

Repurposing LLVM analyses in MLIR:
*Also there and back again across
the Tower of IRs*

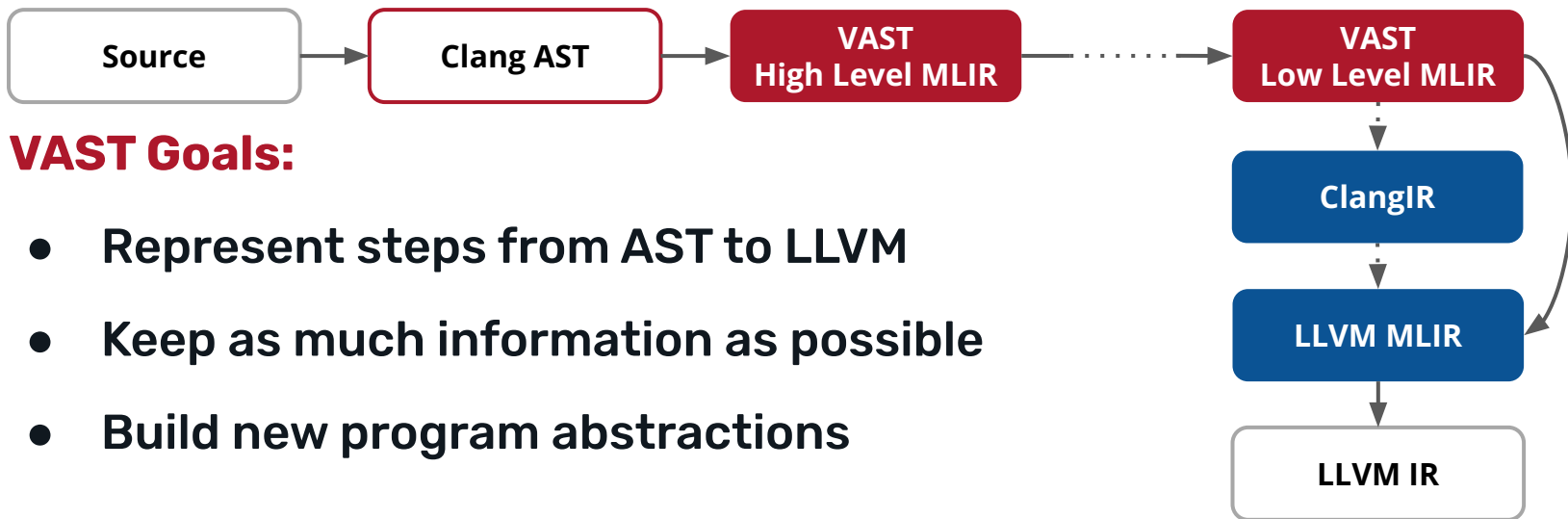
EuroLLVM 2024, 10th April, Henrich Lauko

VAST: Program analysis-focused compiler

- MLIR-based compiler for C/C++
- github.com/trailofbits/vast or try on [compiler explorer](#)

VAST: Program analysis-focused compiler

- MLIR-based compiler for C/C++
- github.com/trailofbits/vast or try on [compiler explorer](#)



VAST Goals:

- Represent steps from AST to LLVM
- Keep as much information as possible
- Build new program abstractions

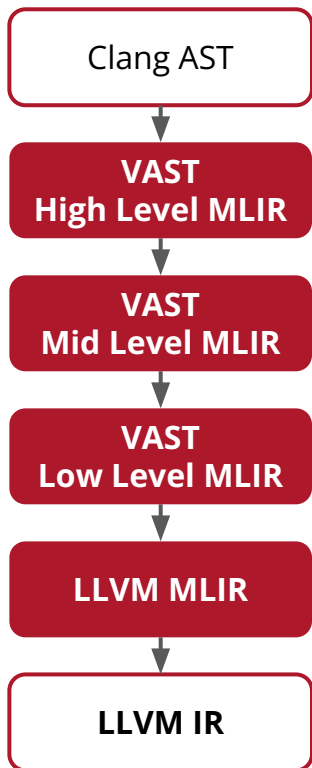
Re-use existing LLVM analyses, don't re-invent!



Goal: Want MLIR to benefit from pre-existing LLVM tools

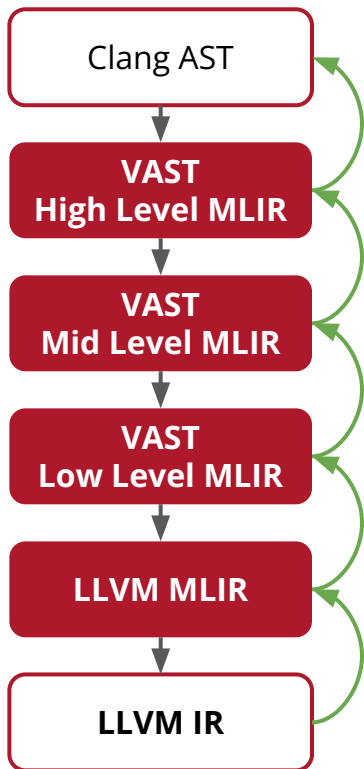
Solution: Lift LLVM analysis results into MLIR

Tower of IRs: Top-down view



MLIR Snapshots

Tower of IRs: Bottom-up view

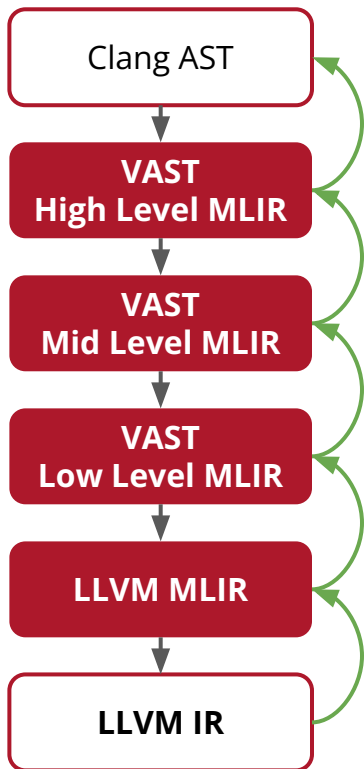


MLIR Snapshots + Provenance Links

=

**Bidirectional mapping between
MLIR modules**

Tower of IRs: The *real* multi-level IR



MLIR Snapshots + Provenance Links

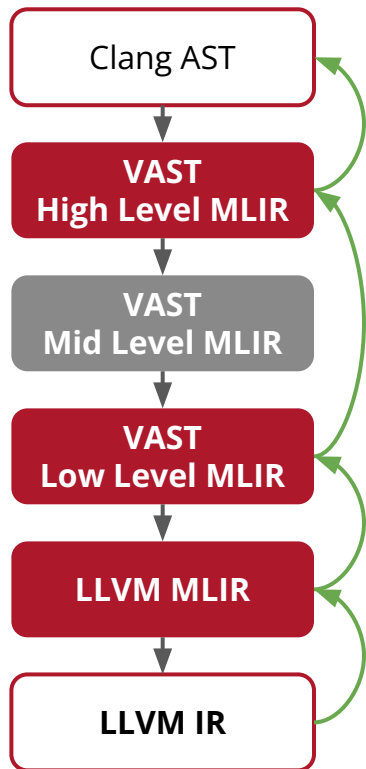
=

**Bidirectional mapping between
MLIR modules**

=

Multi-Level IR

Tower of IRs: The *real* multi-level IR



MLIR Snapshots + Provenance Links

=

**Bidirectional mapping between
MLIR modules**

=

Multi-Level IR

VAST Passes



splice-trailing-scopes

hl-to-hl-builtin

hl-dce

hl-lower-elaborated-types

hl-lower-typedefs

hl-to-std-types

lower-value-categories

emit-abi

lower-abi

hl-to-ll-vars

hl-to-ll-cf

hl-to-lazy-regions

hl-to-ll-geps

fn-args-to-alloca

... to-llvm

The approach is versatile

`mlir::generateLocationsFromIR`

This function generates new locations from the given IR by snapshotting the IR to the given output stream, and using the printed locations within that file.



Let's have some fun ()

```
1: void fun() {  
2:   int a = 2;  
3:   int b = a + 3;  
4:   int c = b * 13;  
5: }
```

© Constants based on today's integer sequence: <https://oeis.org/A100424>

A sieve transform applied three times to the positive integers.

Let's have some fun()

<pre> 1: void fun() { 2: int a = 2; 3: int b = a + 3; 4: int c = b * 13; 5: }</pre>	<pre> vast-front -vast-emit-mlir=hl 15: %2 = hl.var "c" : !hl.value<!hl.int> { 16: %4 = hl.ref %1 : !hl.lvalue<!hl.int> 17: %5 = hl.implicit_cast %4 LValueToRValue : !hl.int 18: %6 = hl.const #hl.integer<13> : !hl.int 19: %7 = hl.mul %5, %6 : !hl.int 20: hl.value.yield %7 : !hl.int 21: } loc(source:4)</pre>
---	---

© Constants based on today's integer sequence: <https://oeis.org/A100424>

A sieve transform applied three times to the positive integers.

Let's have some fun()

```
1: void fun() {
2:   int a = 2;
3:   int b = a + 3;
4:   int c = b * 13;
5: }
```

```
opt -vast-hl-lower-types
15: %2 = hl.var "c" : !hl.value<si32> {
16:   %4 = hl.ref %1 : !hl.lvalue<si32>
17:   %5 = hl.implicit_cast %4 LValueToRValue : si32
18:   %6 = hl.const #hl.integer<13> : si32
19:   %7 = hl.mul %5, %6 : si32
20:   hl.value.yield %7 : si32
21: } loc(high-level:15)
```

Let's have some fun()

```
1: void fun() {  
2:   int a = 2;  
3:   int b = a + 3;  
4:   int c = b * 13;  
5: }
```

```
opt -vast-emit-abi  
opt -vast-lower-abi  
opt -vast-hl-to-ll-func
```

- Skip snapshots of transformations that don't impact the interesting parts of MLIR
- Or we have identity maps between unchanged modules

Let's have some fun()

```
1: void fun() {
2:   int a = 2;
3:   int b = a + 3;
4:   int c = b * 13;
5: }
```

```
opt -vast-hl-to-ll-vars
10: %9 = ll.uninitialized_var : !hl.lvalue<si32>
11: %10 = hl.ref %8 : !hl.lvalue<si32>
12: %11 = hl.implicit_cast %10 LValueToRValue : si32
13: %12 = hl.const #hl.integer<13> : si32
14: %13 = hl.mul %11, %12 : (si32, si32) -> si32
15: %14 = ll.initialize %9, %13 loc(hl-to-ll-func:21)

opt -vast-hl-to-ll-cf
opt -vast-hl-to-lazy-regions
opt -vast-hl-to-ll-geps
```

Let's have some fun()

```
1: void fun() {
2:   int a = 2;
3:   int b = a + 3;
4:   int c = b * 13;
5: }
```

```
opt -vast-hl-lower-value-categories
10: %6 = ll.alloca : !ll.ptr<si32>
11: %7 = ll.load %2 : si32
13: %8 = hl.const #hl.integer<13> : si32
14: %9 = hl.mul %7, %8 : (si32, si32) -> si32
15: ll.store %6, %9 loc(hl-to-ll-geps:15)
```


Let's have some fun()

```
1: void fun() {
2:   int a = 2;
3:   int b = a + 3;
4:   int c = b * 13;
5: }
```

-vast-to-llvm

```
11: %8  = llvm.mlir.constant(1 : index)
12: %9   = llvm.alloca %8 x i32
13: %10  = llvm.load %4
14: %11  = llvm.mlir.constant(13 : i32)
15: %12  = llvm.mul %10, %11
16: llvm.store %12, %9 loc(hl-lower-value-categories:15)
```

Let's have some fun()

```
1: void fun() {  
2:   int a = 2;  
3:   int b = a + 3;  
4:   int c = b * 13;  
5: }
```

```
define void @fun() {  
    %1 = alloca i32, i64 1, align 4  
    store i32 2, ptr %1, align 4  
    %2 = alloca i32, i64 1, align 4  
    %3 = load i32, ptr %1, align 4  
    %4 = add i32 %3, 3  
    store i32 %4, ptr %2, align 4  
    %5 = alloca i32, i64 1, align 4  
    %6 = load i32, ptr %2, align 4  
    %7 = mul i32 %6, 13  
    store i32 %7, ptr %5, align 4  
    ret void  
}
```

Dependence analysis



```
1: void fun() {  
2:   int a = 2;  
3:   int b = a + 3;  
4:   int c = b * 13;  
5: }
```

```
define void @fun() {  
    %1 = alloca i32, i64 1, align 4  
    store i32 2, ptr %1, align 4  
    %2 = alloca i32, i64 1, align 4  
    %3 = load i32, ptr %1, align 4  
    %4 = add i32 %3, 3  
    store i32 %4, ptr %2, align 4  
    %5 = alloca i32, i64 1, align 4  
    %6 = load i32, ptr %2, align 4  
    %7 = mul i32 %6, 13  
    store i32 %7, ptr %5, align 4  
    ret void  
}
```

Walk back the Tower of IRs

```

1: void fun() {
2:   int a = 2;
3:   int b ← a + 3;
4:   int c = b * 13;
5: }

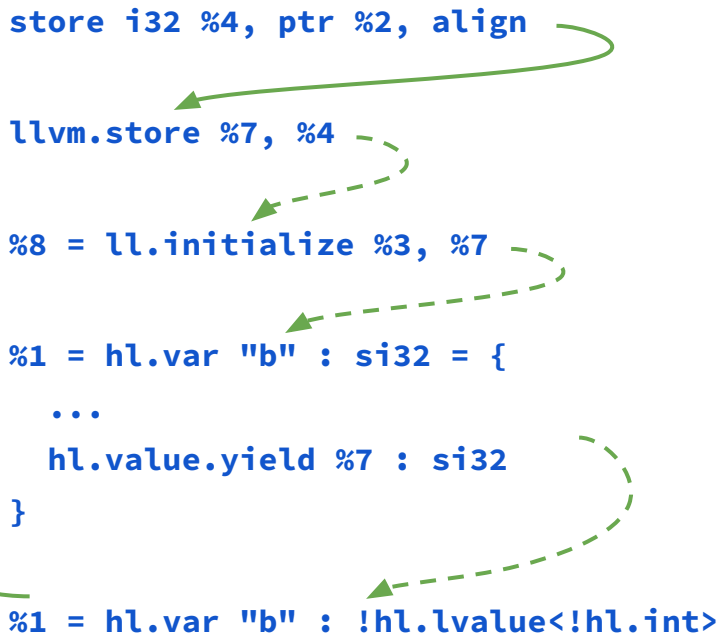
```

**Gather dependencies
across layers**

```

store i32 %4, ptr %2, align
llvm.store %7, %4
%8 = ll.initialize %3, %7
%1 = hl.var "b" : si32 = {
  ...
  hl.value.yield %7 : si32
}
%1 = hl.var "b" : !hl.lvalue<!hl.int>

```



Genericity of the approach



Similar approach **is applicable beyond VAST** in other tools.

Need to be **cautious about** the aggressiveness of **transformations**.

Overly aggressive transformations may **hinder cross-layer linking**.

There and back again across the tower of IRs

Leverage LLVM-based analyses in MLIR toolchains



<https://github.com/trailofbits/vast>

Single layer on compiler-explorer, the tower soon.