MLIR Side Effect Modelling

Jeff Niu, Siddharth Bhat
LLVM Dev US 2023

What **isn’t** a side effect?

```
func @foo(%a: i32, %b: i32) {
  %c = add %a, %b
  %d = add %c, %b
  return %d
}
```

```
func @foo(%a: i32, %b: i32) {
  %d = add %c, %b
  %c = add %a, %b
  return %d
}
```

Legal

no side effects ~ "Can be rearranged freely"

What **isn't** a side effect?

```
func @bar(%p: i32, %q: i32) {
  %r = add %a, %b
  ... (LOTS OF CODE)
  %s = add %a, %b
  return %s
}
```

```
func @bar(%p: i32, %q: i32) {
  %r = add %p, %q
  ... (LOTS OF CODE)
  %s = %r
  return %s
}
```

Legal

no side effects ~ "output depends *only* on SSA inputs"

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  ...
  ...
  ...
  ...
  ...
  return %v2
}
```

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  ...
  ...
  ...
  return %v2
}
```

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32
  ...
  return %v2
}
```

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32
  ...
  return %v2
}
```

```
func @baz(%a: memref<?xi32>, %ix: index) {
  %v1 = memref.load %a [%ix] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%ix] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%ix] : i32
  ...
  return %v2
}
```

Illegal

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32
  %v2 = memref.load %a[%i] : i32
  return %v2
}
```

What **is** a side effect?

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32
  %v2 = memref.load %a[%i] : i32
  return %v2
}
```

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32
  %v2 = %v1
  return %v2
}
```

Illegal

What **is** a side effect?

```
func @baz(%mem: memref<?xi32>, %ix: index) {
  %v1 = memref.load %a [%ix] : i32
  %c42 = constant 42 : i32
  memref.store %c42, %a[%ix] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%ix] : i32
  %v2 = memref.load %mem [%ix] : i32
  return %v2
}
```

side effects ~= "output depends on *implicit state* (e.g. memory)"

Side effect modeling in MLIR

`SideEffect`

`EffectKind (TypeID)`

`Resource (singleton)`

`Affected (Value)`

Side effect modeling in MLIR

SideEffect

EffectKind (TypeID)

Resource (singleton)

Affected (Value)

EffectOpInterface

```
%2:2 = "i_have.side_effects"(%0, %1)
  : (pointer, index) -> pointer
```

SideEffect

SideEffect

SideEffect

It’s **super** abstract!

Example: `MemoryEffect`

`MemoryEffect`

EffectKind
- Alloc
- Free
- Read
- Write

Resource
- GPU shared
- Runtime handles
- JIT contexts
- ...

Affected
- memref
- alloca
- Global symbol
- ...

Example: MemoryEffect

```
func @baz(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32            store(%a)
  %c42 = constant 42 : i32
  memref.store %c42, %a[%i] : i32            load(%a)
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32          store(%a)
  %v2 = memref.load %a[%i] : i32             load(%a)
  return %v2
}
```

CSE: `MemoryEffects` in action!

```
func @cse_reads(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32              load(%a)
  %v2 = arith.addi %v1, %v1 : i32
  %v3 = memref.load %a[%i] : i32              load(%a)
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32            store(%a)
  %v4 = arith.addi %v2, %v3
  return %v4 : i32
}
```

No other effects

CSE: MemoryEffects in action!

```
func @cse_reads(%a: memref<?xi32>, %i: index) {
  %v1 = memref.load %a[%i] : i32            load(%a)
  %v2 = arith.addi %v1, %v1 : i32
  %v3 = memref.load %a[%i] : i32
  %c100 = constant 100 : i32
  memref.store %c100, %a[%i] : i32          store(%a)
  %v4 = arith.addi %v2, %v3
  return %v4 : i32
}
```

Is `MemoryEffect::Alloc` elidable?

```
func @unused_heap_alloc(%size: index, %align: index) {
  %0 = heap.aligned_alloc %size, %align : pointer<f32>      alloc(%0)
  return
}
```

`applyPatternsAndFoldGreedily` happily deletes this!

What about Leak Sanitizer?

```
func @unused_heap_alloc(%size: index, %align: index) {
  %0 = heap.aligned_alloc %size, %align : pointer<f32>
  return
}
```

Result: memory leak is "**optimized away**" by DCE

Other effect stuff

`effectOnFullRegion`: "if this side effect acts on every single value of resource" 🧐 😕

- **2** google search results (https://shorturl.at/kvwzB)
- **0** GitHub search results outside of llvm-project and forks (https://github.com/search?q=getEffectOnFullRegion+&type=code)

`stage`: Order of effects on the same operation

- An operation can `Read` and then `Free` a resource, if the stage of the `Read` is earlier than the `Free`

`RecursiveMemoryEffects`: Operation effects := union of all effects in region

ConditionallySpeculatable

TL;DR: can the op be hoisted?

Example: div

Potential UB

```
func @divs_safe(%lhs: index, %rhs: index) -> index {
  %idx0 = index.constant 0
  %0 = index.cmp eq(%rhs, %idx0)
  %1 = scf.if %0 -> index {
    scf.yield %idx0 : index
  } else {
    %2 = index.divs %lhs, %rhs
    scf.yield %2 : index
  }
  return %1 : index
}
```

What’s missing?

Memory effects on functions and function arguments

- Enables propagating memory effects to callers
- E.g. `readnone`, `readonly`, `read+write` (i.e. local to the caller)

`MemoryEffectOpInterface` “black hole”

- Super-privileged in MLIR infra
- Users are forced into an often substandard model to reuse upstream infra

Example: DCE

```
// Trait for enforcing that a side-effecting op is executed, even if it would be
// considered dead by MLIR (see b/195782952).
// The trait is implemented as a write effect for a fake resource which is
// ignored by side effect analysis, so it does not affect execution order
// constraints and control dependencies at all (for example, multiple ops with
// this trait do not have to execute in order).
def TF_MustExecute : MemoryEffects<[MemWrite<TF_MustExecuteResource>]>;
```

“Dear MLIR, please don’t delete this op”

Example: DCE

Whenever an op in the loop is modified, canonicalizer checks if the loop can be DCE'd

⇒ `RecursiveMemoryEffects` rewalks the whole body

⇒💥

```
func @big_loop(%lb: index, %ub: index) {
  %step = index.constant 1
  scf.for %i = %lb to %ub step %step {
    // Huge inlined loop body with more loops inside
  }
  return
}
```

Some Researchy Ideas for Side Effect Modelling`

Key Idea: Move side effects into the def-use chain, have interfaces for the `Ops`

Eg. `MemoryEffect → MemorySSA, Async → Async.token, poison`

Pro: Reuses MLIR's strengths! MemorySSA analysis becomes another dialect.

Con: UB does not fit into this model **:(**

Synthesis: def-use chain for those that can, Interfaces that which cannot.

[1] Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. Logical Methods in Computer Science, Volume 9, Issue 4, December 2013.
[2]Novillo, Diego. "Memory SSA-a unified approach for sparsely representing memory operations." *Proc of the GCC Developers' Summit*. 2007.

Even More Bleeding Edge Ideas! (Koka)

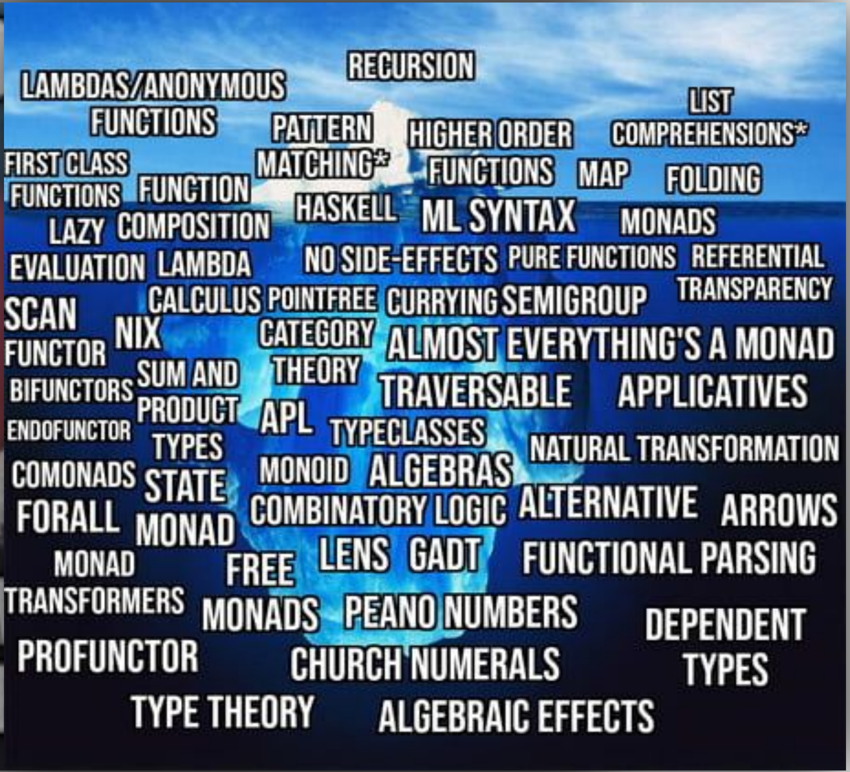
```
fun sqr    :
fun divide :
fun turing :
fun print  :
fun rand   :
```

The precise eff

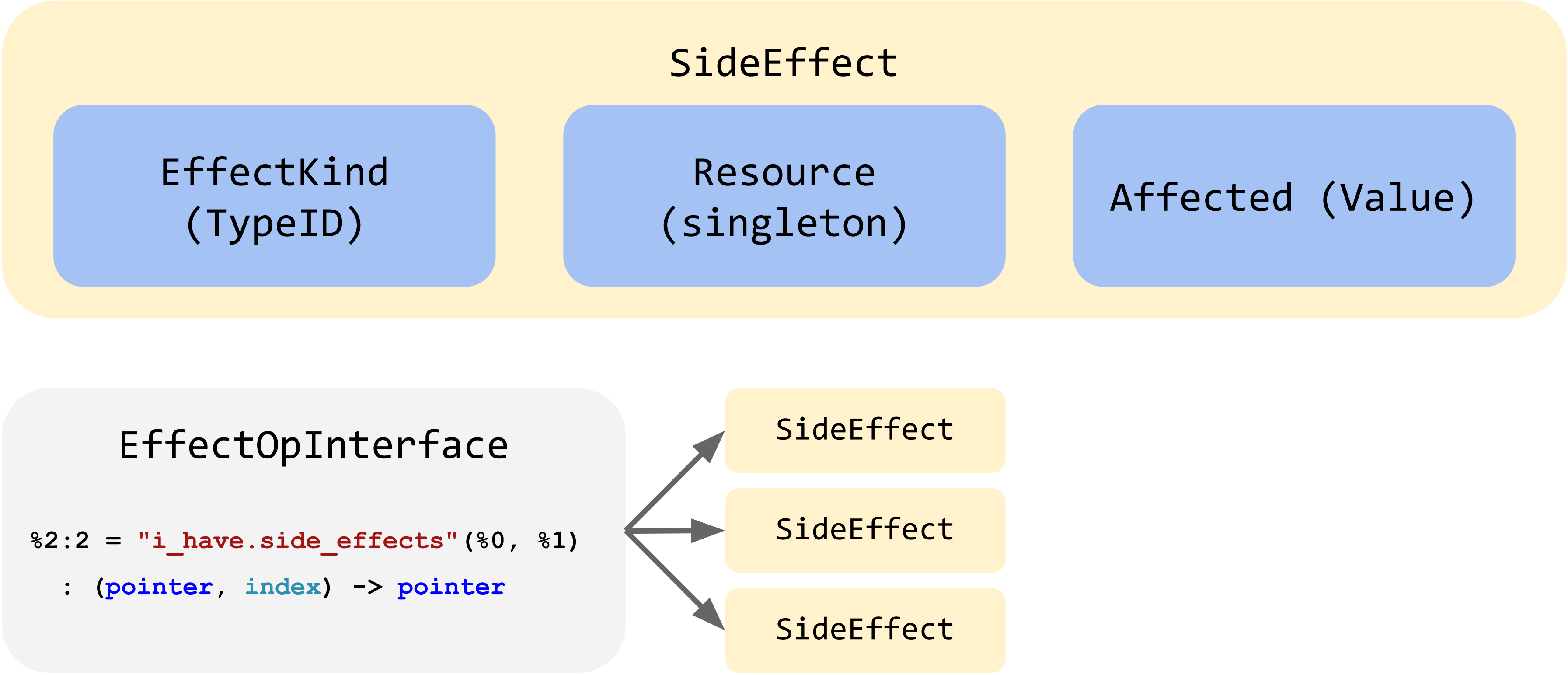
well-studied ca

and compilers

Japanese word



Conclusion


SideEffect
EffectKind (TypeID)
Resource (singleton)
Affected (Value)
EffectOpInterface
%2:2 = "i_have.side_effects"(%0, %1)
: (pointer, index) -> pointer
SideEffect
SideEffect
SideEffect