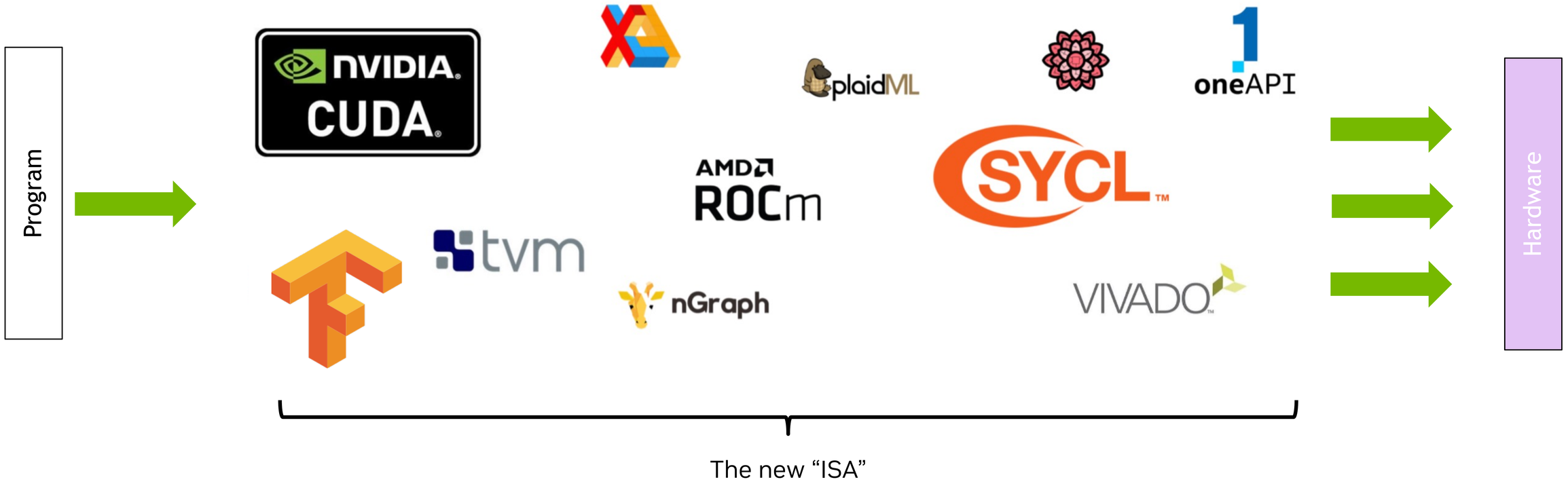
Embedding Domain-Specific Languages in C or C++ with Polygeist

Lorenzo Chelini (NVIDIA) and William S. Moses (University of Illinois Urbana-Champaign - UIUC)

Domain-specific Languages (DSLs): The new ISA

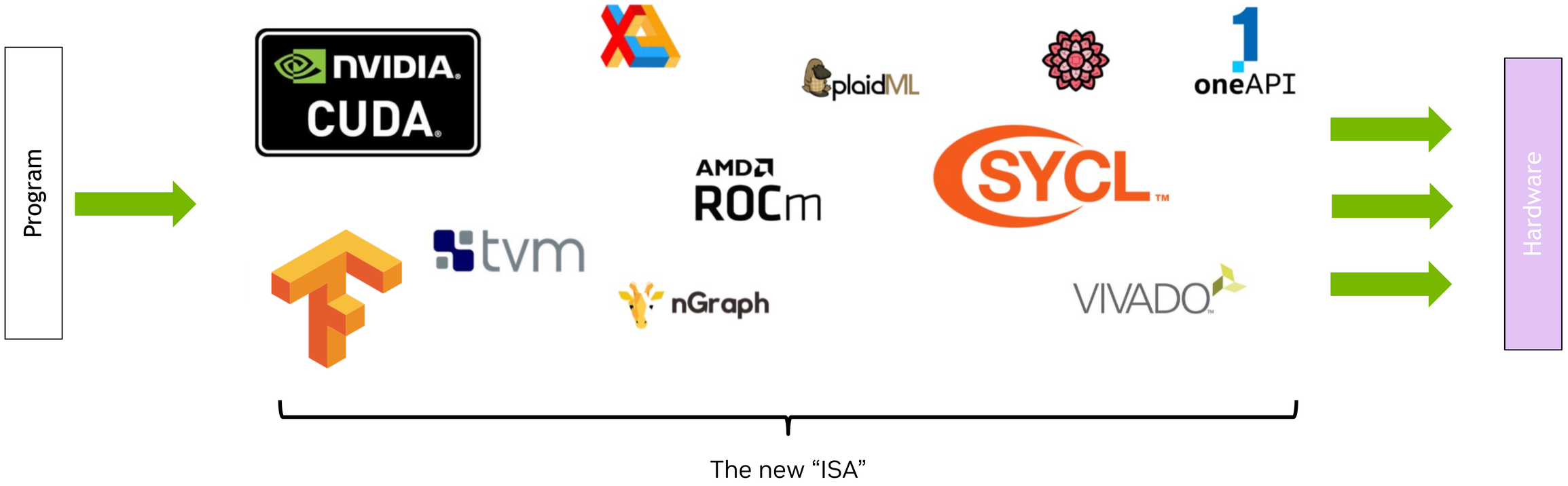


High  performance thanks to a richer semantics….

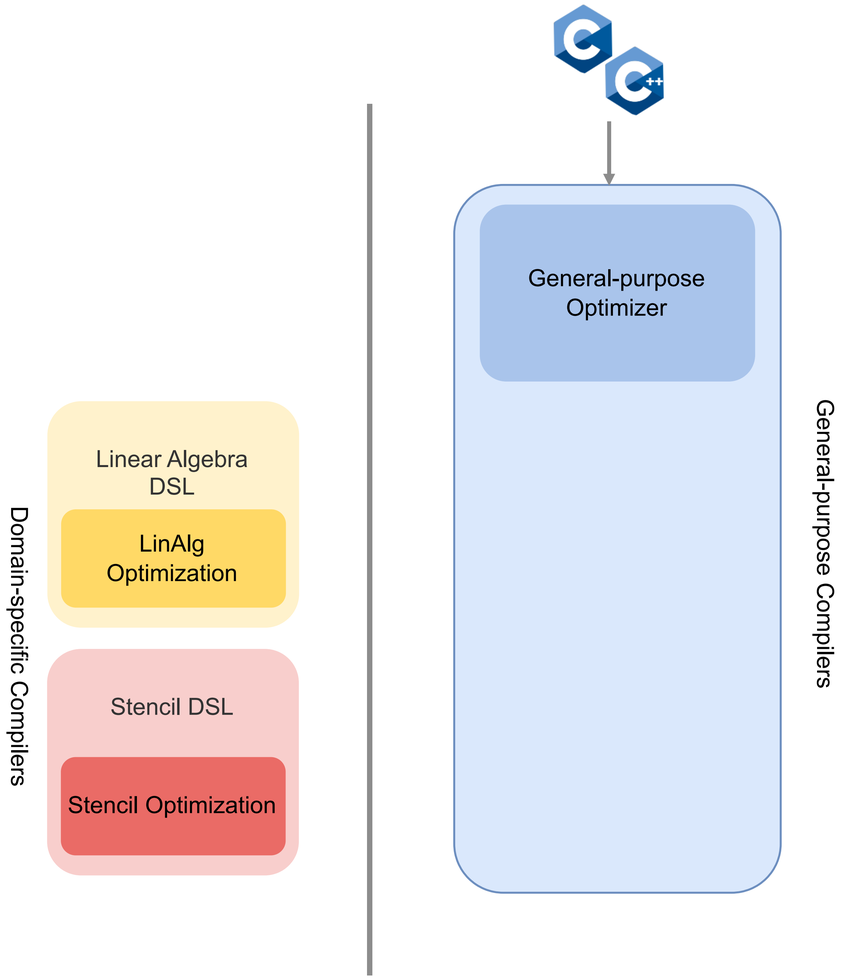
The Golden Age of Compiler Design in an Era of HW/SW Co-design by Dr. Chris Lattner

Domain-specific Languages (DSLs): The new ISA

But not compatible and hard to integrate into large codebase…

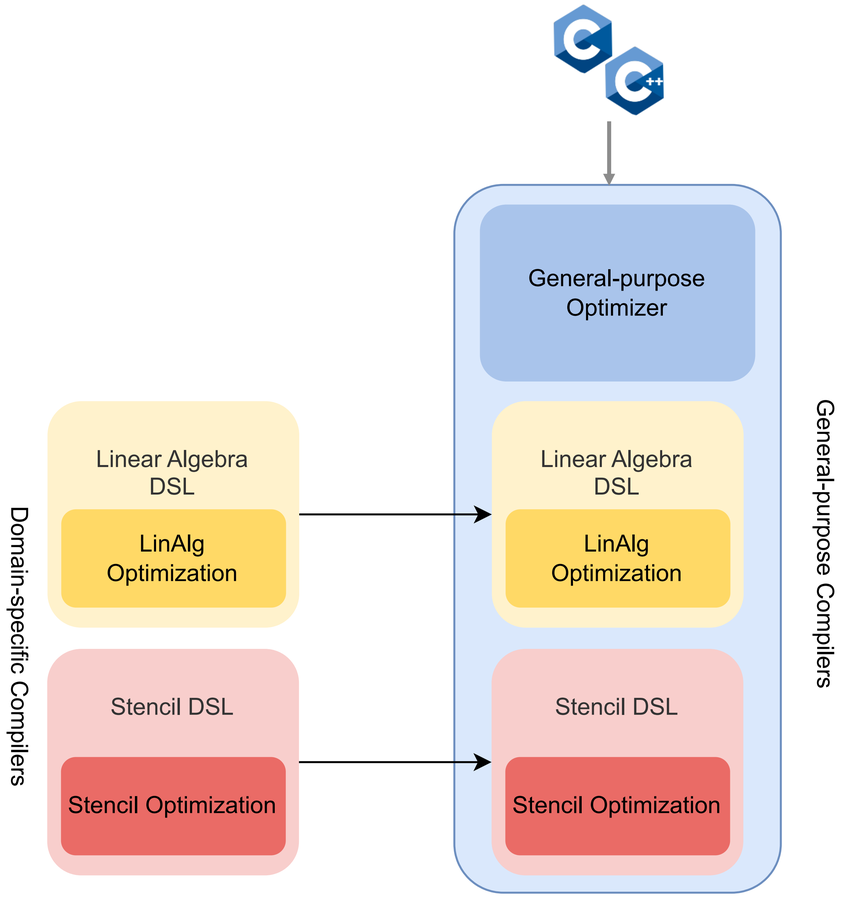


Problem statement



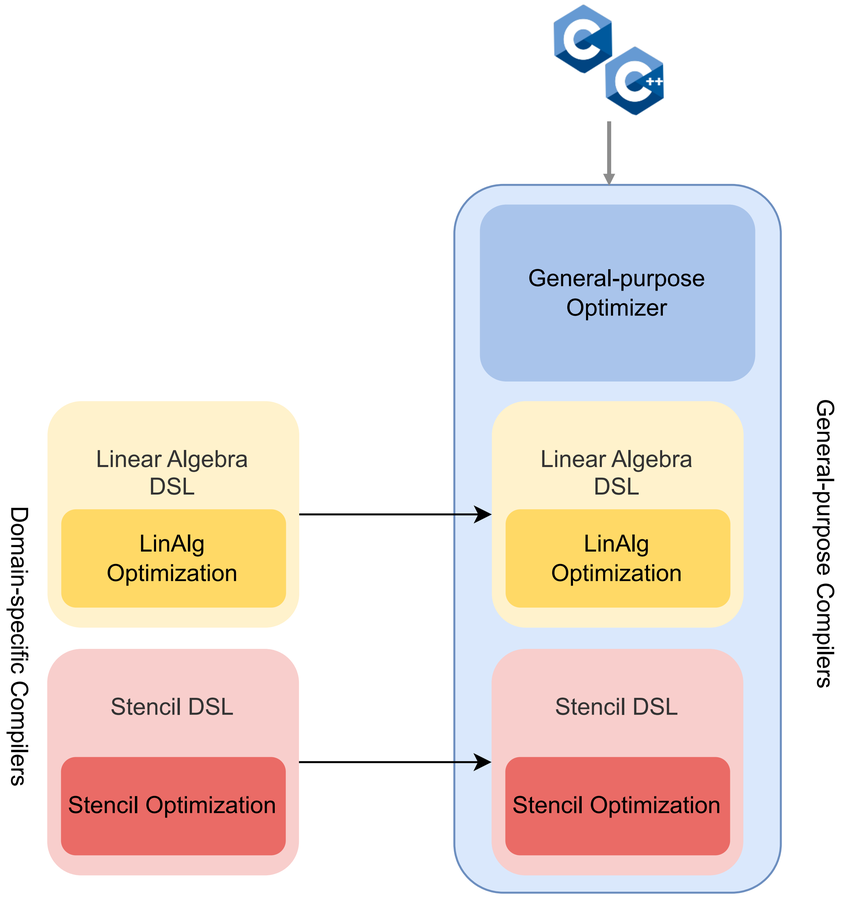
Design a mechanism *to bring* DSLs to existing C or C++ projects

Problem statement



Design a mechanism *to bring* DSLs to existing C or C++ projects

Problem statement



Syntax Plugin

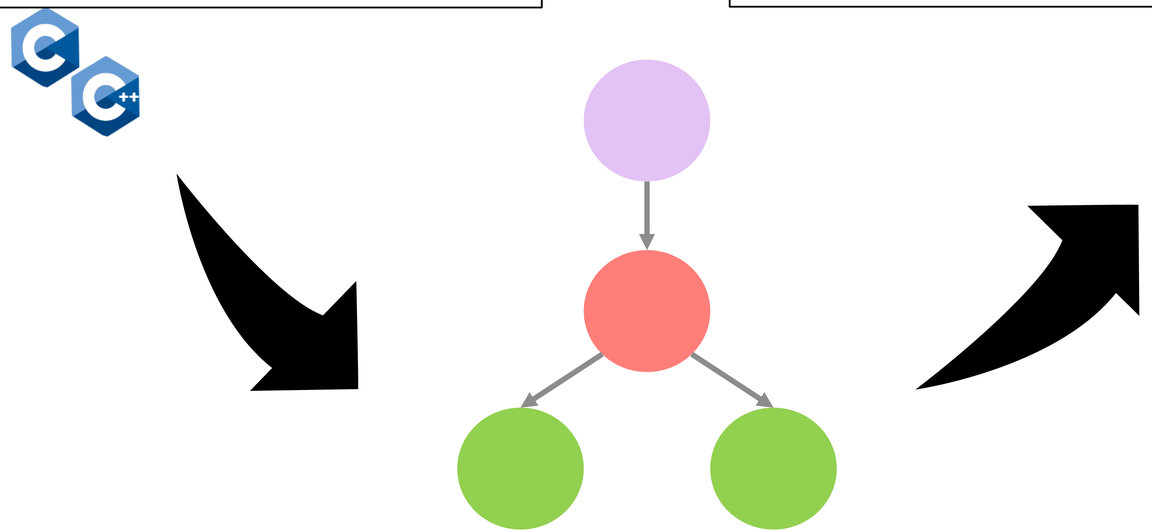
Polygeist

Design a mechanism *to bring* DSLs to existing C or C++ projects

Polygesit Frontend

```
void transpose( int X[Rows][Cols],
                int Y[Cols][Rows] ) {
  for(int i = 0; i < Rows; i++) {
    for (int j = 0; j < Cols; j++)
      Y[j][i] = X[i][j];
  }
}
```

```
func.func @transpose (.. ..) {
  scf.for %i = %c0 to %c_rows step %c1 {
    scf.for %j = %c0 to %c_cols step %c1 {
      %0 = memref.load %X[%i, %j] : memref<?x?xi32>
      memref.store %0, %Y[%j, %i] : memref<?x?xi32>
    }
  }
}
```

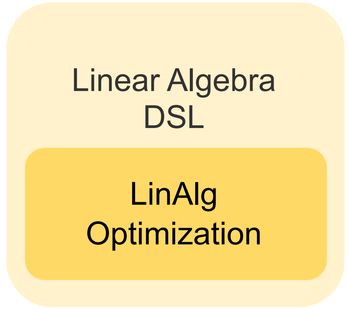


Clang AST

Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT ‘21

How do we bring domain-specific knowledge in General-Purpose compilers?

We need three ingredients...



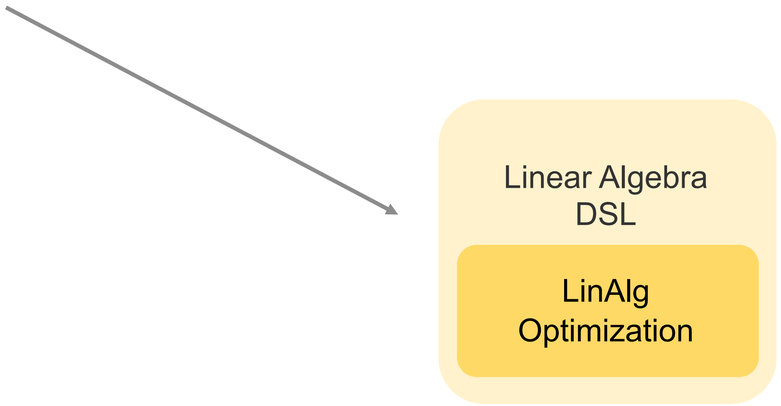
How do we bring domain-specific knowledge in General-Purpose compilers?

We need three ingredients...

Syntax

1

```
// transpose

Y(i,j) = X(j,i)
```



How do we bring domain-specific knowledge in General-Purpose compilers?

We need three ingredients...

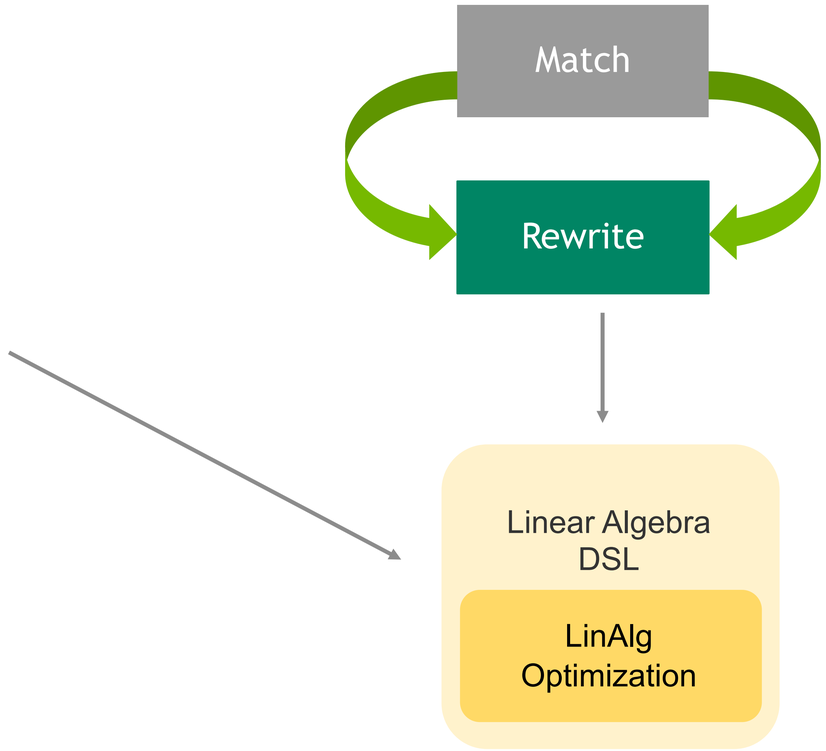
Syntax

1

Rewriting rules

2

```
// transpose

Y(i,j) = X(j,i)
```



How do we bring domain-specific knowledge in General-Purpose compilers?

We need three ingredients...

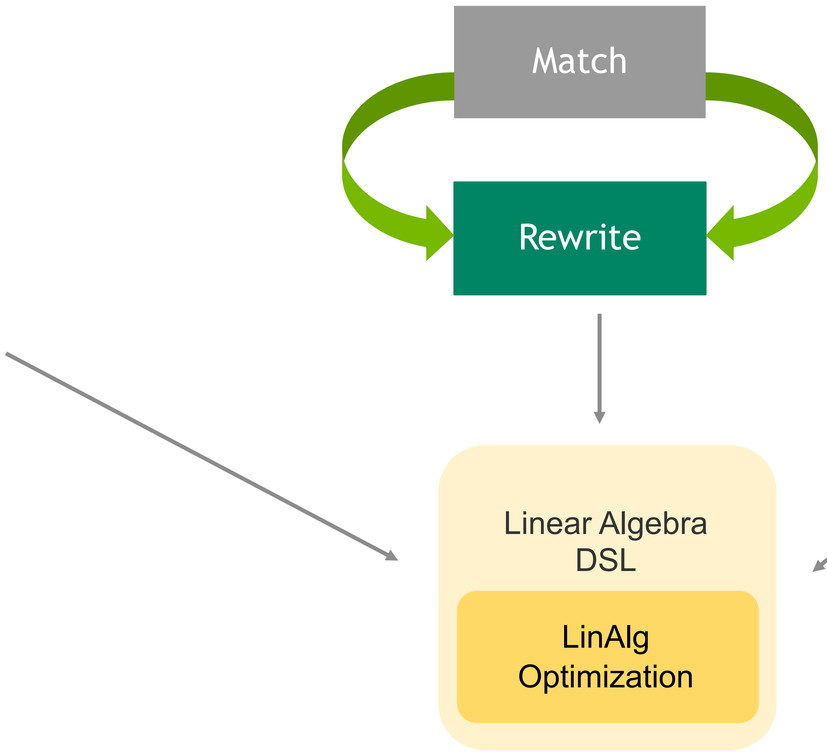
Syntax

1

```
// transpose

Y(i,j) = X(j,i)
```

Rewriting rules

2

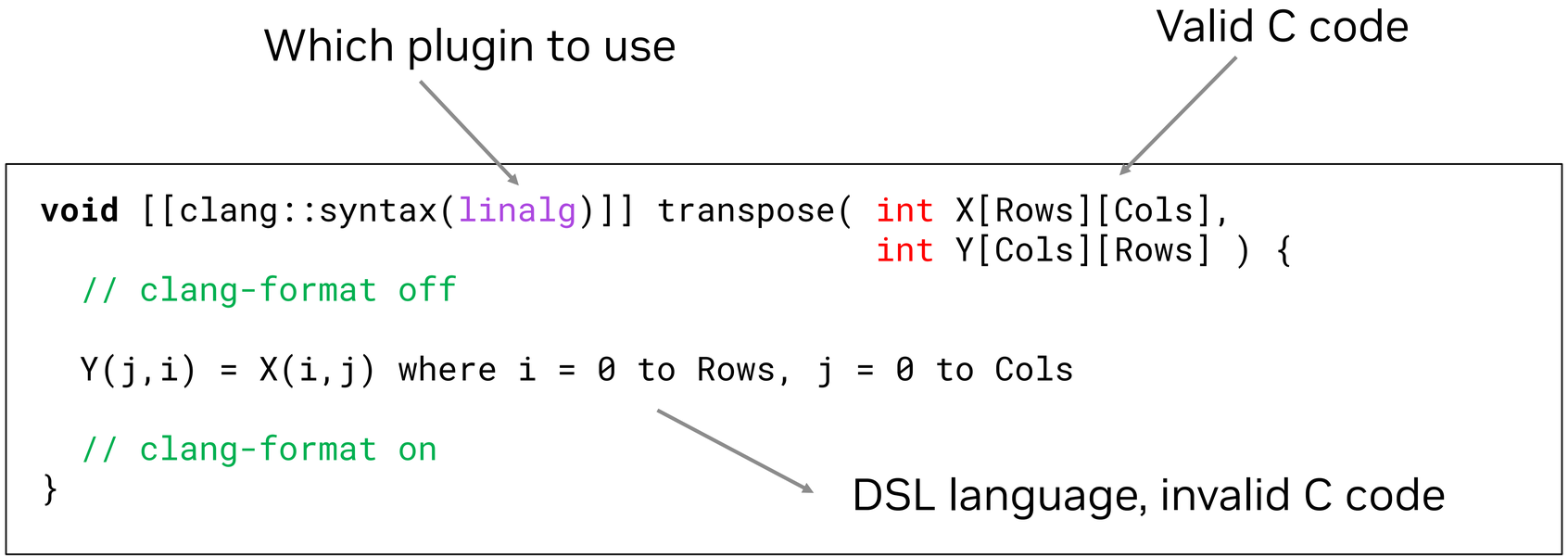


operations

3

```
__attribute__(custom_op(linalg.transpose))
void transpose ( int Y[Rows][Cols],
                 int X[Cols][Rows] ) {

  Y(i,j) = X(j,i)

}
```

1 How do we bring DSL syntax?

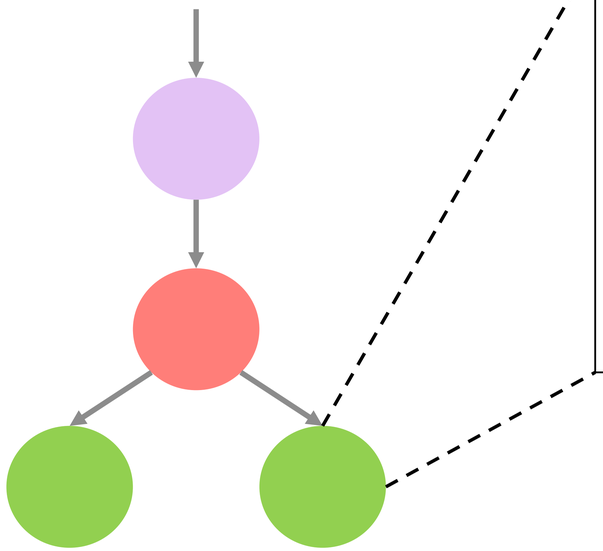


-DENABLE_LINALG_PLUGIN=1 cgeist –S –function transpose transpose.c

Really Embedding Domain-specific Languages into C++; Finkel, et al.
Tensor Comprehensions: Framework-Agnostic High-performance Machine Learning Abstraction; Vasilache, et al.
https://github.com/andidr/teckyl

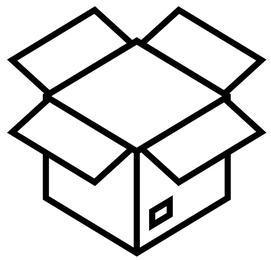
How does it work in practice?

Parsing phase



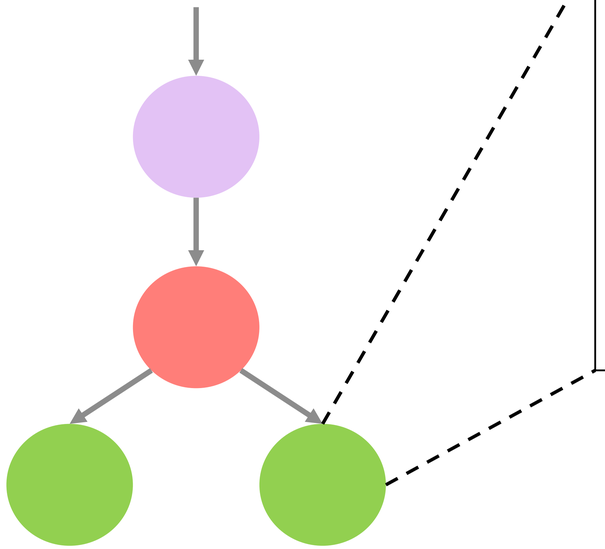
```
void [[clang::syntax(linalg)]] transpose( int X[Rows][Cols],
                                          int Y[Cols][Rows] ) {
  // clang-format off

  Y(j,i) = X(i,j) where i = 0 to Rows, j = 0 to Cols

  // clang-format on
}
```

Parser::ParseFunctionDefinition



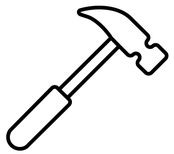
How does it work in practice?

IR building phase

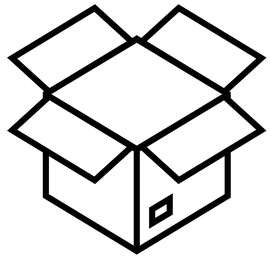


AST::FunctionDecl

```
func.func [[clang::syntax(linalg)]] transpose( %X = memref<RowsxCols>,
                                               %Y = memref<ColsxRows> ) {
  // clang-format off

  .. empty body ..

  // clang-format on
}
```



LinalgPlugin.so



Transpose example

```
(def
  (ident transpose)
  (list
    (param
      (ident X)
      (tensor_type
        (int32)
        (list (ident M) (ident N)))))
  (list
    (param
      (ident Y)
      (tensor_type
        (int32)
        (list (ident N) (ident M)))))
  (list
    (comprehension
      (ident C)
      (list (ident j) (ident i))
      (=)
      (apply
        (ident In)
        (list (ident i) (ident j)))
      )))
```

Polygeist

```
func.func @transpose(%X: memref<?x?xi32>,
                     %Y: memref<?x?xi32>) {
  %arg0 = to_tensor %X : memref<?x?xi32>
  %arg1 = to_tensor %Y : memref<?x?xi32>

  %2 = linalg.generic {
    indexing_maps = [affine_map<(d0,d1)->(d1,d0),
                     affine_map<(d0,d1)->(d0,d1)]
    iterators_types = [parallel, parallel]
    ins( %arg0 : tensor<?x?xi32>)
    outs(%arg1 : tensor<?x?xi32>) {
    ^bb0(%in: i32, %out: i32):
      linalg.yield %in
    }
  }

  %3 = materialize_in_dst %2 in %arg1 : tensor<?x?xi32>

  return
}
```

Plugin

2 How do we bring optimizations?

```
namespace my_linalg_lib {
     void [[clang::syntax(linalg)]] transpose( int X[Rows][Cols],
                                               int Y[Cols][Rows] ) {
       // clang-format off

       Y(j,i) = X(i,j) where i = Rows, j = Cols









        // clang-format on

     } // end transpose

} // namespace my_linalg_lib
```

2 How do we bring optimizations?

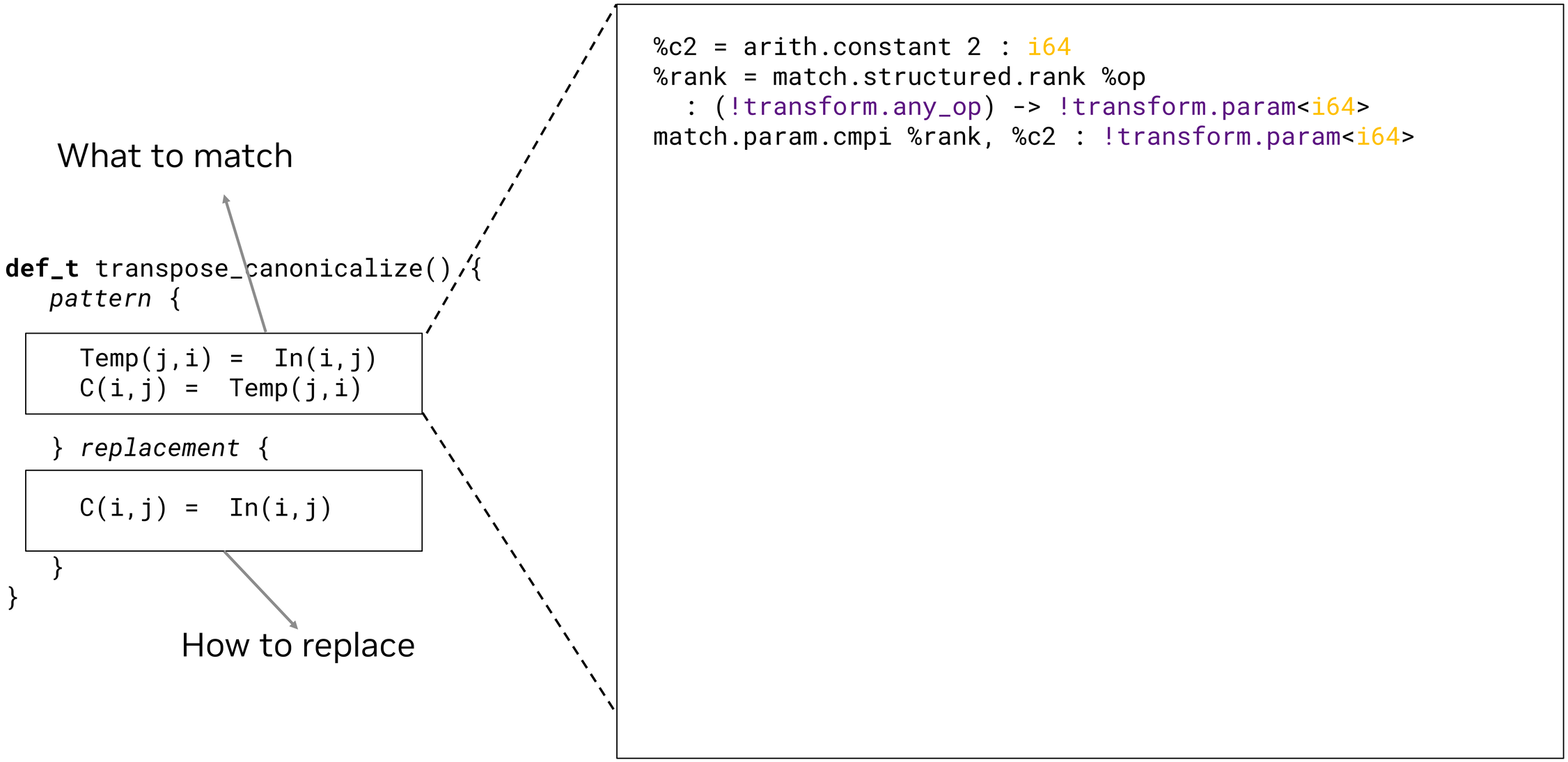
```
namespace my_linalg_lib {
     void [[clang::syntax(linalg)]] transpose( int X[Rows][Cols],
                                               int Y[Cols][Rows] ) {
       // clang-format off

       Y(j,i) = X(i,j) where i = Rows, j = Cols
```

```
def_t transpose_canonicalize() {
  pattern {
    Temp(j,i) =  In(i,j)
    C(i,j) =  Temp(j,i)
  } replacement {
    C(i,j) =  In(i,j)
  }
 }
```

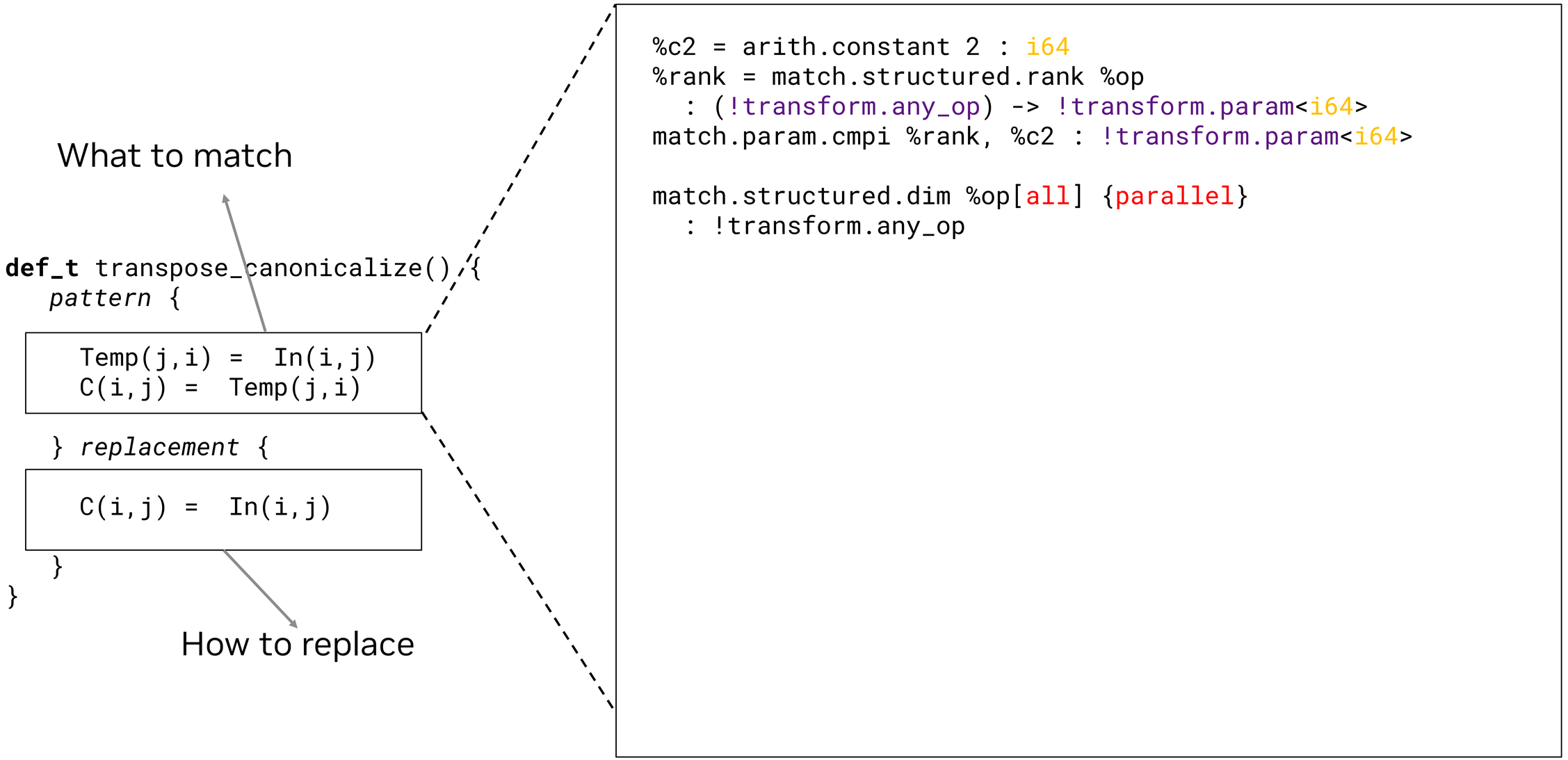
```
       // clang-format on

     } // end transpose

} // namespace my_linalg_lib
```

C = transpose(transpose(A)) ➔ C = A

How do we bring rewriting rules?



How do we bring rewriting rules?



How do we bring rewriting rules?

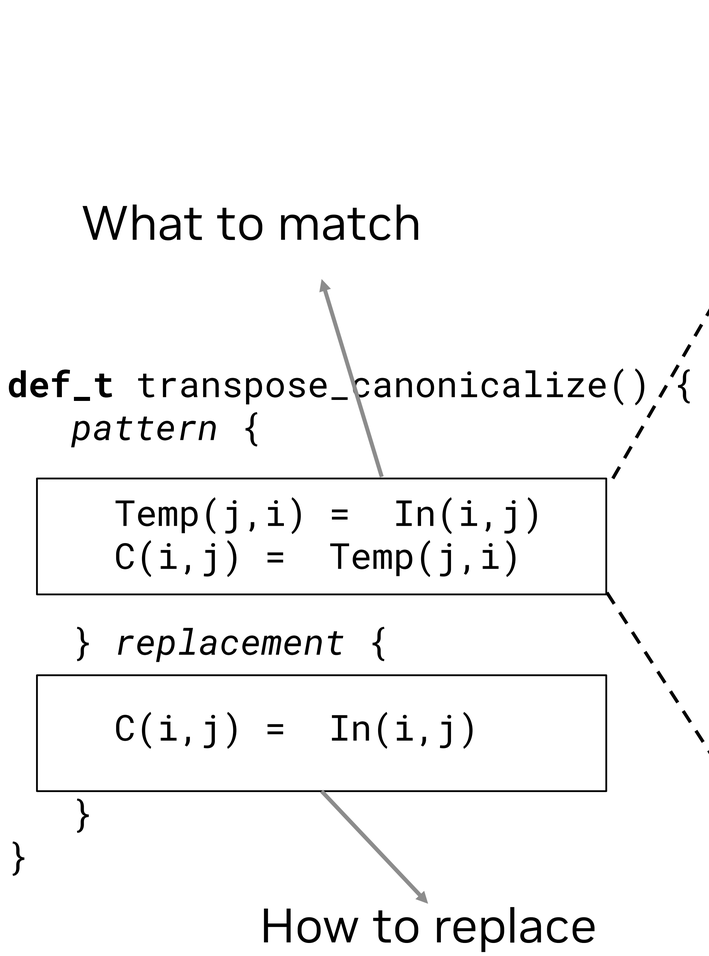
What to match

```
def_t transpose_canonicalize() {
   pattern {
      Temp(j,i) =  In(i,j)
      C(i,j) =  Temp(j,i)
   } replacement {
      C(i,j) =  In(i,j)
   }
}
```

How to replace

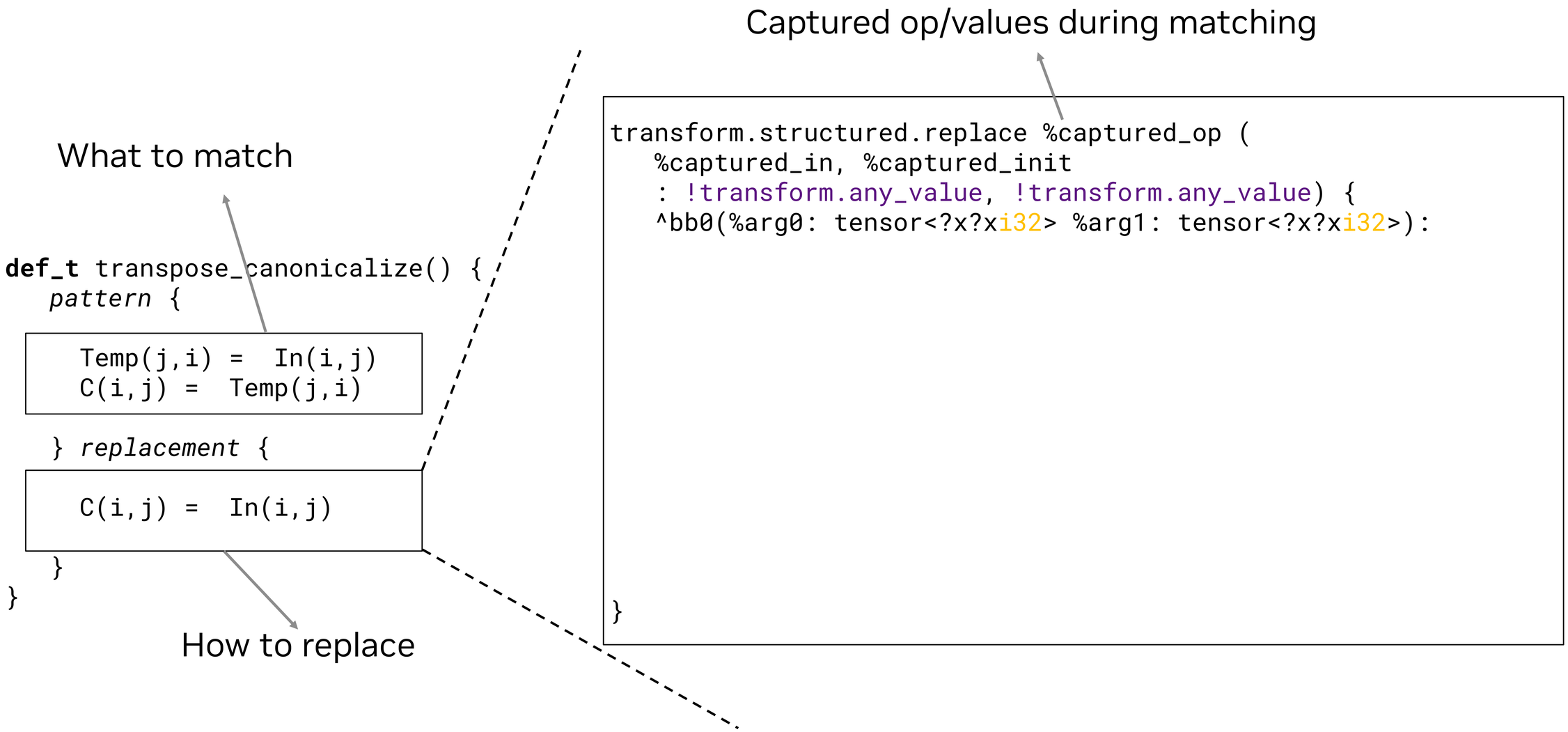
```
%c2 = arith.constant 2 : i64
%rank = match.structured.rank %op
  : (!transform.any_op) -> !transform.param<i64>
match.param.cmpi %rank, %c2 : !transform.param<i64>

match.structured.dim %op[all] {parallel}
  : !transform.any_op

%c1 = arith.constant 1 : i64
%num_inputs = match.structured.num_inputs %op
  : (!transform.any_op) -> !transform.param<i64>
%num_inits = match.structured.num_inits %op
  : (!transform.any_op) -> !transform.param<i64>
match.param.cmpi %num_inputs, %c1 : !transform.param<i64>
match.param.cmpi %num_inits, %c1 : !transform.param<i64>
```

How do we bring rewriting rules?

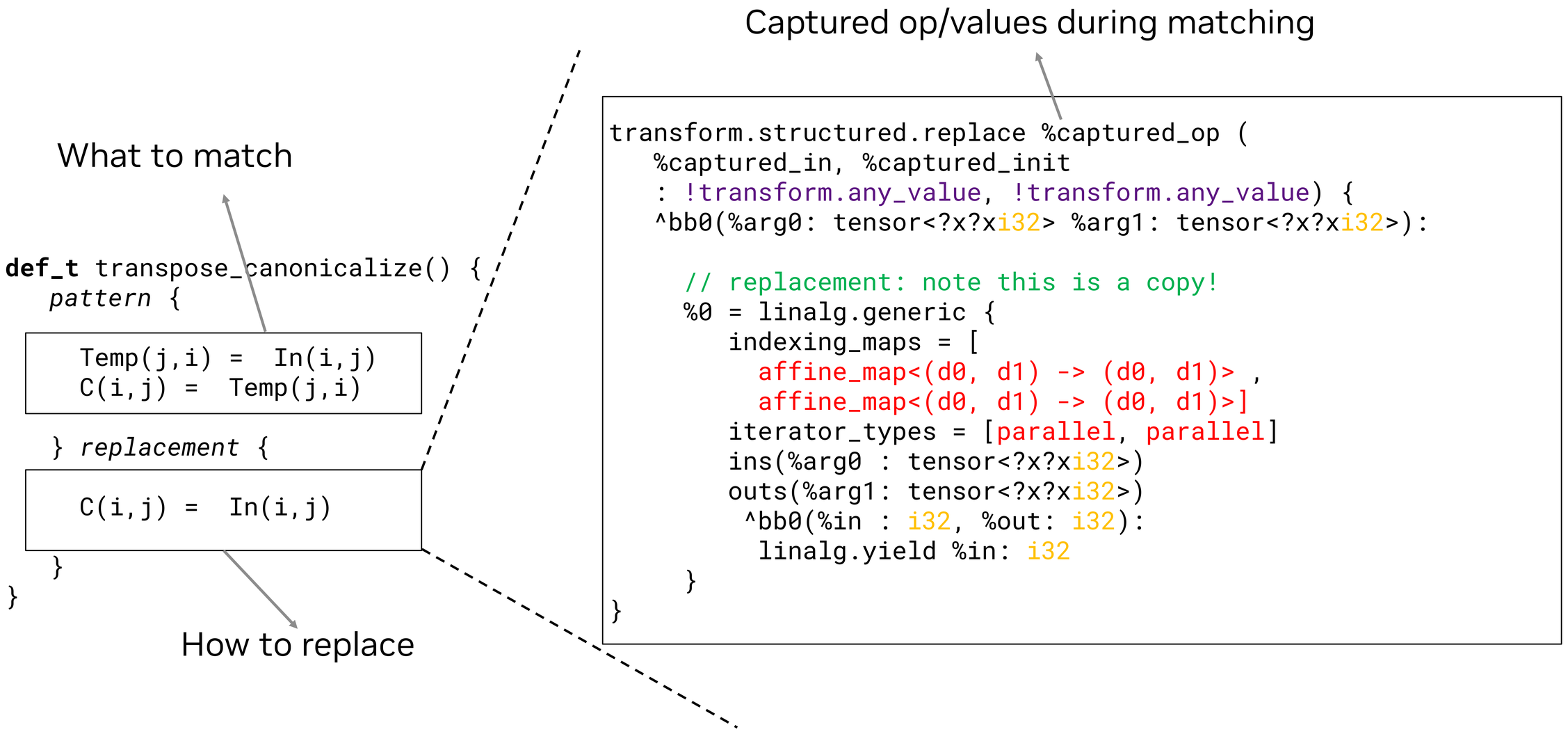



```
%c2 = arith.constant 2 : i64
%rank = match.structured.rank %op
  : (!transform.any_op) -> !transform.param<i64>
match.param.cmpi %rank, %c2 : !transform.param<i64>

match.structured.dim %op[all] {parallel}
  : !transform.any_op

%c1 = arith.constant 1 : i64
%num_inputs = match.structured.num_inputs %op
  : (!transform.any_op) -> !transform.param<i64>
%num_inits = match.structured.num_inits %op
  : (!transform.any_op) -> !transform.param<i64>
match.param.cmpi %num_inputs, %c1 : !transform.param<i64>
match.param.cmpi %num_inits, %c1 : !transform.param<i64>

match.structured.input %op[0] {identity}
  : !transform.any_op
match.structured.init %op[0] {projected_permutation}
  : !transform.any_op

match.structured.body {passthrough}
  : !transform.any_op
```

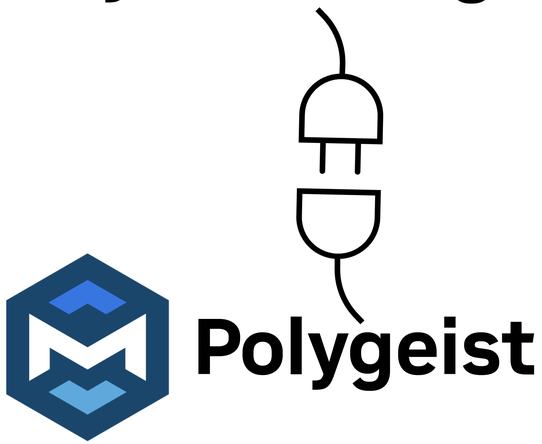
How do we bring rewriting rules?

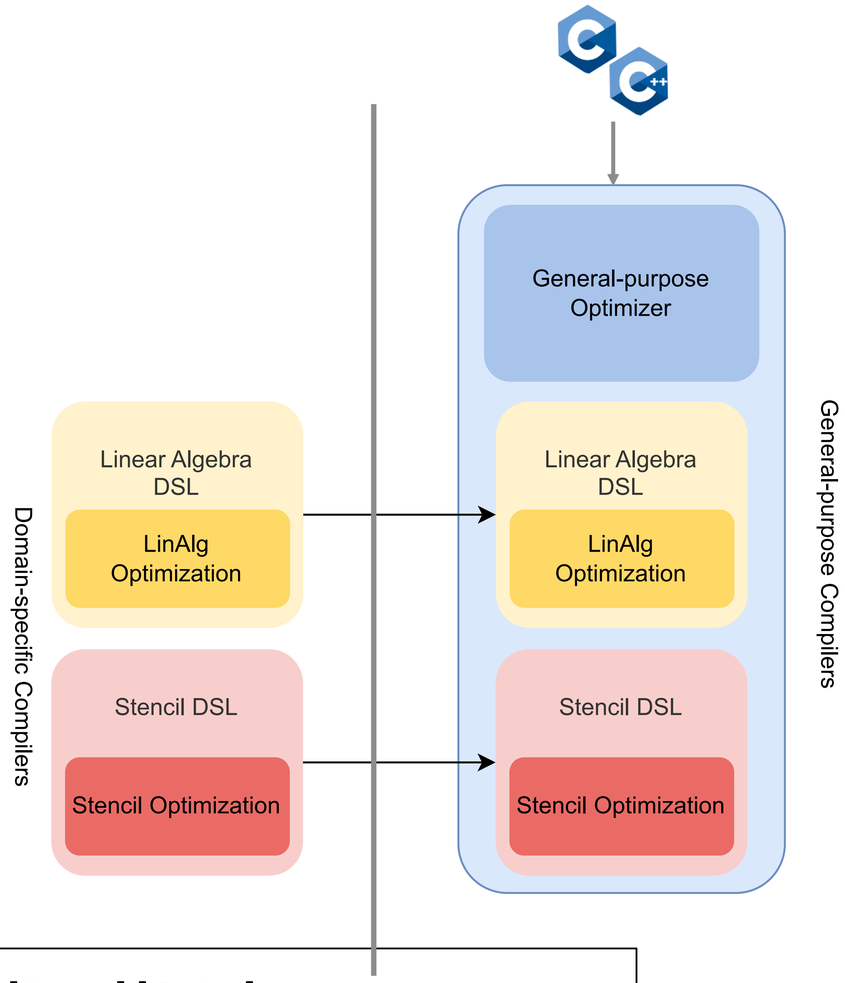


How do we bring rewriting rules?



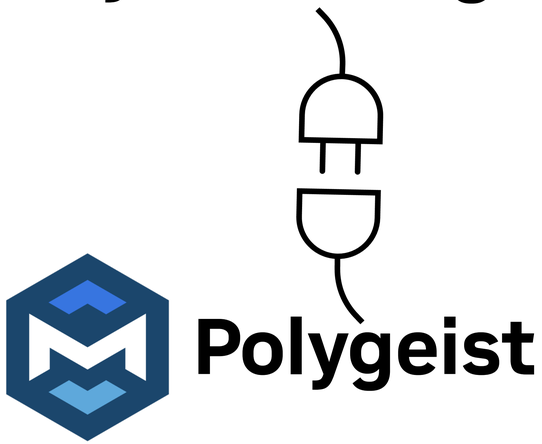
Syntax Plugin



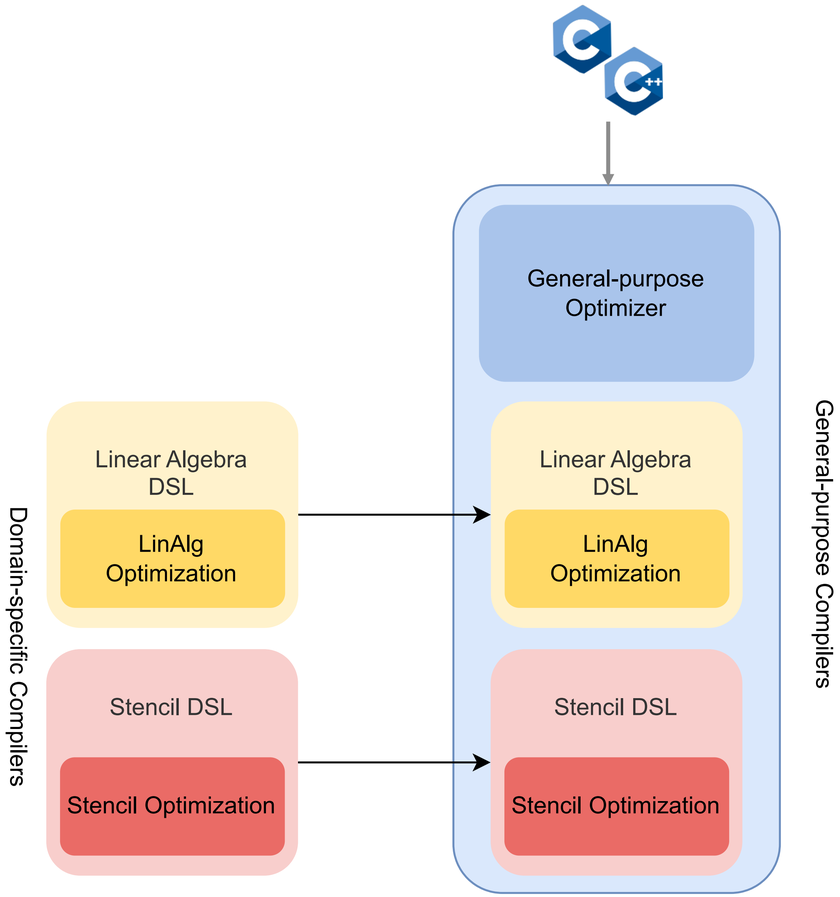



```
void [[clang::syntax(linalg)]] transpose( int X[Rows][Cols],
                                          int Y[Cols][Rows] ) {
  // clang-format off

  Y(j,i) = X(i,j) where i = 0 to Rows, j = 0 to Cols

  // clang-format on
}
```

Syntax Plugin



Polygeist



Get involved !!

l.chelini@icloud.com wsmoses@illinois.edu

https://github.com/llvm/Polygeist