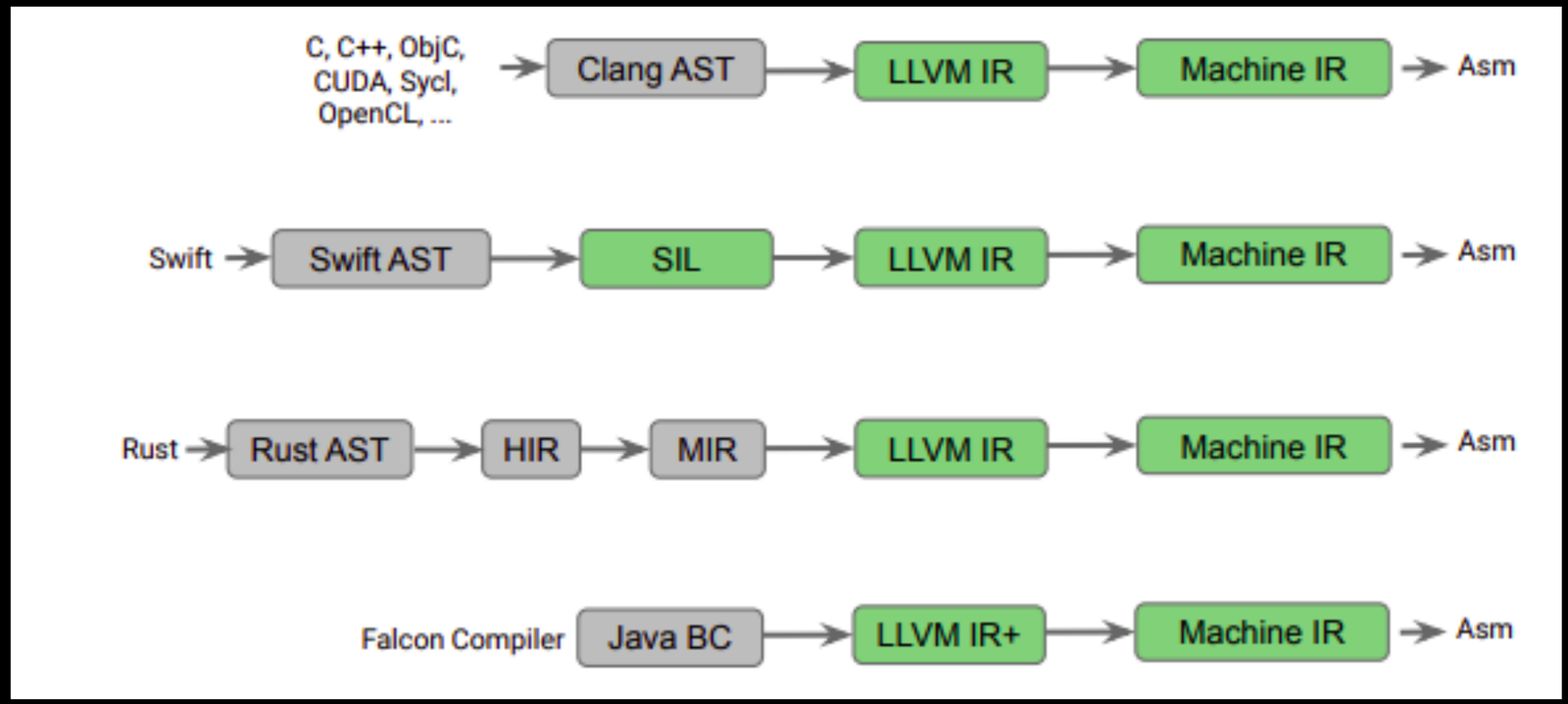
MLIR GENERATION

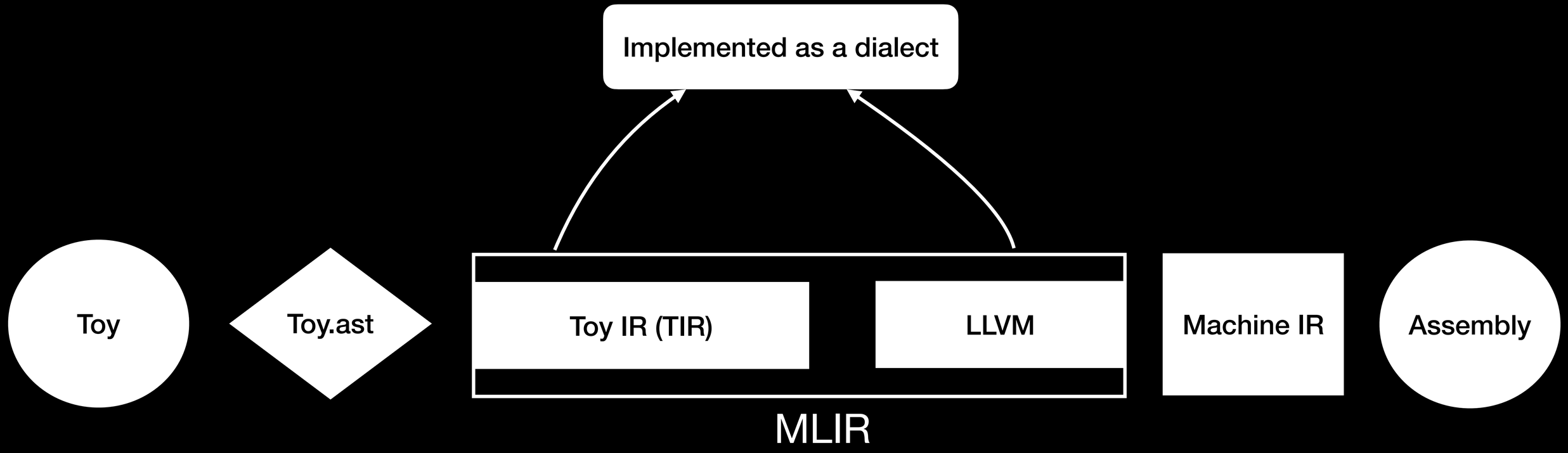
First Update on the generation of LLVM for self defined Language

Akhilarka J
Aditya Prasanna

What has been happening?



MLIR - Multi Level Intermediate Representation



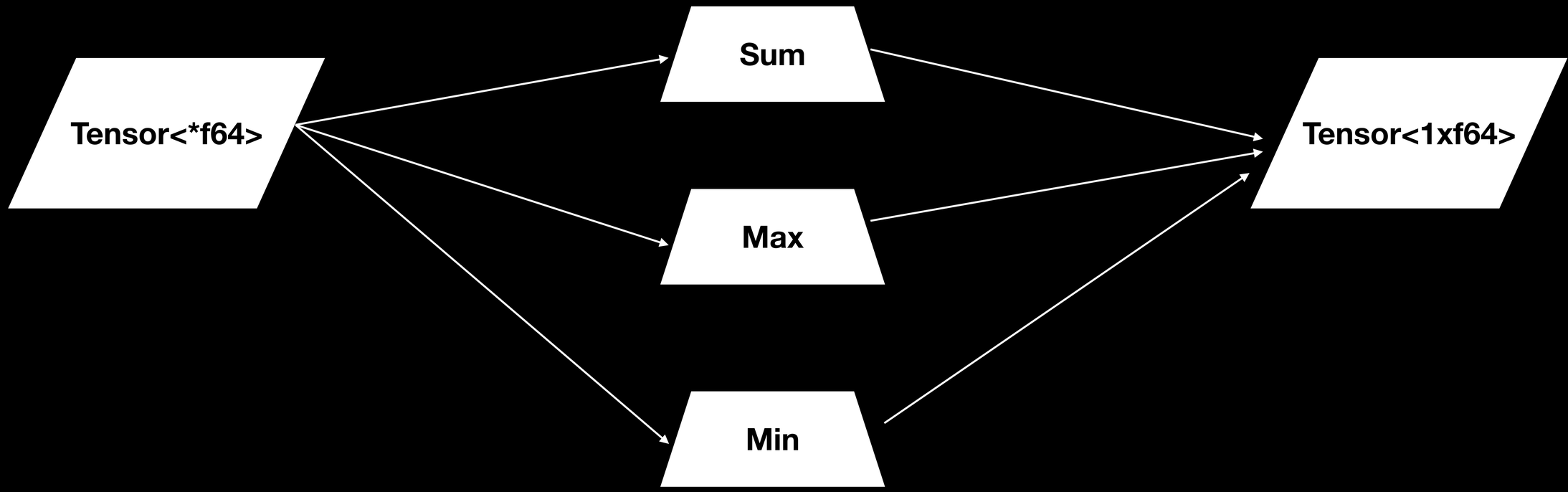
The Sim Language.

A simple language created to demonstrate the MLIR workflow

Data Type Input

In-built Operations

Data Type Output



Mid Review

A visualisation of the intermediate representations achieved

```
Module:
  Function
    Proto 'main' @example.sim:1:1
    Params: []
    Block {
      VarDecl a<> @example.sim:3:2
        Literal: <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00] @example.sim:3:10
      VarDecl b<> @example.sim:4:2
        Call 'max' [ @example.sim:4:10
          var: a @example.sim:4:14
        ]
      VarDecl c<> @example.sim:5:2
        Call 'min' [ @example.sim:5:10
          var: a @example.sim:5:14
        ]
      VarDecl d<> @example.sim:6:2
        Call 'sum' [ @example.sim:6:10
          var: a @example.sim:6:14
        ]
      Print [ @example.sim:7:2
        var: a @example.sim:7:8
      ]
    } // Block
```

AST

```
def main()
{
    val a = [1, 2, 3];
    val b = max(a);
    val c = min(a);
    val d = sum(a);
    print(a);
}
```

Input Dialect (.sim File)

```
module {
  func @main() {
    %0 = sim.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00]> : tensor<3xf64>
    %1 = sim.max(%0 : tensor<3xf64>) to tensor<*xf64>
    %2 = sim.min(%0 : tensor<3xf64>) to tensor<*xf64>
    %3 = sim.sum(%0 : tensor<3xf64>) to tensor<*xf64>
    sim.print %0 : tensor<3xf64>
    sim.return
  }
```

sim.mlir

The Roadblocks !!

A visualisation of what went wrong and needs work to be pruned

```
 input is .mlir file

sim.mlir:3:10: error: 'sim.max' op result #0 must be tensor of 64-bit float values, but got 'f64'
    %1 = sim.max(%0 : tensor<3xf64>) to f64
         ^
sim.mlir:3:10:       see current operation: %1 = "sim.max"(%0) : (tensor<3xf64>) -> f64
Error can't load file sim.mlir
```

Improper Type Conversion MLIR<Value> not able to be typecast to llvm::memref type

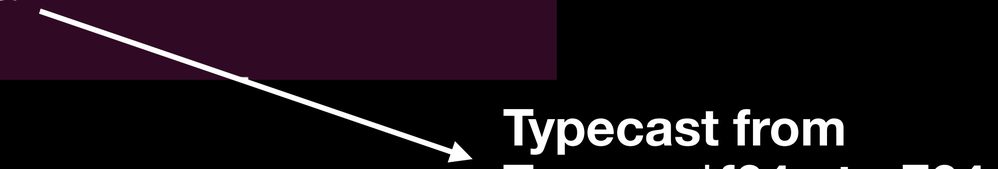
```
[HNS-MacBook-Pro:llvm-project hnprasannaiyengar$ build/bin/toyc-ch10 sim.mlir --emit=mlir-affine

 input is .mlir file


 isLoweringToAffine


infering min op shape

Bye Worldmin op lowering
sum op lowering
sim.mlir:3:10: error: 'std.cmpf' op requires the same shape for all operands and results
    %1 = sim.max(%0 : tensor<3xf64>) to tensor<*xf64>
         ^
sim.mlir:3:10:       see current operation: %4 = "std.cmpf"(%3, %3) {predicate = 4 : i64} : (memref<3xf64>, memref<3xf64>) -> i1
```

Dimension clash in std::cmpf as the input and output dimension expected to be the same. Need clarity on mlir::tensorType

Perfecting the SumOp.

Attempt on generation of LLVM for SumOp

```
module {
  func @main() {
    %0 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00]> : tensor<3xf64>
    %1 = toy.sum(%0 : tensor<3xf64>) to f64
    toy.print %1 : f64
    toy.return
  }
```



Typecast from
Tensor<*f64> to F64

```
def SumOp : Toy_Op<"sum", [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  let summary = "sum operation";

  let arguments = (ins F64Tensor:$input);
  let results = (outs F64);

  let assemblyFormat = [{
    `(` $input `:` type($input) `)` attr-dict `to` type(results)
  }];

  let builders = [
    OpBuilder<"OpBuilder &b, OperationState &state, Value input">
  ];

  let verifier = [{ return ::verify(*this); }];

}
```

Input File

```
1 def main() {
2   var d = [1, 2, 3];
3   var e = sum(d);
4   print(e);
5 }
```

Roadblocks (SumOp)

Issues faced while lowering SumOp to a target dialect

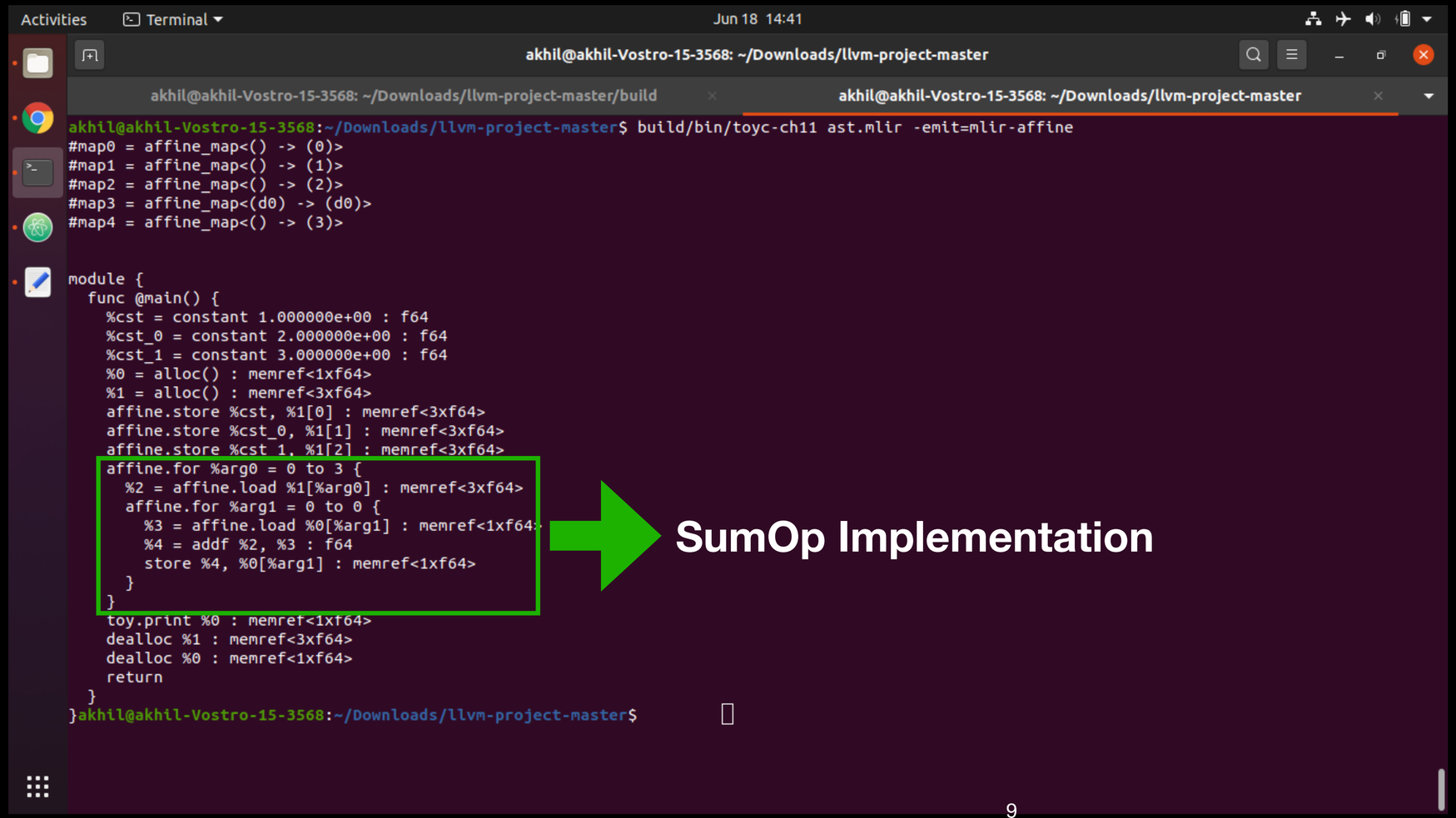
```
struct SumOpLowering : public ConversionPattern {
  SumOpLowering(MLIRContext *ctx)
      : ConversionPattern(toy::SumOp::getOperationName(), 1, ctx) {}

  LogicalResult
  matchAndRewrite(Operation *op, ArrayRef<Value> operands,
                  ConversionPatternRewriter &rewriter) const final {
    auto loc = op->getLoc();
    std::cout << "\n sum op lowering\n";
    mlir::MLIRContext context;
    //ArrayRef (const std::vector< T, A > &Vec)
    long int * data = (long int *)malloc(sizeof(long int));

    auto v = llvm::ArrayRef<long int>(data,1);

    auto memRefType = MemRefType::get(v, FloatType::get(mlir::StandardTypes::F64, &context));
    //auto memRefType = convertTensorToMemRef(tensorType);
    auto alloc = insertAllocAndDealloc(memRefType, loc, rewriter);
    //auto alloc = rewriter.create<AllocOp>(loc, memRefType);
    ArrayRef<Value> memRefOperands;
    ArrayRef<Value> loopIvs;
    toy::SumOpOperandAdaptor SumAdaptor(memRefOperands);
    Value input = SumAdaptor.input();
    auto addval = rewriter.create<LoadOp>(loc, input, loopIvs);

    rewriter.create<AddFOp>(loc, alloc, addval);
    rewriter.replaceOp(op, alloc);
    /*
```

Segmentation Fault!!

Experimenting with some implementations using the affine dialect. Aim to use std::AddFOp but trouble iterating through the ArrayRef<Value> and creating a memref to store the result of the addition.

SumOp v1

Not taking optimal condition into consideration.



SumOp Implementation

Improper initial assignment to sum store.

Affine Lowering

The Simple Language.

The Work Flow Summary

```
def main() {
  var a = [1, 2, 3, 4];
  var b = sum(a);
  var c = max(a);
  var d = min(a);
  print(b);
  print(c);
  print(d);
}
```

```
module {
  func @main() {
    %0 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00]> : tensor<4xf64>
    %1 = toy.sum(%0 : tensor<4xf64>) to tensor<1xf64>
    %2 = toy.max(%0 : tensor<4xf64>) to tensor<1xf64>
    %3 = toy.min(%0 : tensor<4xf64>) to tensor<1xf64>
    toy.print %1 : tensor<1xf64>
    toy.print %2 : tensor<1xf64>
    toy.print %3 : tensor<1xf64>
    toy.return
  }
```

```
%c0_3 = constant 0 : index
%cst_4 = constant 0.000000e+00 : f64
store %cst_4, %2[%c0_3] : memref<1xf64>
affine.for %arg0 = 0 to 4 {
  %4 = affine.load %3[%arg0] : memref<4xf64>
  %5 = affine.load %2[%c0_3] : memref<1xf64>
  %6 = addf %5, %4 : f64
  affine.store %6, %2[%c0_3] : memref<1xf64>
}
%c0_5 = constant 0 : index
%cst_6 = constant 1.1754943508222875E-38 : f64
affine.store %cst_6, %1[%c0_5] : memref<1xf64>
affine.for %arg0 = 0 to 4 {
  %4 = affine.load %3[%arg0] : memref<4xf64>
  %5 = affine.load %1[%c0_5] : memref<1xf64>
  %6 = cmpf "olt", %4, %5 : f64
  %7 = select %6, %5, %4 : f64
  affine.store %7, %1[%c0_5] : memref<1xf64>
}
%c0_7 = constant 0 : index
%cst_8 = constant 3.4028234663852886E+38 : f64
affine.store %cst_8, %0[%c0_7] : memref<1xf64>
affine.for %arg0 = 0 to 4 {
  %4 = affine.load %3[%arg0] : memref<4xf64>
  %5 = affine.load %0[%c0_7] : memref<1xf64>
  %6 = cmpf "olt", %4, %5 : f64
  %7 = select %6, %4, %5 : f64
  affine.store %7, %0[%c0_7] : memref<1xf64>
}
```

The Simple Language.

ConvOp Work Flow Summary

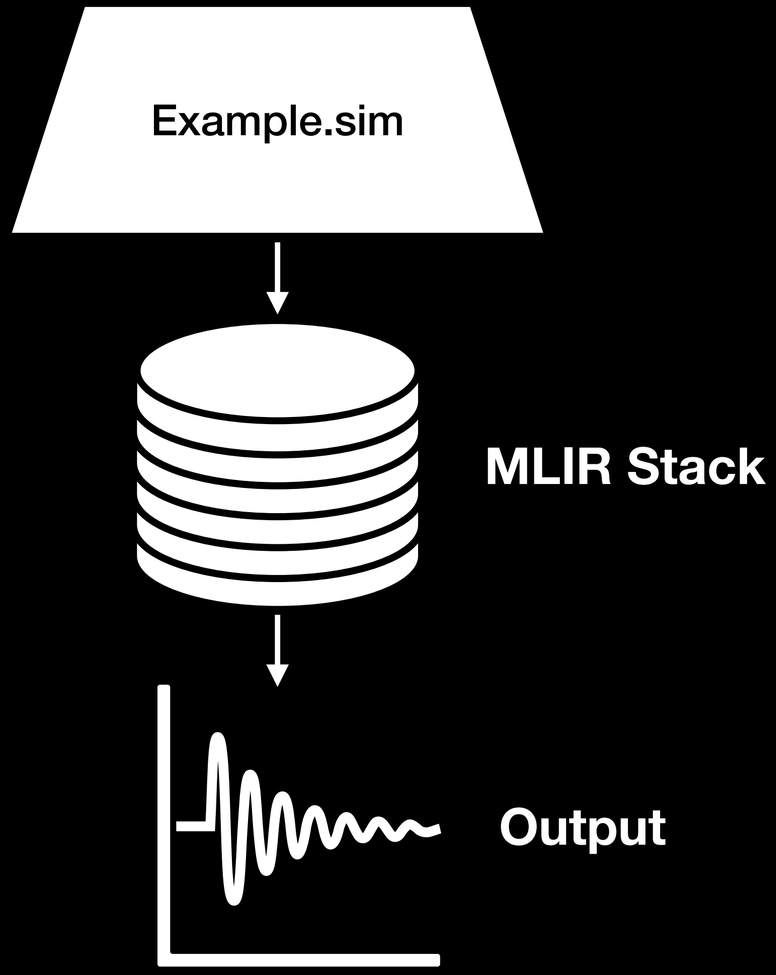
```
%cst_36 = constant 0.000000e+00 : f64
affine.for %arg0 = 0 to 3 {
  affine.for %arg1 = 0 to 3 {
    affine.store %cst_36, %0[%arg0, %arg1] : memref<3x3xf64>
    affine.for %arg2 = 0 to 3 {
      affine.for %arg3 = 0 to 3 {
        %3 = affine.load %1[%arg2, %arg3] : memref<3x3xf64>
        %4 = addi %arg2, %arg0 : index
        %5 = addi %arg3, %arg1 : index
        %6 = load %2[%4, %5] : memref<5x5xf64>
        %7 = mulf %3, %6 : f64
        %8 = affine.load %0[%arg0, %arg1] : memref<3x3xf64>
        %9 = addf %7, %8 : f64
        affine.store %9, %0[%arg0, %arg1] : memref<3x3xf64>
      }
    }
  }
}
```

```
def main() {
  var a = [[3,3,2,1,0],[0,0,1,3,1],[3,1,2,2,3],[2,0,0,2,2],[2,0,0,0,1]];
  var b = [[0,1,2],[2,2,0],[0,1,2]];
  var c = conv(a,b);
  print(c);
}
```

```
module {
  func @main() {
    %0 = sim.constant dense<[[3.000000e+00, 3.000000e+00, 2.000000e+00, 1.000000e+00, 0.000000e+00], [0.000000e+00, 0.000000e+00, 1.000000e+00,
3.000000e+00, 1.000000e+00], [3.000000e+00, 1.000000e+00, 2.000000e+00, 2.000000e+00, 3.000000e+00], [2.000000e+00, 0.000000e+00, 0.000000e+00,
2.000000e+00, 2.000000e+00], [2.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00, 1.000000e+00]]> : tensor<5x5xf64>
    %1 = sim.constant dense<[[0.000000e+00, 1.000000e+00, 2.000000e+00], [2.000000e+00, 2.000000e+00, 0.000000e+00], [0.000000e+00, 1.000000e+00
, 2.000000e+00]]> : tensor<3x3xf64>
    %2 = "sim.conv"(%0, %1) : (tensor<5x5xf64>, tensor<3x3xf64>) -> tensor<3x3xf64>
    sim.print %2 : tensor<3x3xf64>
    sim.return
  }
```

Goal.

To develop a compiler targeted at the CPU for the simple language.



Thank You!