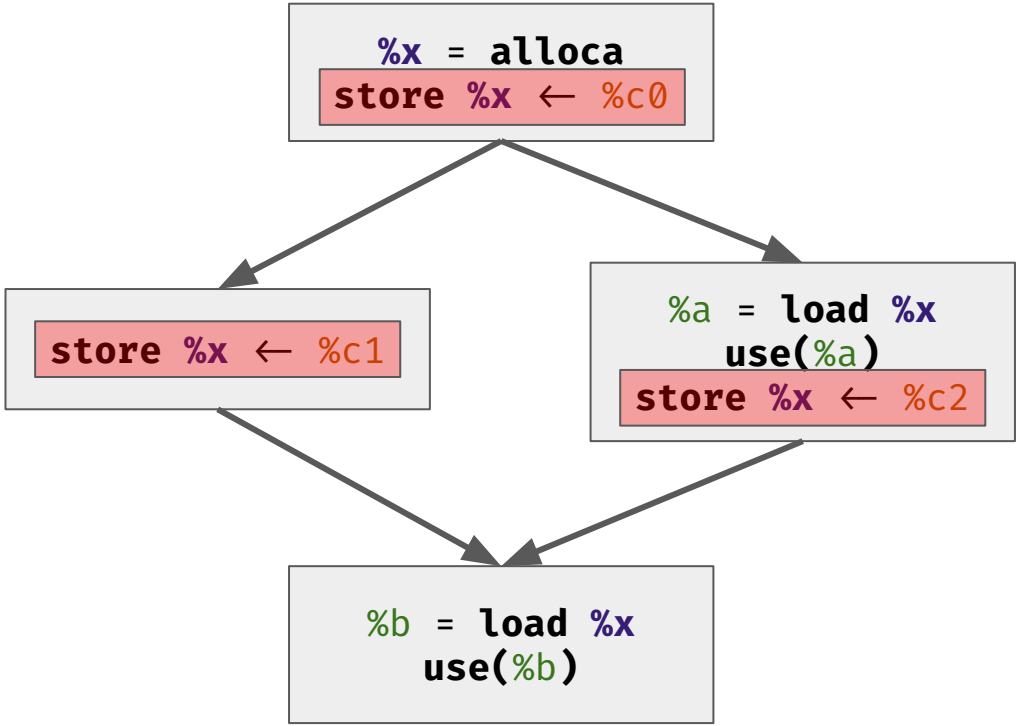
What is Mem2Reg?



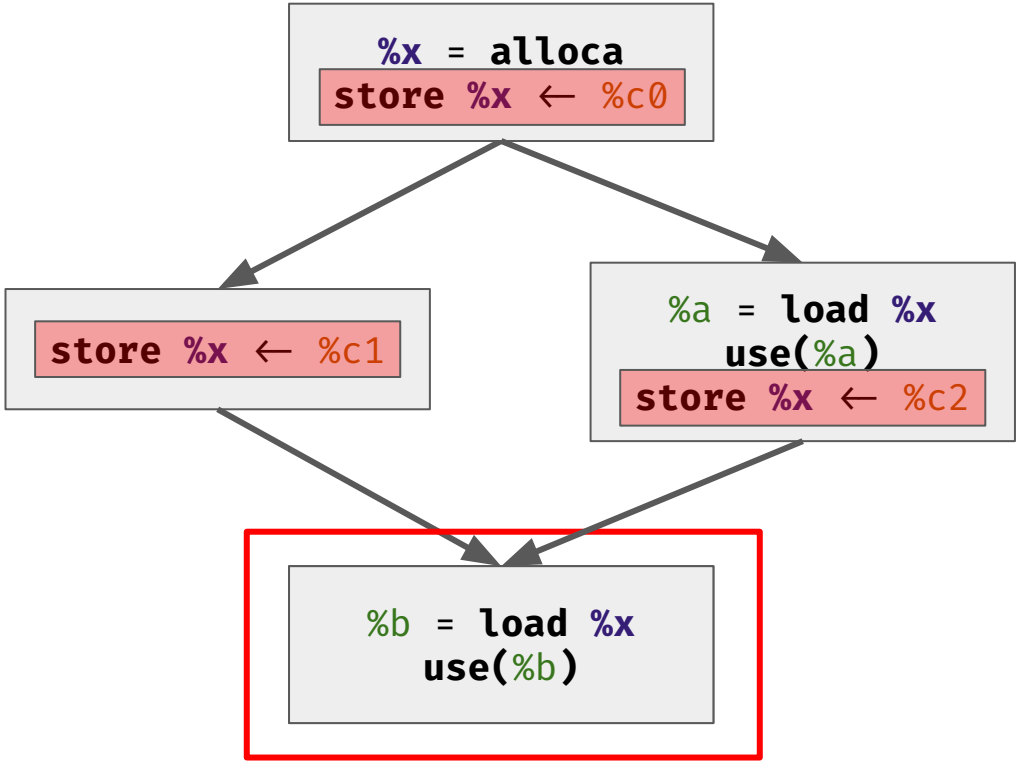
What is Mem2Reg?



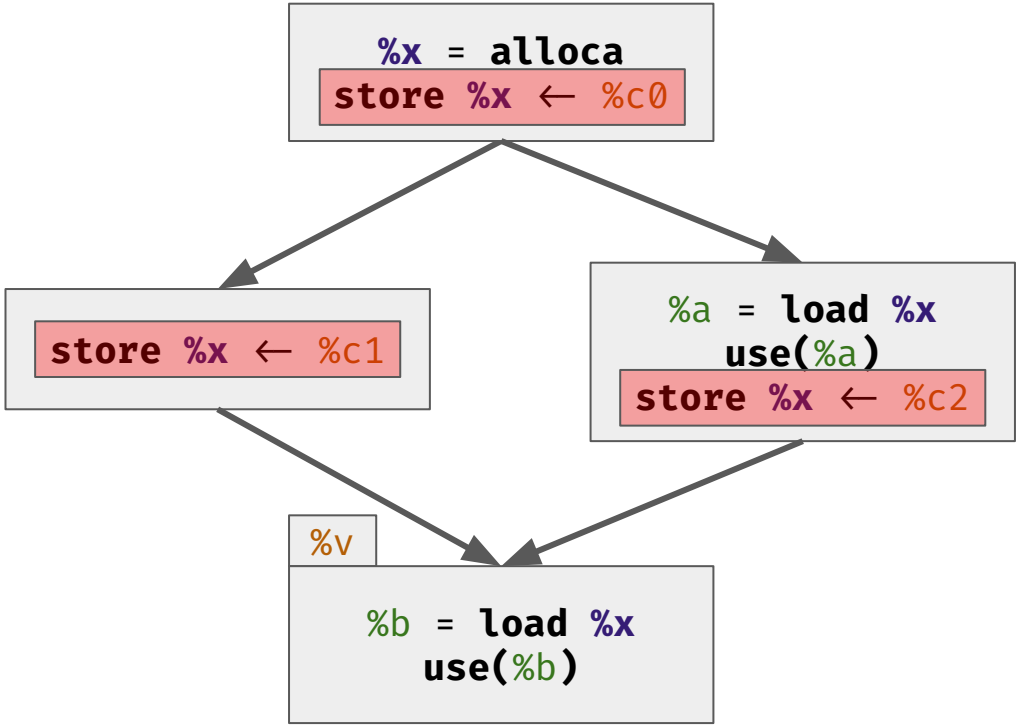
What is Mem2Reg?



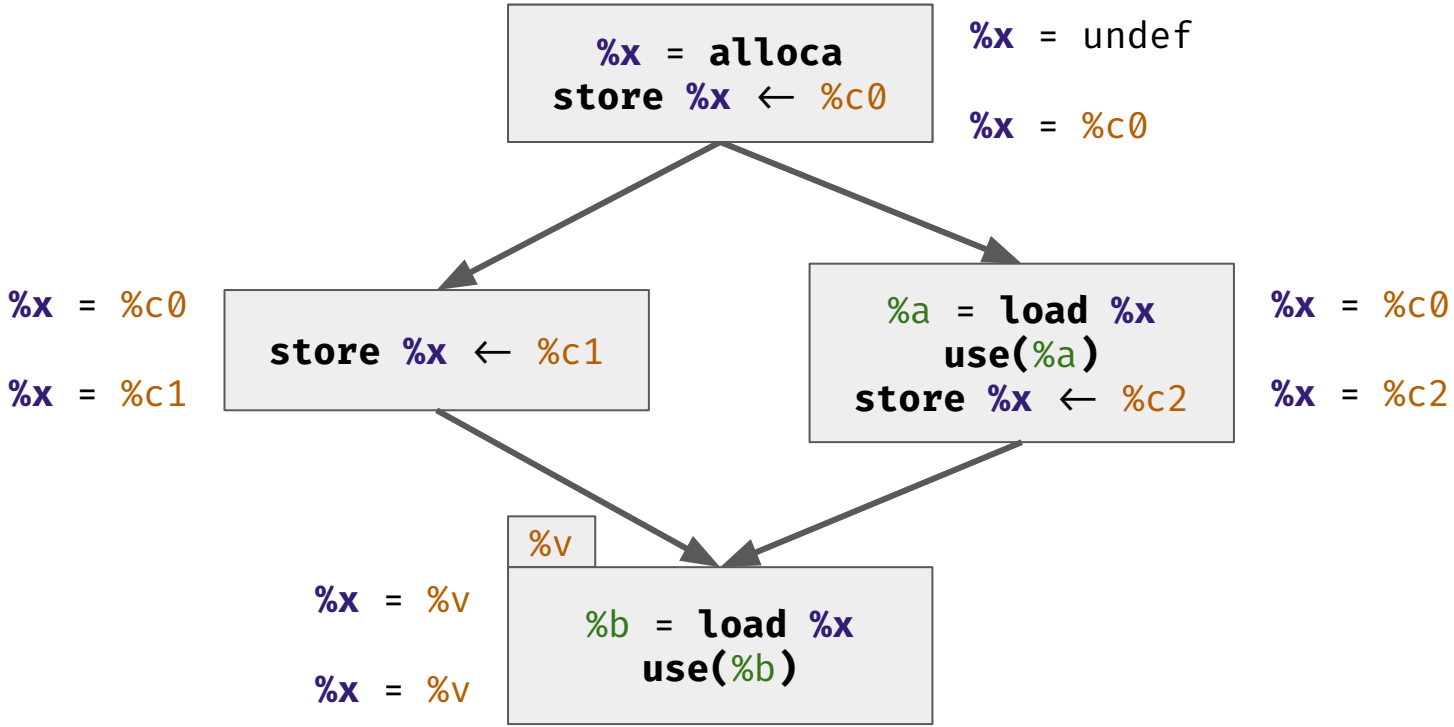
What is Mem2Reg?



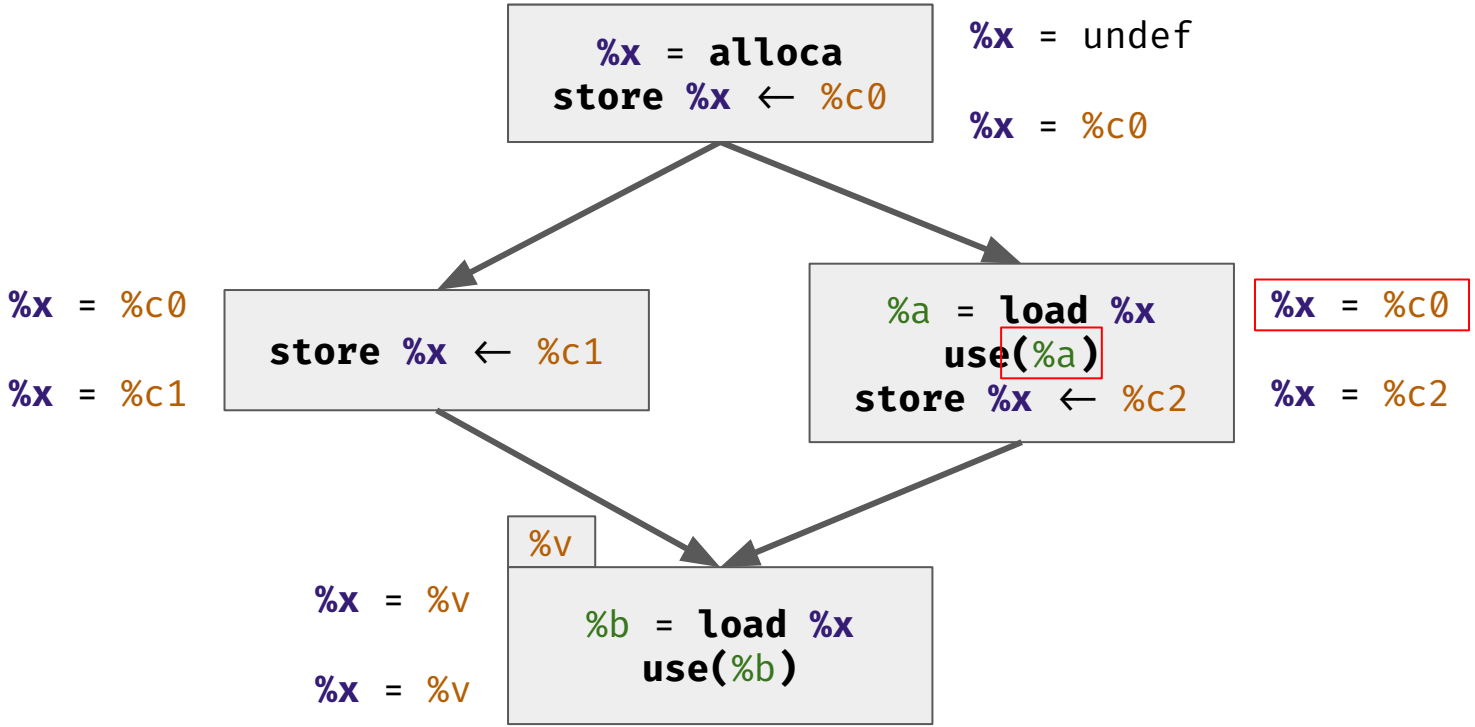
What is Mem2Reg?



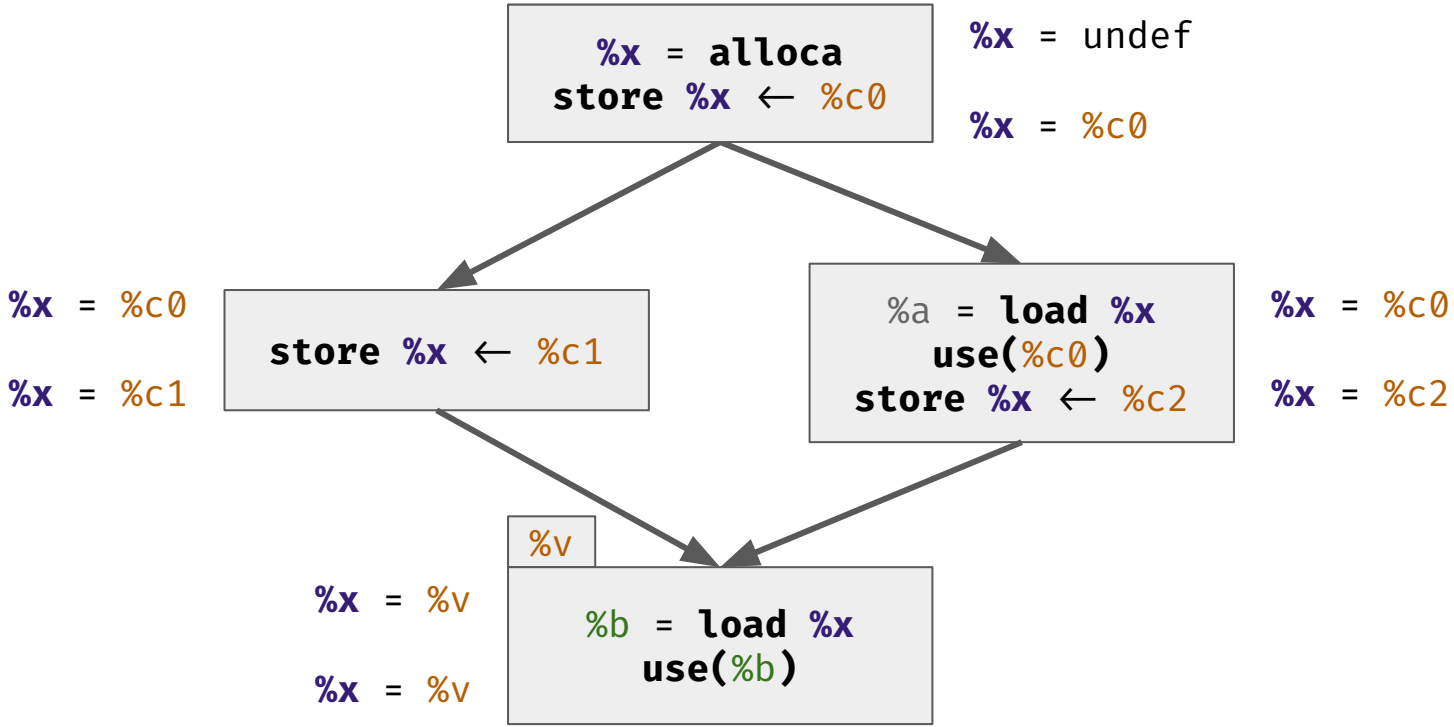
What is Mem2Reg?



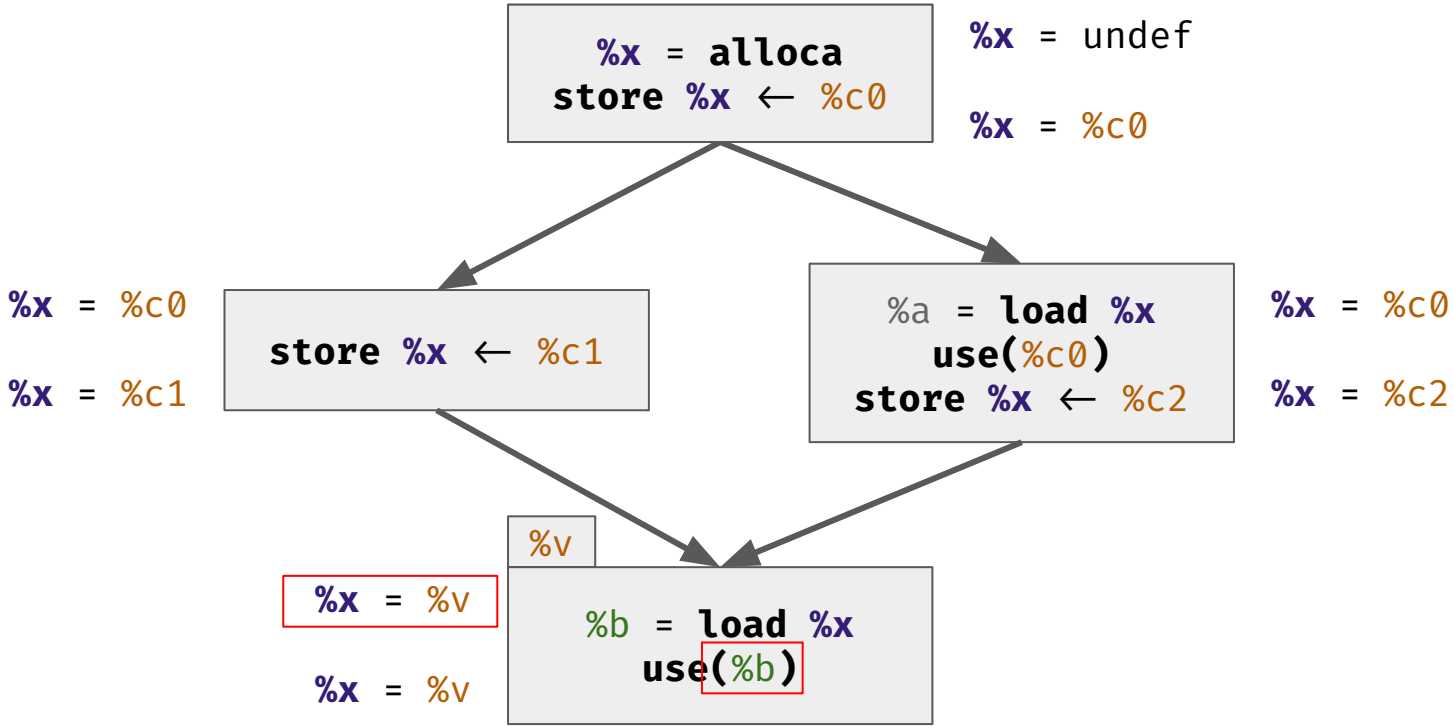
What is Mem2Reg?



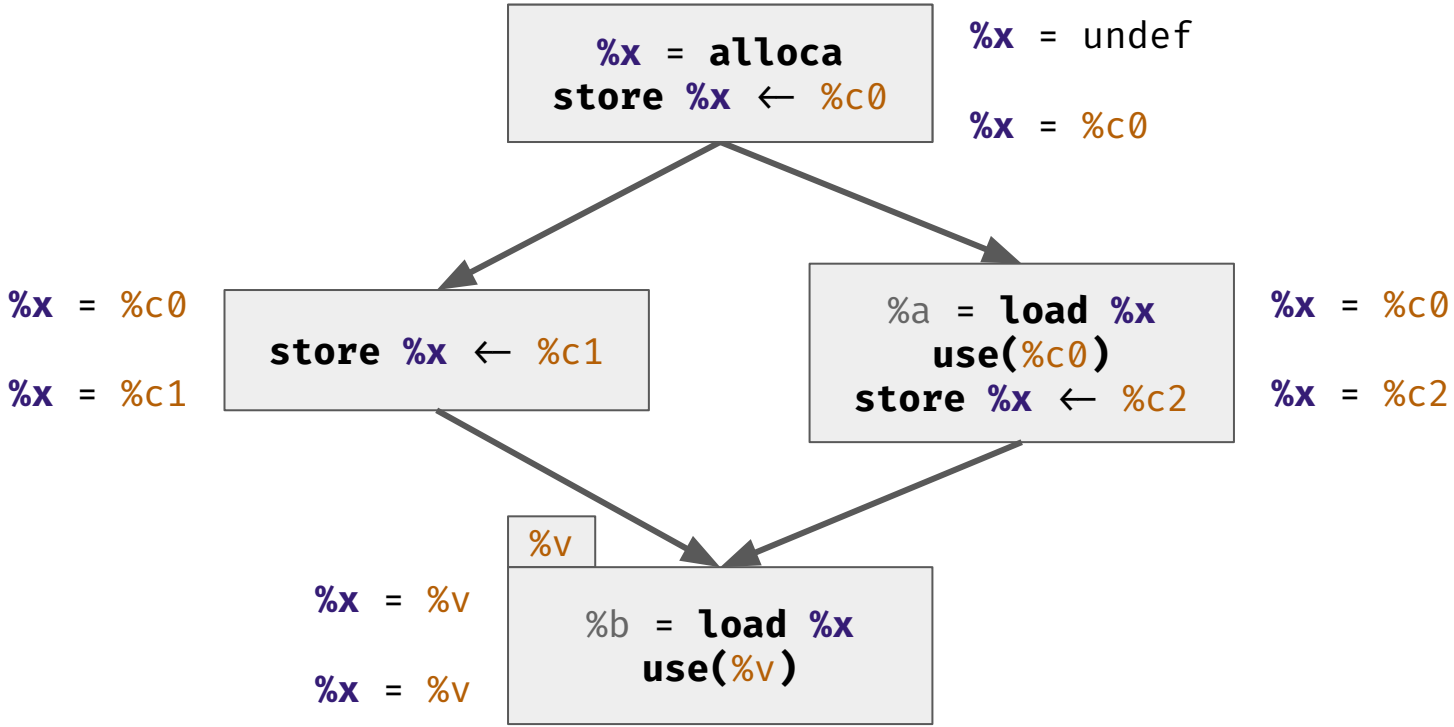
What is Mem2Reg?



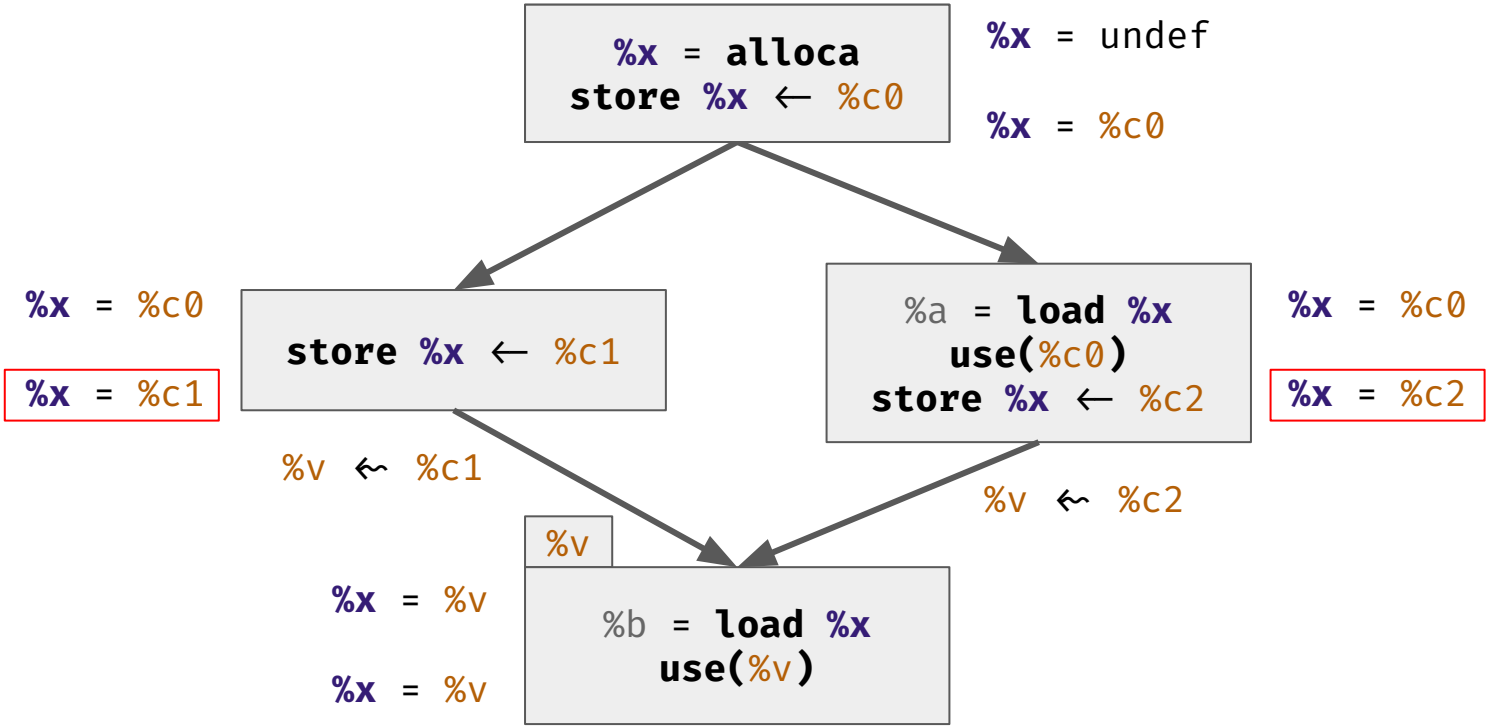
What is Mem2Reg?



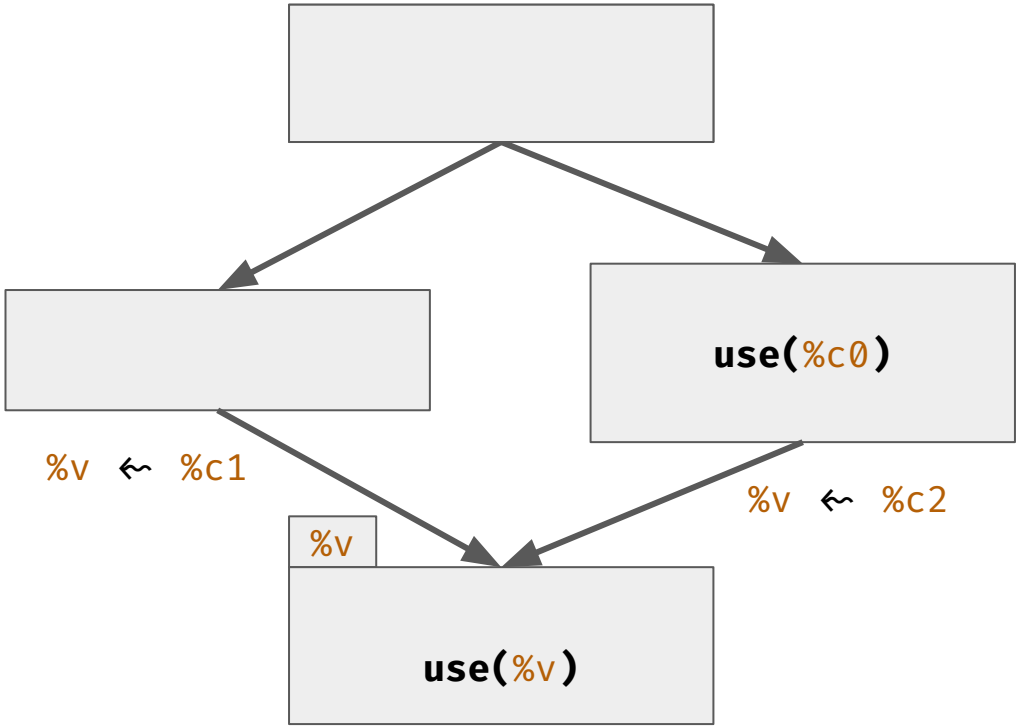
What is Mem2Reg?



What is Mem2Reg?



What is Mem2Reg?



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

Create MemorySlots

PromotableAllocationOpInterface for `list.alloca`

```
SmallVec<MemorySlot> getPromotableSlots();

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

std::optional<PromotableAllocationOpInterface>
AllocaOp::handlePromotionComplete(const MemorySlot &slot,
                                  Value defaultValue,
                                  OpBuilder &builder);
```

```
%ref = list.alloca()
  : !list.ref<!list.list<i32>>
```

```
MemorySlot {
  .ptr = %ref,
  .elemType = !list.list<i32>,
}
```

Replace usage

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

    list.store %const, %ref : !list.ref<i32>
    %l = list.load %ref : !list.ref<i32>

    return %l : i32
}
```

“blocking” uses

Check usage

PromotableMemOpInterface for `list.store`

```
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 blocking use is the ref, and is the slot pointer.
- How to remove them?
  We can just delete the op by returning
  `DeletionKind::Delete`

Check usage

PromotableMemOpInterface for `list.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 blocking use is the ref, and is the slot pointer.
- How to remove them?
  We need to replace uses then return
  `DeletionKind::Delete`

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

Implement it!

- Complete the implementations of the interfaces for **`list.alloca`**, **`list.load`** and **`list.store`**.

ListLang interpreter

MemorySlotInterfaces.td

It even works on control-flow!

But for our own control flow…

ListLang

```
1 (0..10).fold_int(0, |x, acc| x + acc)
```

Modelling control-flow

PromotableRegionOpInterface for `list.fold`

```
bool isRegionPromotable(const MemorySlot &slot,
                        const Region *region,
                        bool hasValueStores);

void setupPromotion(
    const MemorySlot &slot,
    Value reachingDef,
    bool hasValueStores,
    Map<Region *, Value> regionsToProcess
);

Value finalizePromotion(
    const MemorySlot &slot,
    Value entryReachingDef,
    bool hasValueStores,
    Map<Block *, Value> reachingAtBlockEnd,
    OpBuilder builder
);
```

Always!

Add the reaching definition block argument

Update terminators, add new result for the new reaching definition

Adding new results in MLIR

- Results in MLIR are immutable
- This requires creating a new operation…
- …but Mem2Reg requires regions to persist for state-tracking reasons!

When replacing results for Mem2Reg,
always **move** regions, do not clone them.

Helpers are available.

Implement it!

MemorySlotUtils.h

MemorySlotInterfaces.td

Open projects

- Implement the region interfaces for **`list.map`**
- Create an operation appending an integer to the last element of a list ref without loading or storing it, then add support for Mem2Reg.
- Add a **`list.sublist`** `%ref : !list.ref<...>` operation and the corresponding aliaser interface