

Alive-mutate

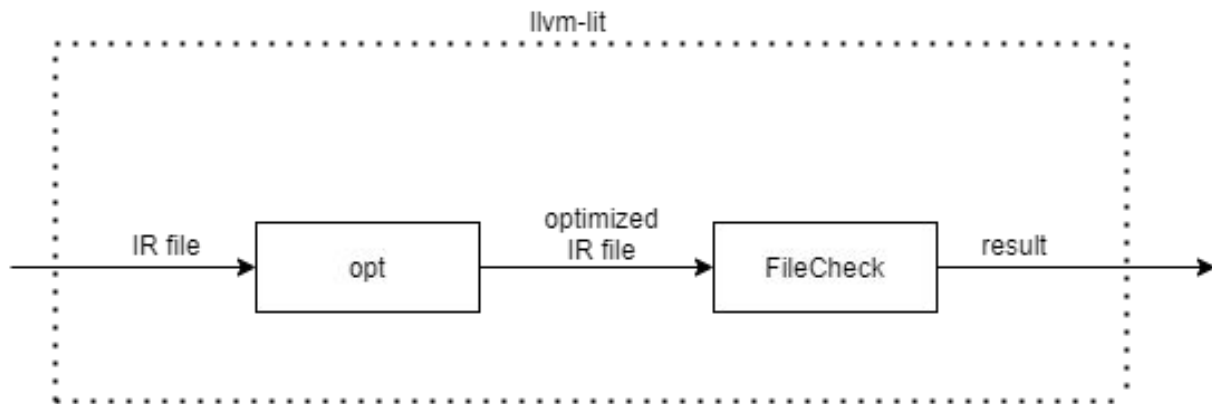
Yuyou Fan, John Regehr
University of Utah

a fuzzer that cooperates with Alive2 to find LLVM bugs

Let's start with testing...

There are two important problems in testing:

1. Writing a good test case
2. Knowing what each test case should do



A real example:

This is a test case from InstCombine Pass:

[canonicalize-clamp-like-pattern-between-negative-and-positive-thresholds.ll](#)

```
define i32 @t1_ult_slt_0(i32 %x, i32 %low, i32 %high) {  
  %t0 = icmp slt i32 %x, -16  
  %t1 = select i1 %t0, i32 %low, i32 %high  
  %t2 = add i32 %x, 16  
  %t3 = icmp ult i32 %t2, 144  
  %r = select i1 %t3, i32 %x, i32 %t1  
  ret i32 %r  
}
```

A real example:

This is a slightly mutated version of the previous test case:

```
define i32 @t1_ult_slt_0(i32 %x, i32 %low, i32 %high) {  
  %t0 = icmp slt i32 %x, 0  
  %t1 = select i1 %t0, i32 %low, i32 %high  
  %t2 = icmp ult i32 %x, 65536  
  %1 = xor i1 %t2, true  
  %r = select i1 %1, i32 %x, i32 %t1  
  ret i32 %r  
}
```

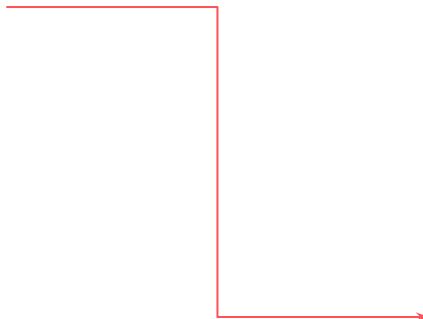
Easy to understand!

```
if x >= 65536  
  return x  
else if x < 0  
  return low  
else  
  return high
```

After calling opt with InstCombine

The thing changed a little bit after opt....

```
define i32 @t1_ult_slt_0(i32 %x, i32 %low, i32 %high) {  
  %1 = icmp slt i32 %x, 0  
  %2 = icmp sgt i32 %x, 65535  
  %3 = select i1 %1, i32 %low, i32 %x  
  %4 = select i1 %2, i32 %high, i32 %3  
  ret i32 %4  
}
```



```
if x > 65535  
  return high  
else if x < 0  
  return low  
else  
  return x
```

The transformation is wrong!

Consider both functions with input (x=2, low=0, high=1)

```
if x >= 65536
  return x
else if x < 0
  return low
else
  return high
```

opt

```
if x > 65535
  return high
else if x < 0
  return low
else
  return x
```

Before opt, it returns 1.

After opt, it returns 2.

<https://github.com/llvm/llvm-project/issues/53252>

Solving the second problem by Alive2!

Alive2 provides refinement check for LLVM IR transformations.

By using Alive2, programmers can be more clear about “Knowing what a test case should do”.

Transformation doesn't verify!

ERROR: Value mismatch

Example:

```
i32 %x = #x00000002 (2)
i32 %low = #x00000000 (0)
i32 %high = #x00000001 (1)
```

Source:

```
i1 %t0 = #x0 (0)
i32 %t1 = #x00000001 (1)
i1 %t2 = #x1 (1)
i1 %1 = #x0 (0)
i32 %r = #x00000001 (1)
```

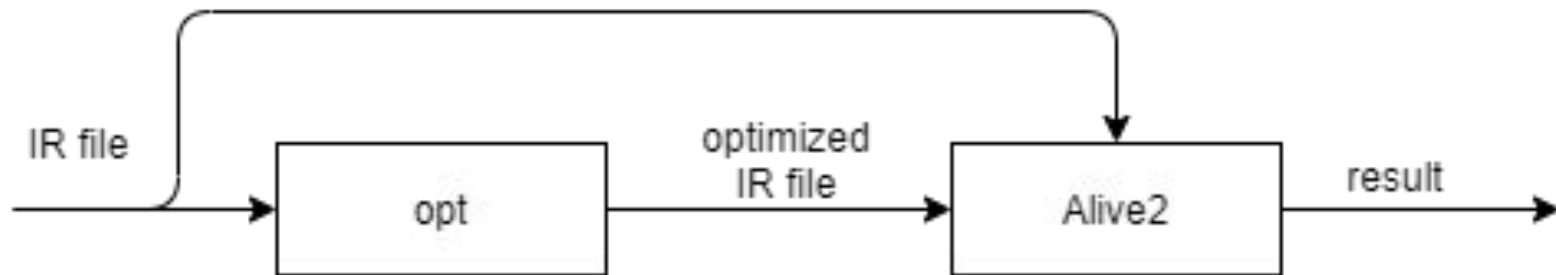
Target:

```
i1 %1 = #x0 (0)
i1 %2 = #x0 (0)
i32 %3 = #x00000002 (2)
i32 %4 = #x00000002 (2)
Source value: #x00000001 (1)
Target value: #x00000002 (2)
```

Validating with Alive2

We first send the IR file into optimization pass. Alive2 receives both IR files as input and produce result of the transformation correctness.

Our new workflow:



Let's try to solve the first problem!

If a test case have found a bug before, a modified version of it should be a good test case too because of defects clustering.

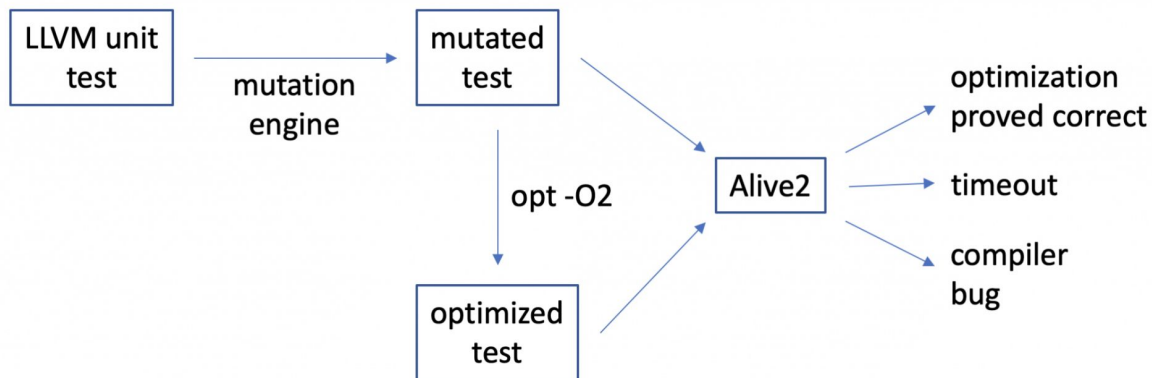
LLVM repository has so many good test cases so we can use them to generate more for saving people's effort.

As a result, we can build to a mutation engine to do such things.

Alive-mutate

Alive-mutate is a tool that cooperates with Alive2

Alive-mutate would automatically generate mutated IR files based on an input, and send them to Alive2 to verify the correctness of transformation.



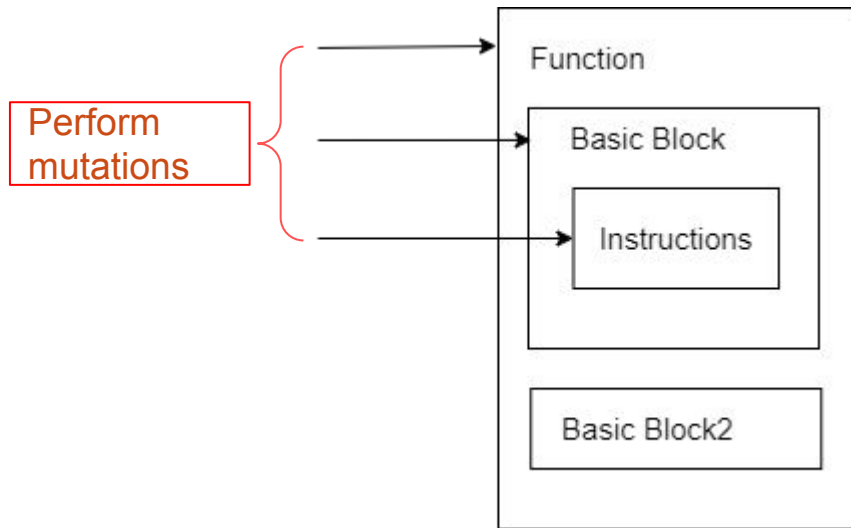
Mutation Features

Mutations

We support mutations on function, basic block and instruction levels.

It's easy to add or disable some mutations we want.

We always generate valid IRs thus we can maintain a high throughput



Set/unset attributes of a function and its arguments

```
define i1 @t(i64 %0, i64 %1, i64* %ptr) {
```

```
...  
}
```

```
define signext i1 @t(i64 %0, i64 signext %1, i64* nocapture dereferenceable(1) %ptr) #0{
```

```
...  
}
```

```
attributes #0 = { nofree }
```

Shuffle independent instructions

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){
```

```
  %v1 = load i64, i64* %ptr
```

```
  %v2 = ashr i64 %0, 23
```

```
  %v3 = trunc i64 %0 to i8
```

```
  %v4 = icmp ult i64 %0, %1
```

```
  ret i1 %v4
```

```
}
```

(1, 2, 3, 4)

(4, 3, 1, 2)

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){
```

```
  %v4 = icmp ult i64 %0, %1
```

```
  %v3 = trunc i64 %0 to i8
```

```
  %v1 = load i64, i64* %ptr
```

```
  %v2 = ashr i64 %0, 23
```


```
  ret i1 %v4
```

```
}
```

Random move an instruction

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %v2 = ashr i64 %v1, 23  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```

```
define i1 @t(i64 %0, i64 %1, i64* %ptr, i64 %2){  
  %v2 = ashr i64 %2, 23  
  %v1 = load i64, i64* %ptr  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```



Replace operands to others

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %v2 = ashr i64 %v1, 23  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %new = and i64 %v1, 1589327438  
  %v2 = ashr i64 %new, 23  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```


Special mutations on binary operators

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %v2 = ashr i64 %v1, 23  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %3 = lshr exact i64 23, %v1  
  %v3 = trunc i64 %3 to i8  
  %v4 = icmp ult i64 %v1, %3  
  ret i1 %v4  
}
```

Change the width of an integer definition

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %v2 = ashr i64 %v1, 23  
  %v3 = trunc i64 %v2 to i8  
  %v4 = icmp ult i64 %v1, %v2  
  ret i1 %v4  
}
```

v2: i64

newv2: i8

```
define i1 @t(i64 %0, i64 %1, i64* %ptr){  
  %v1 = load i64, i64* %ptr  
  %oldv2 = ashr i64 %v1, 23  
  %3 = trunc i64 %v1 to i8  
  %4 = trunc i64 23 to i8  
  %newv2 = ashr i8 %3, %4  
  %last = zext i8 %newv2 to i64  
  %v3 = trunc i64 %last to i8  
  %v4 = icmp ult i64 %v1, %last  
  ret i1 %v4  
}
```

Inline a function call

```
define void @f(){  
  %int64 = alloca i64  
  store i64 0, ptr %int64  
  %gep1 = getelementptr i64, ptr %int64, i64 0  
  call void @t(i64 0, i64 0, i64* %gep1)  
  ret void  
}
```

```
define void @f(){  
  %int64 = alloca i64  
  store i64 0, ptr %int64  
  %gep1 = getelementptr i64, ptr %int64, i64 0  
  %v1.i = load i64, i64* %gep1  
  %v2.i = ashr i64 %v1.i, 23  
  %v3.i = trunc i64 %v2.i to i8  
  %v4.i = icmp ult i64 %v1.i, %v2.i  
  ret void  
}
```

Remove a void function call

```
define void @f(){  
  %int64 = alloca i64  
  store i64 0, ptr %int64  
  %gep1 = getelementptr i64, ptr %int64, i64 0  
  call void @t(i64 0, i64 0, i64* %gep1)  
  ret void  
}
```

```
define void @f(){  
  %int64 = alloca i64  
  store i64 0, ptr %int64  
  %gep1 = getelementptr i64, ptr %int64, i64 0  
  ret void  
}
```



Experiment result

Result

29 bugs so far in LLVM.

We divided bugs into 3 categories.

1. 19 bugs generating wrong code.
2. 9 bugs causing crashes.
3. 1 test file is originally wrong.

Typical bugs

For generating wrong code, we found 10 bugs in arm64 backend. Most of them have simple form. For example:

```
define i64 @lsr_zext_i1_i64(i1 %b) {  
  %1 = zext i1 %b to i64  
  %2 = lshr i64 %1, 1  
  ret i64 %2  
}
```

It miscompiles due to global isel (fixed)

<https://github.com/llvm/llvm-project/issues/55129>

For crashing bugs, we found 5 related InstCombine pass. For example(fixed):

```
define i8 @smax_offset(i8 %x) {  
  %1 = add nuw nsw i8 50, %x  
  %m = call i8 @llvm.smax.i8(i8 %1, i8 -124)  
  ret i8 %m  
}
```

<https://github.com/llvm/llvm-project/issues/52884>

Thank for listening!