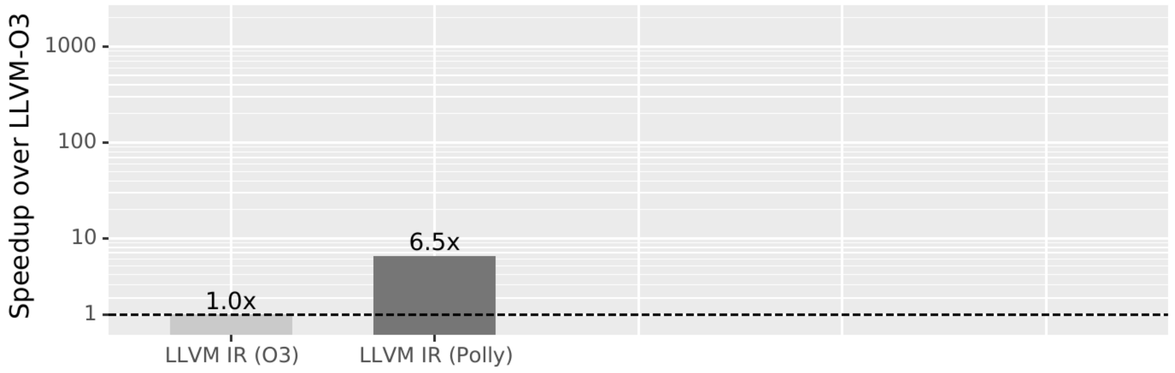
mlirSynth

Synthesizing Domain-Specific Programs in MLIR

Alexander Brauckmann

Elizabeth Polgreen
Tobias Grosser
Michael O'Boyle

Raising to High-Level IRs



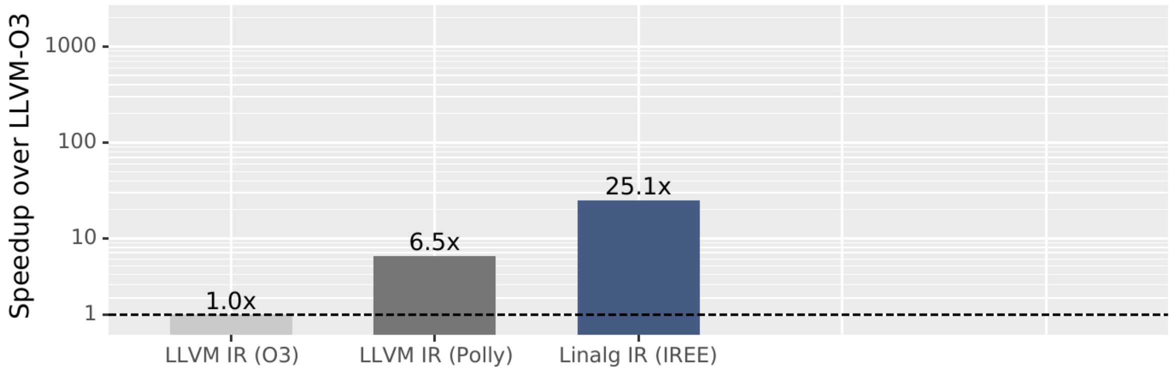
C Program

```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

CPU: AMD Ryzen 9 3900X

Motivation

Raising to High-Level IRs



C Program

```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

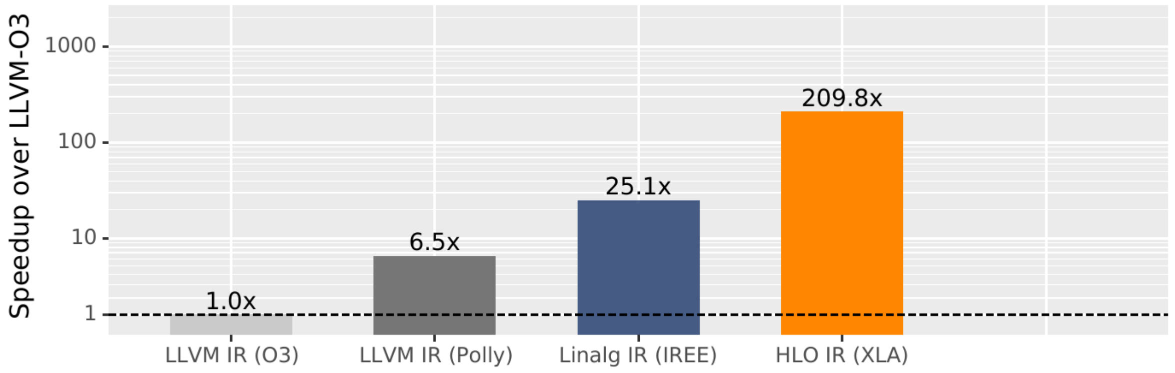
Linalg IR

```
%0 = tensor.collapse_shape %arg0 [[0, 1], [2]]
  : tensor<150x140x160xf64>
  into tensor<2100x160xf64>
%1 = linalg.matmul
  ins(%0, %arg1 : tensor<21000x160xf32>,
                  tensor<160x160xf32>)
  outs(%1 : tensor<21000x160xf32>)
  -> tensor<21000x160xf32>
%2 = tensor.expand_shape %1 [[0, 1], [2]]
  : tensor<2100x160xf64>
  into tensor<150x140x160xf64>
```

CPU: AMD Ryzen 9 3900X

Motivation

Raising to High-Level IRs



HLO IR

```
%0 = mhlo.dot_general (%arg0, %arg1) {
  dot_dimension_numbers = #mhlo.dot<
    lhs_contracting_dimensions = [2],
    rhs_contracting_dimensions = [0]>}
  : (tensor<150x140x160xf32>,
    tensor<160x160xf32>)
  -> tensor<150x140x160xf32>
```

Linalg IR

```
%0 = tensor.collapse_shape %arg0 [[0, 1], [2]]
  : tensor<150x140x160xf64>
  into tensor<2100x160xf64>
%1 = linalg.matmul
  ins(%0, %arg1 : tensor<21000x160xf32>,
                  tensor<160x160xf32>)
  outs(%1 : tensor<21000x160xf32>)
  -> tensor<21000x160xf32>
%2 = tensor.expand_shape %1 [[0, 1], [2]]
  : tensor<2100x160xf64>
  into tensor<150x140x160xf64>
```

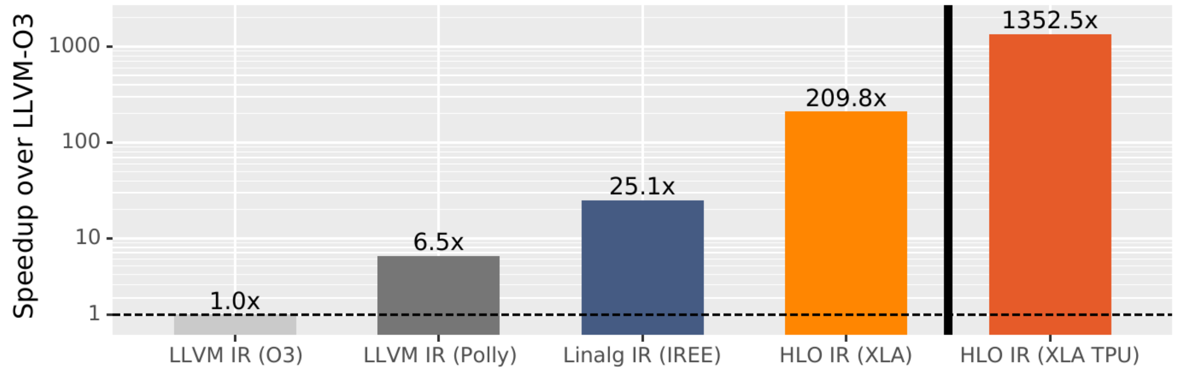
C Program

```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

CPU: AMD Ryzen 9 3900X

Motivation

Raising to High-Level IRs



HLO IR

```
%0 = mhlo.dot_general (%arg0, %arg1) {
  dot_dimension_numbers = #mhlo.dot<
    lhs_contracting_dimensions = [2],
    rhs_contracting_dimensions = [0]>}
  : (tensor<150x140x160xf32>,
    tensor<160x160xf32>)
  -> tensor<150x140x160xf32>
```

Linalg IR

```
%0 = tensor.collapse_shape %arg0 [[0, 1], [2]]
  : tensor<150x140x160xf64>
  into tensor<2100x160xf64>
%1 = linalg.matmul
  ins(%0, %arg1 : tensor<21000x160xf32>,
                   tensor<160x160xf32>)
  outs(%1 : tensor<21000x160xf32>)
  -> tensor<21000x160xf32>
%2 = tensor.expand_shape %1 [[0, 1], [2]]
  : tensor<2100x160xf64>
  into tensor<150x140x160xf64>
```

C Program

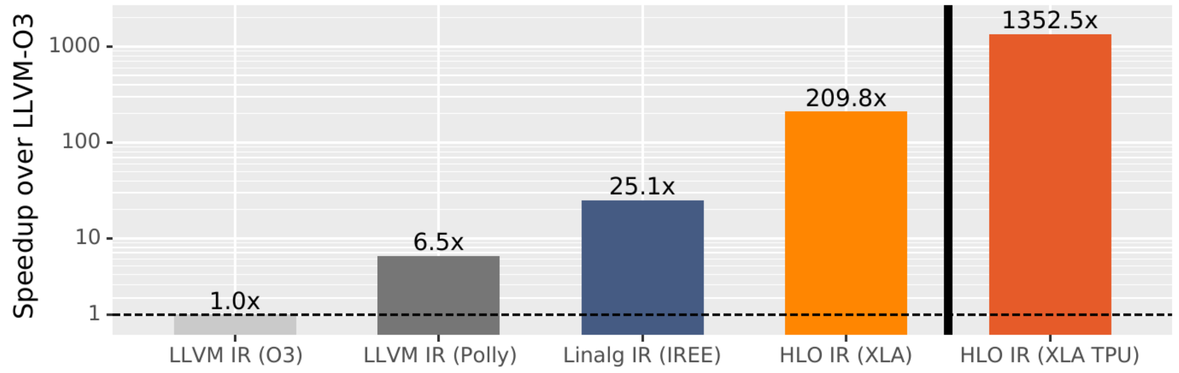
```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

CPU: AMD Ryzen 9 3900X

TPU: TPUv2

Motivation

Raising to High-Level IRs



HLO IR

```
%0 = mhlo.dot_general (%arg0, %arg1) {
  dot_dimension_numbers = #mhlo.dot<
    lhs_contracting_dimensions = [2],
    rhs_contracting_dimensions = [0]>}
  : (tensor<150x140x160xf32>,
    tensor<160x160xf32>)
  -> tensor<150x140x160xf32>
```

Linalg IR

```
%0 = tensor.collapse_shape %arg0 [[0, 1], [2]]
  : tensor<150x140x160xf64>
  into tensor<2100x160xf64>
%1 = linalg.matmul
  ins(%0, %arg1 : tensor<21000x160xf32>,
                  tensor<160x160xf32>)
  outs(%1 : tensor<21000x160xf32>)
  -> tensor<21000x160xf32>
%2 = tensor.expand_shape %1 [[0, 1], [2]]
  : tensor<2100x160xf64>
  into tensor<150x140x160xf64>
```

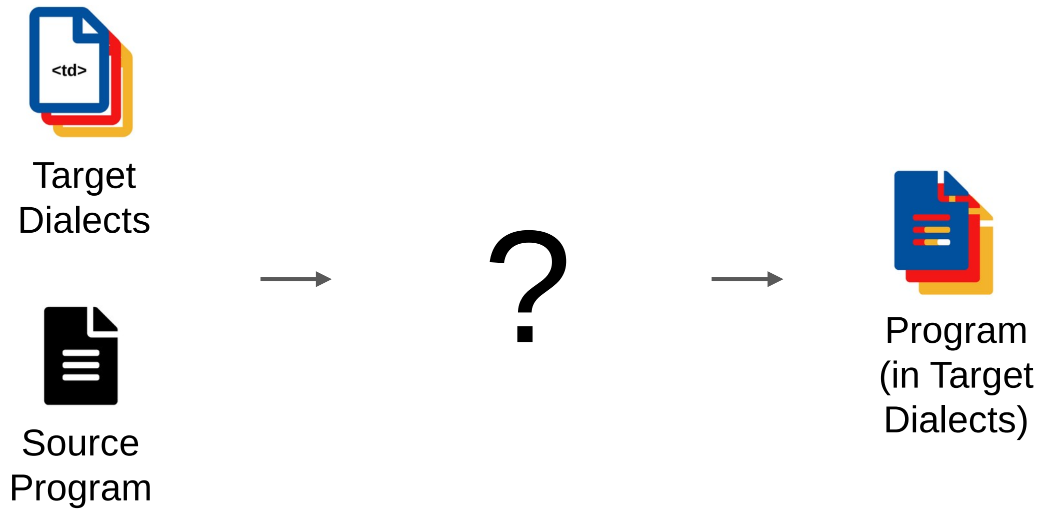
C Program

```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

Raising
Raising

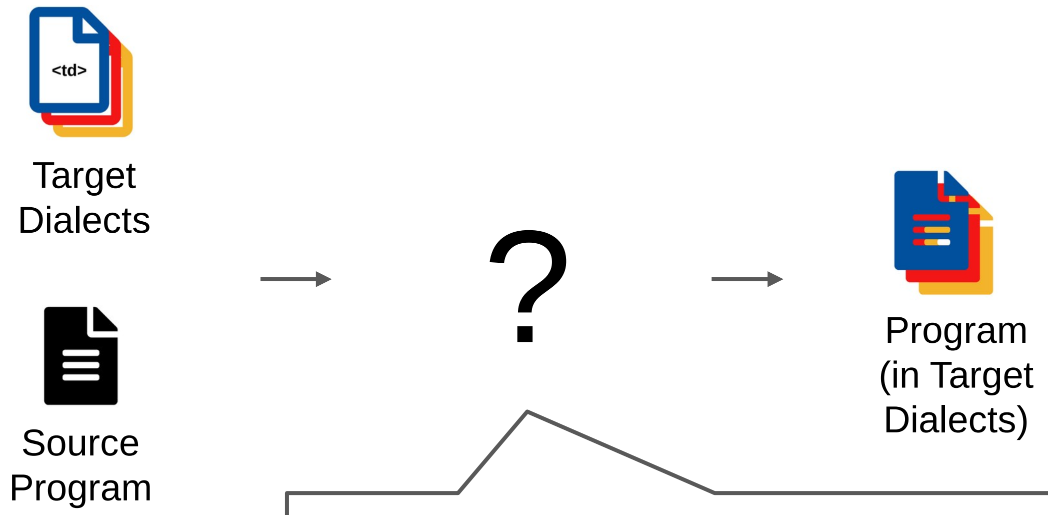
CPU: AMD Ryzen 9 3900X
TPU: TPUv2

Raising with Synthesis



Motivation

Raising with Synthesis



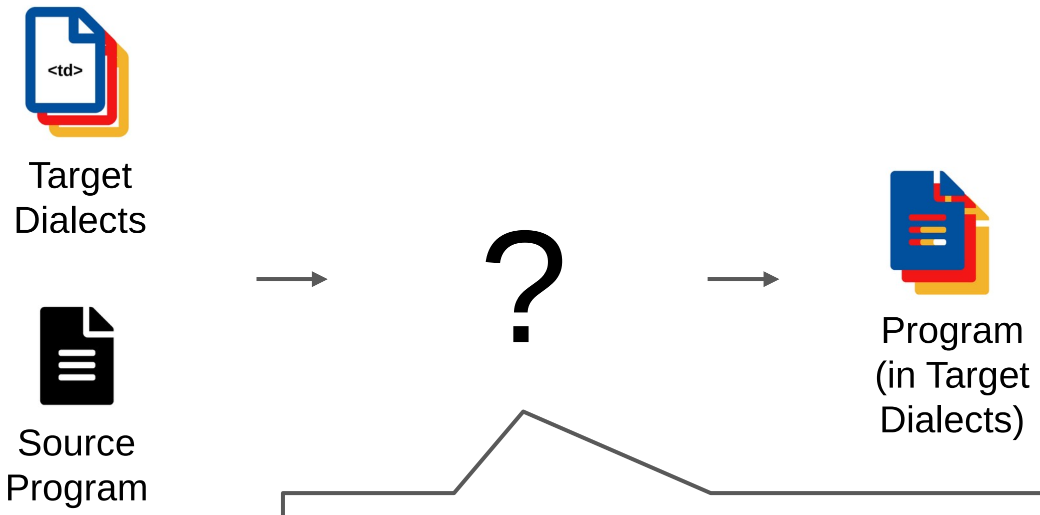
Fast

Robust

Automatic

Motivation

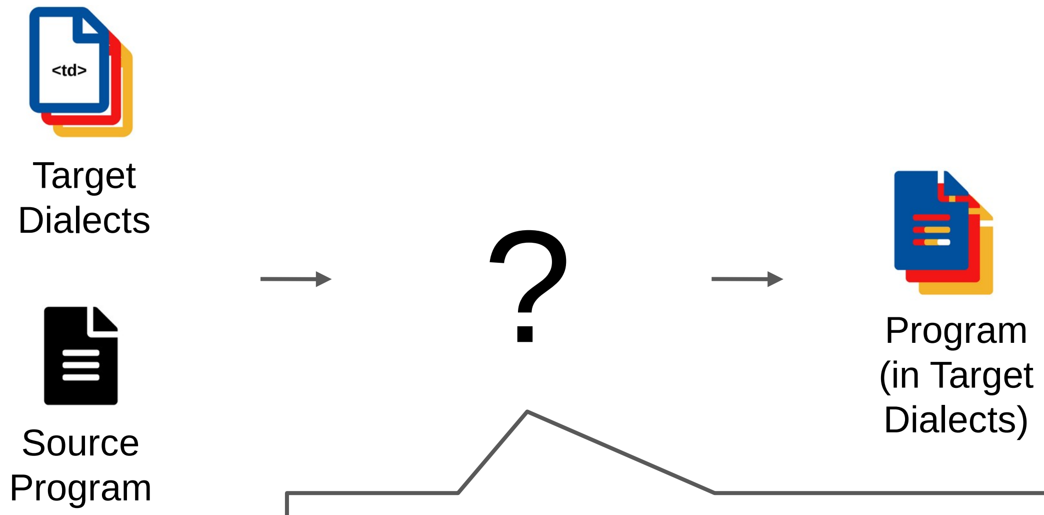
Raising with Synthesis



	Pattern Matching
Fast	✅
Robust	❌
Automatic	❌

Motivation

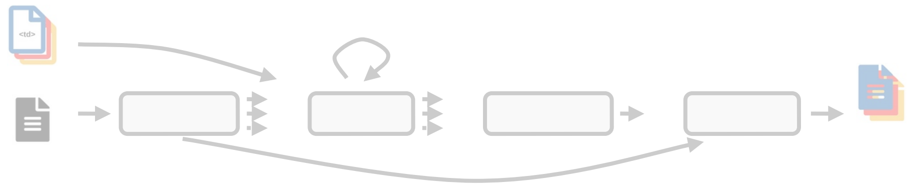
Raising with Synthesis



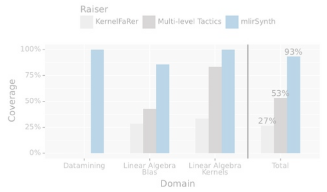
	Pattern Matching	<u>Synthesis</u>
Fast	✅	❌
Robust	❌	✅
Automatic	❌	✅

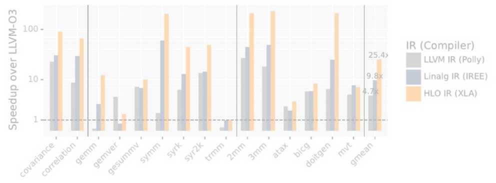
Outline

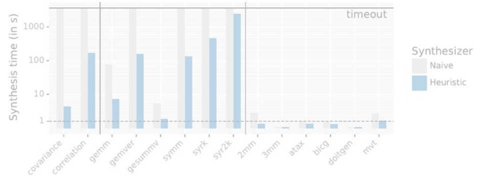
mlirSynth



Results



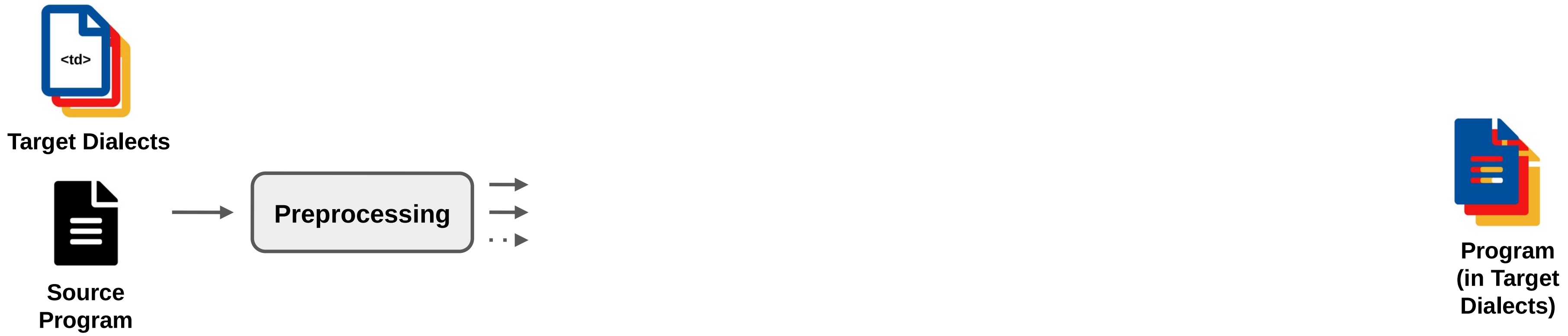




mlirSynth

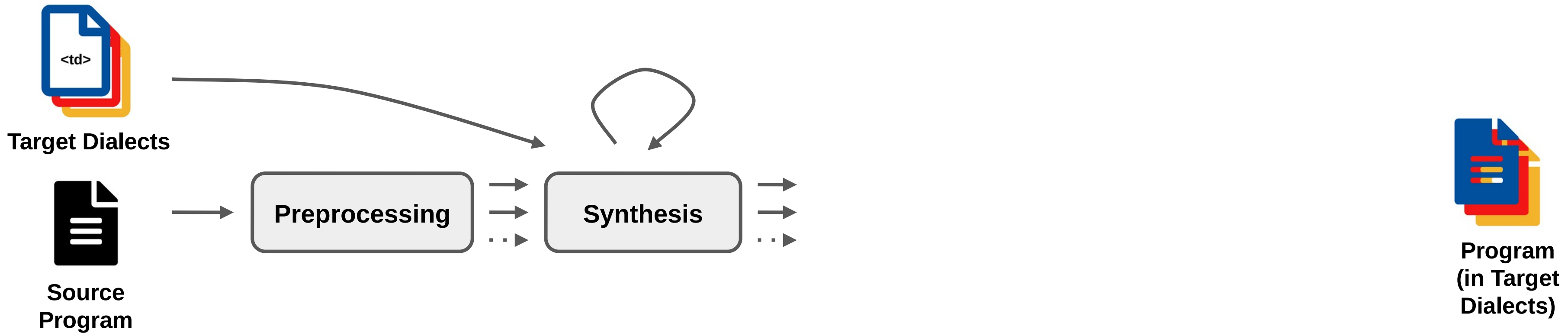
<td>
Target Dialects
Source Program
Program (in Target Dialects)

mlirSynth



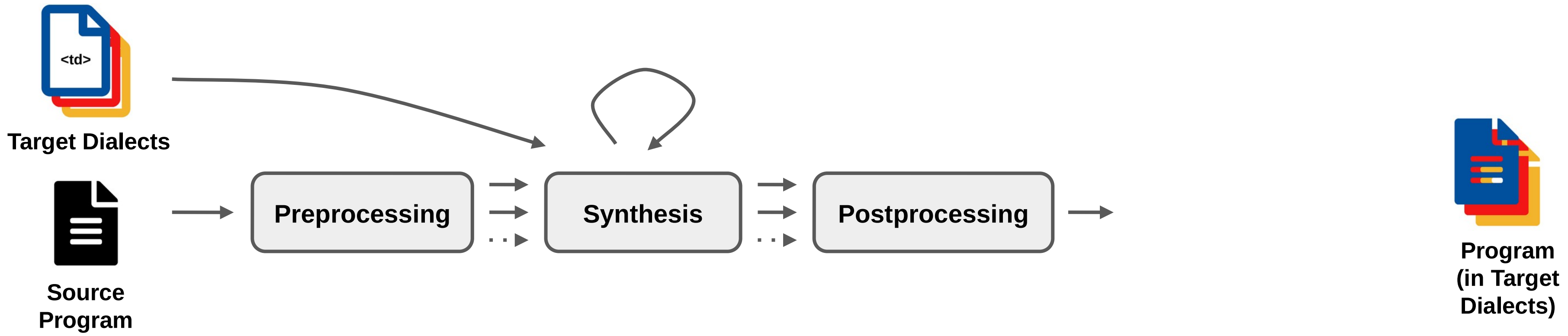
<td>
Target Dialects
Source Program
Preprocessing
Program (in Target Dialects)

mlirSynth



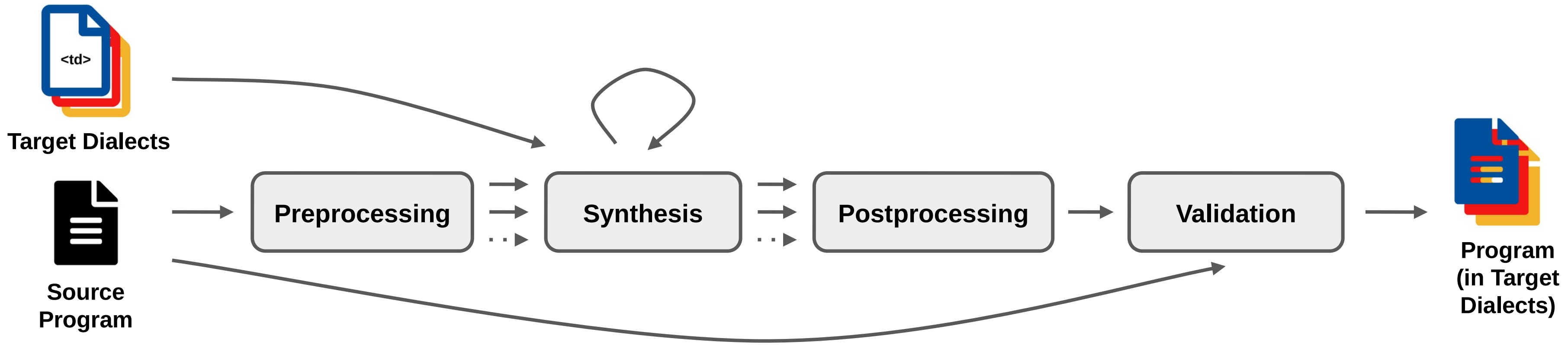
<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Program (in Target Dialects)

mlirSynth

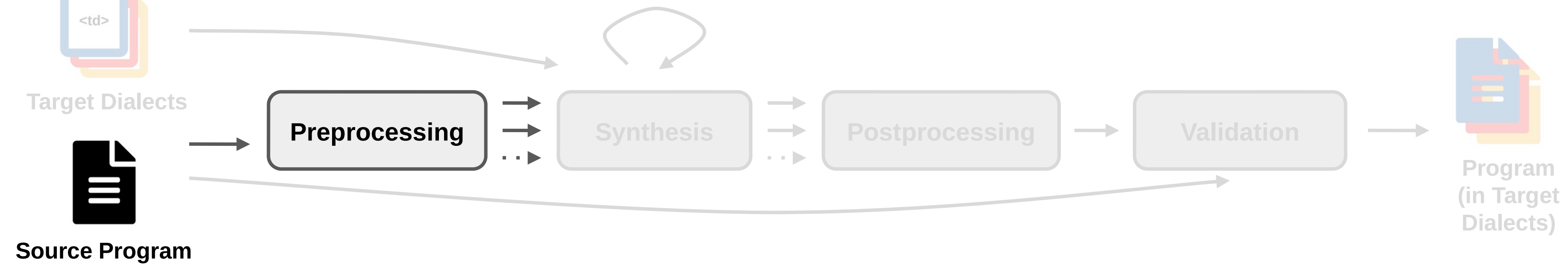


<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Program (in Target Dialects)

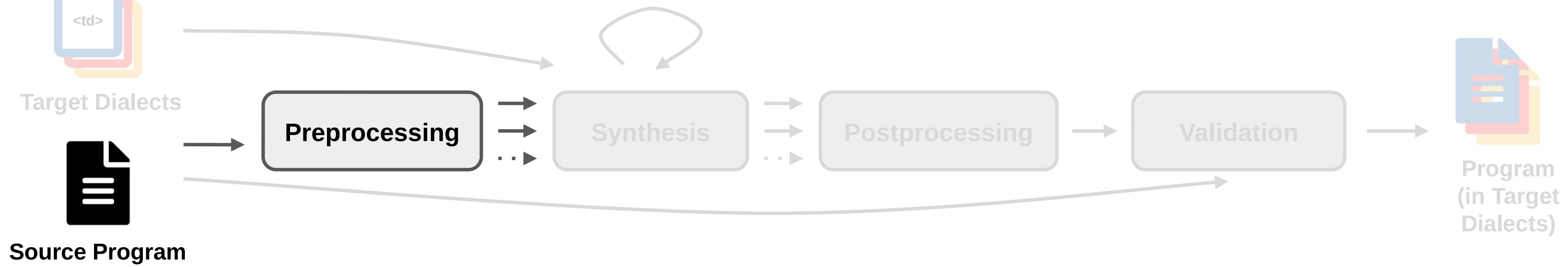
mlirSynth



<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)

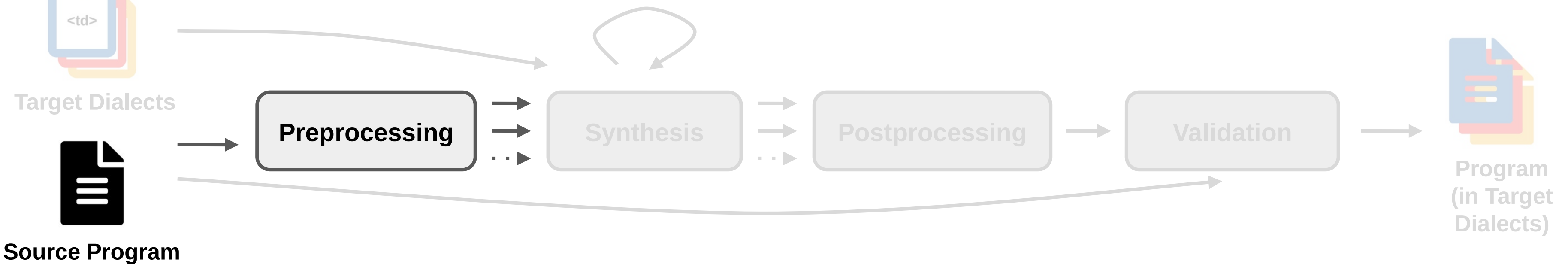


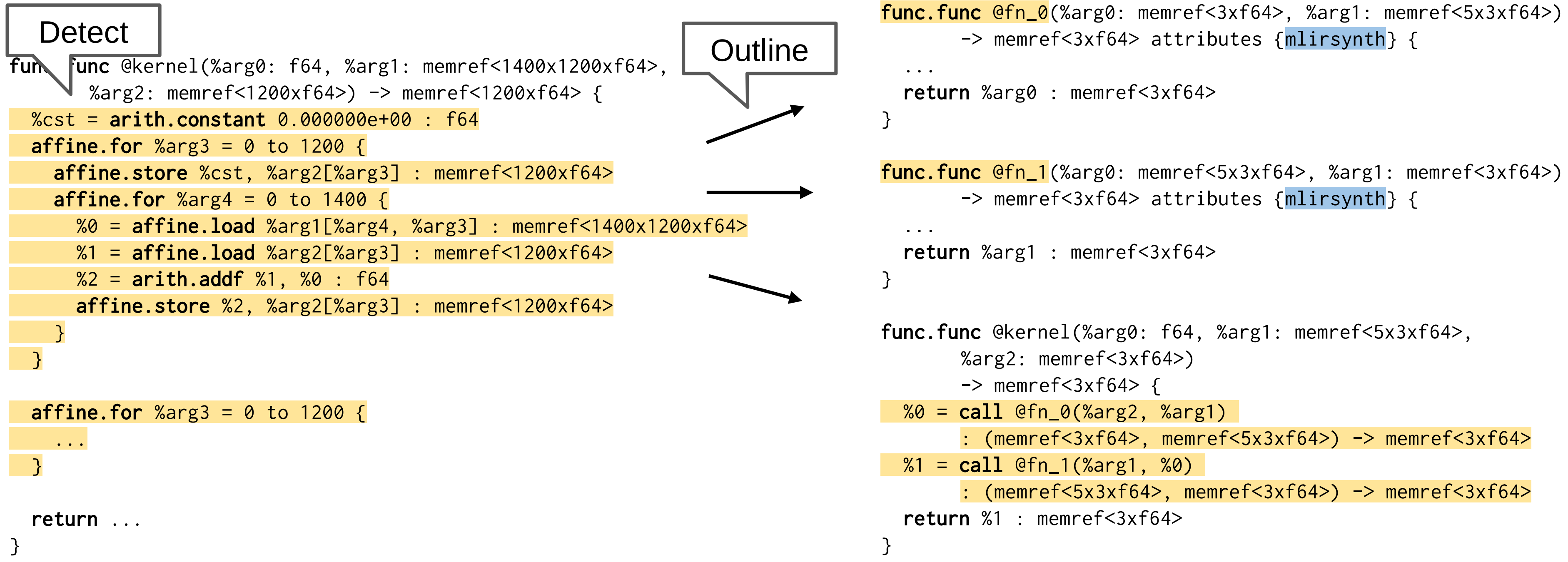
```
func.func @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
       %arg2: memref<1200xf64>) -> memref<1200xf64> {
  %cst = arith.constant 0.000000e+00 : f64
  affine.for %arg3 = 0 to 1200 {
    affine.store %cst, %arg2[%arg3] : memref<1200xf64>
    affine.for %arg4 = 0 to 1400 {
      %0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
      %1 = affine.load %arg2[%arg3] : memref<1200xf64>
      %2 = arith.addf %1, %0 : f64
      affine.store %2, %arg2[%arg3] : memref<1200xf64>
    }
  }

  affine.for %arg3 = 0 to 1200 {
    ...
  }

  return ...
}
```



Detect

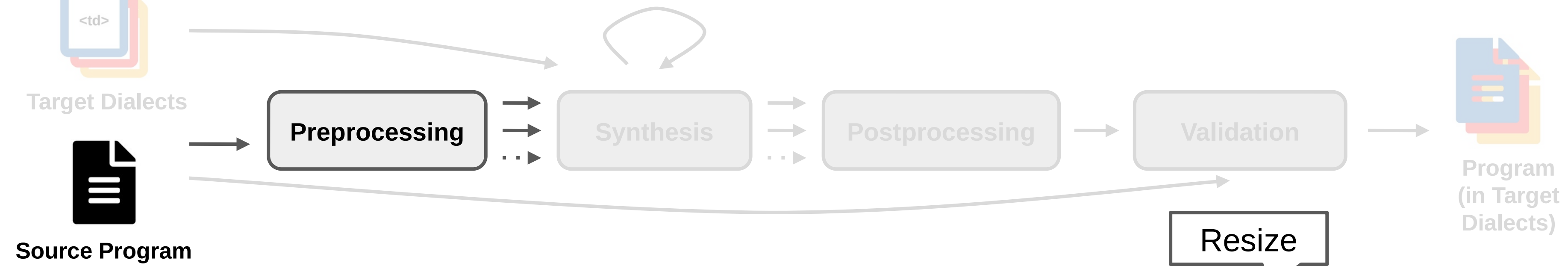
```
func.func @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
    %arg2: memref<1200xf64>) -> memref<1200xf64> {
  %cst = arith.constant 0.000000e+00 : f64
  affine.for %arg3 = 0 to 1200 {
    affine.store %cst, %arg2[%arg3] : memref<1200xf64>
    affine.for %arg4 = 0 to 1400 {
      %0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
      %1 = affine.load %arg2[%arg3] : memref<1200xf64>
      %2 = arith.addf %1, %0 : f64
      affine.store %2, %arg2[%arg3] : memref<1200xf64>
    }
  }

  affine.for %arg3 = 0 to 1200 {
    ...
  }

  return ...
}
```





```
fun  func @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
         %arg2: memref<1200xf64>) -> memref<1200xf64> {
  %cst = arith.constant 0.000000e+00 : f64
  affine.for %arg3 = 0 to 1200 {
    affine.store %cst, %arg2[%arg3] : memref<1200xf64>
    affine.for %arg4 = 0 to 1400 {
      %0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
      %1 = affine.load %arg2[%arg3] : memref<1200xf64>
      %2 = arith.addf %1, %0 : f64
      affine.store %2, %arg2[%arg3] : memref<1200xf64>
    }
  }

  affine.for %arg3 = 0 to 1200 {
    ...
  }

  return ...
}
```

```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  ...
  return %arg0 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  ...
  return %arg1 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
       %arg2: memref<3xf64>)
       -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
       : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
       : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```



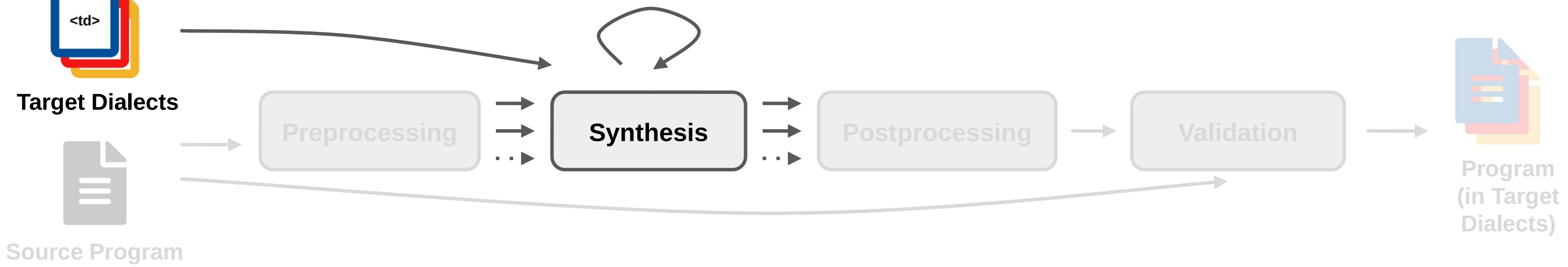
Detect

```
fun  func @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
       %arg2: memref<1200xf64>) -> memref<1200xf64> {
  %cst = arith.constant 0.000000e+00 : f64
  affine.for %arg3 = 0 to 1200 {
    affine.store %cst, %arg2[%arg3] : memref<1200xf64>
    affine.for %arg4 = 0 to 1400 {
      %0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
      %1 = affine.load %arg2[%arg3] : memref<1200xf64>
      %2 = arith.addf %1, %0 : f64
      affine.store %2, %arg2[%arg3] : memref<1200xf64>
    }
  }

  affine.for %arg3 = 0 to 1200 {
    ...
  }

  return ...
}
```

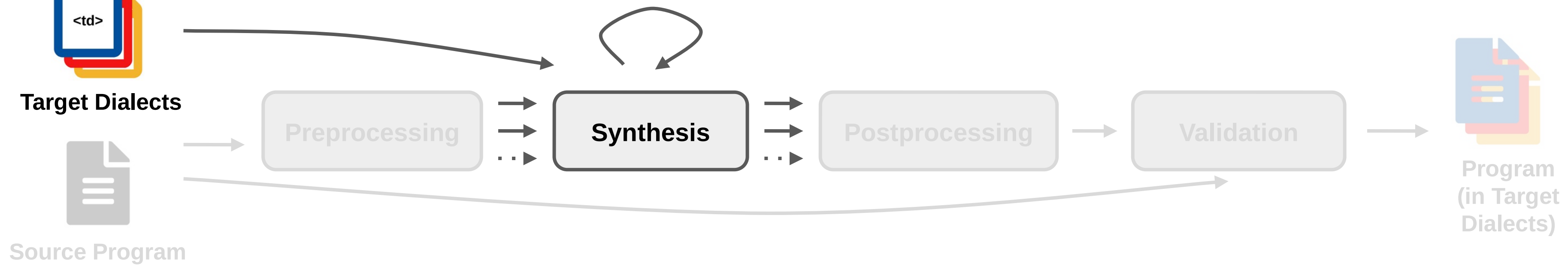
Outline

Resize

```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  ...
  return %arg0 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  ...
  return %arg1 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
       %arg2: memref<3xf64>)
       -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
       : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
       : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```

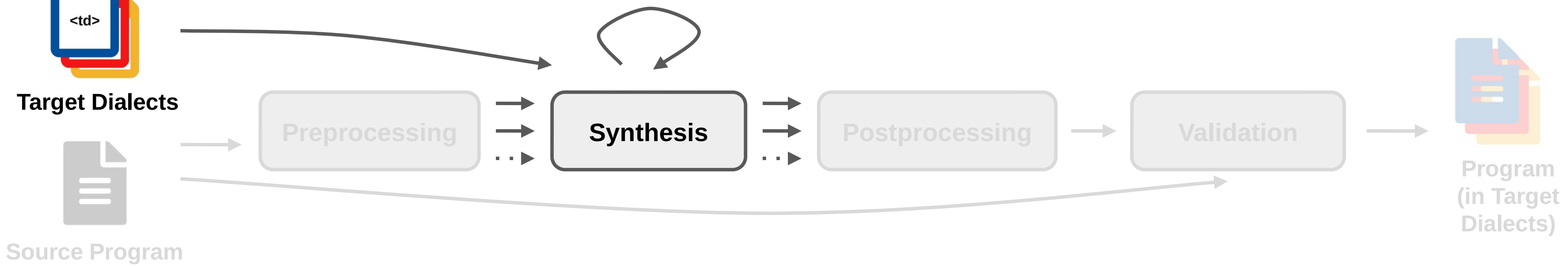


<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)



Specification

- Generate Input/Output example

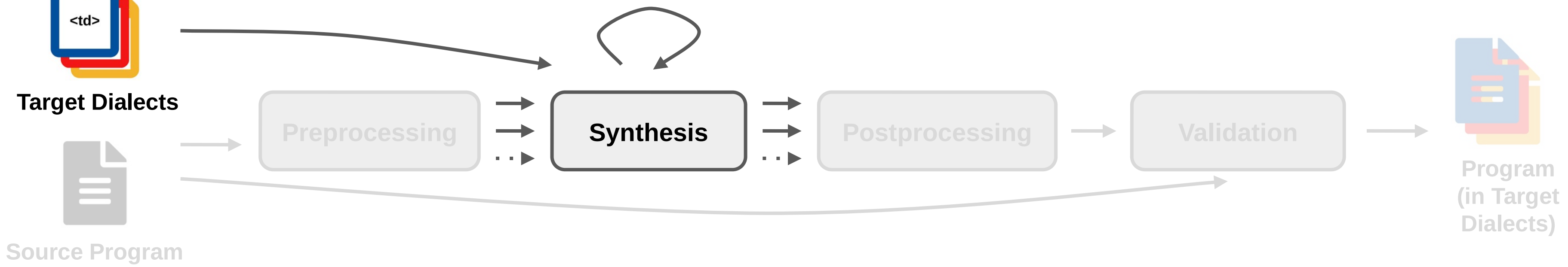


Specification

- Generate Input/Output example

Bottom-up enumerative search

- Progressively grow a candidate set by combining simpler to more complex ones
- Initialization: Basic programs (returning arguments, constants)
- Terminate when specification matched



Specification

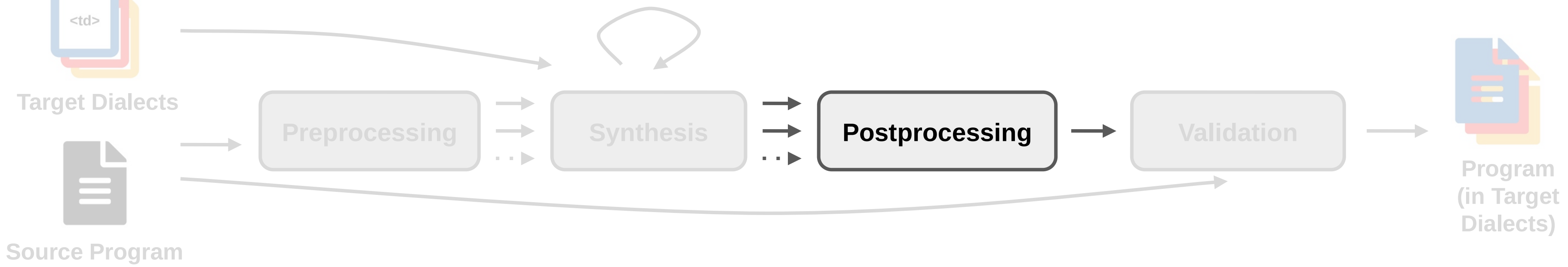
- Generate Input/Output example

Bottom-up enumerative search

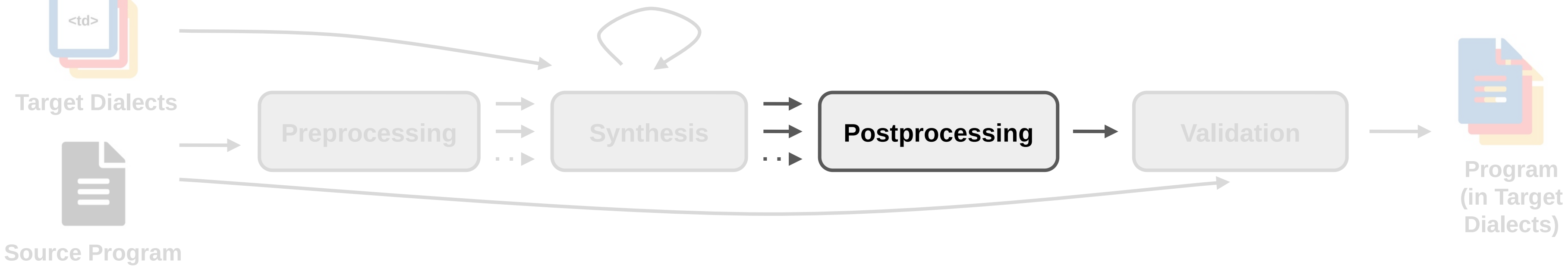
- Progressively grow a candidate set by combining simpler to more complex ones
- Initialization: Basic programs (returning arguments, constants)
- Terminate when specification matched

Optimization techniques

- Type correct by construction
- Identify classes of observationally equivalent candidates
- Polyhedral-based heuristics for guiding synthesis



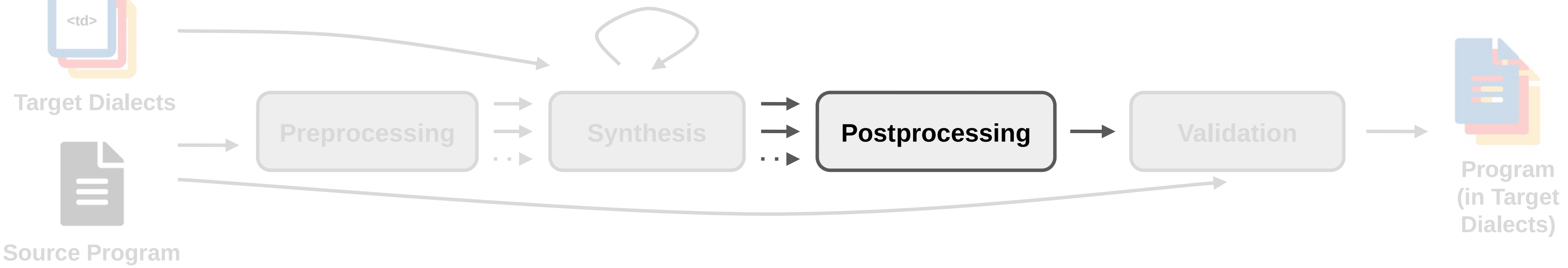
```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  %1 = op(%0, %arg1) : memref<3xf64>
  return %1 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  return %0 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
       %arg2: memref<3xf64>)
       -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
       : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
       : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```



```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  %1 = op(%0, %arg1) : memref<3xf64>
  return %1 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
       -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  return %0 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
       %arg2: memref<3xf64>)
       -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
       : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
       : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```

Inline

```
func.func @kernel(%arg0: f64,
       %arg1: memref<1400x1200xf64>,
       %arg2: memref<1200xf64>)
       -> memref<1200xf64> {

  // fn_0
  %0 = op(%arg2, %arg1) : memref<1200xf64>
  %1 = op(%0, %arg1) : memref<1200xf64>

  // fn_1
  %2 = op(%arg1, %1) : memref<1200xf64>

  return %2 : memref<1200xf64>
}
```

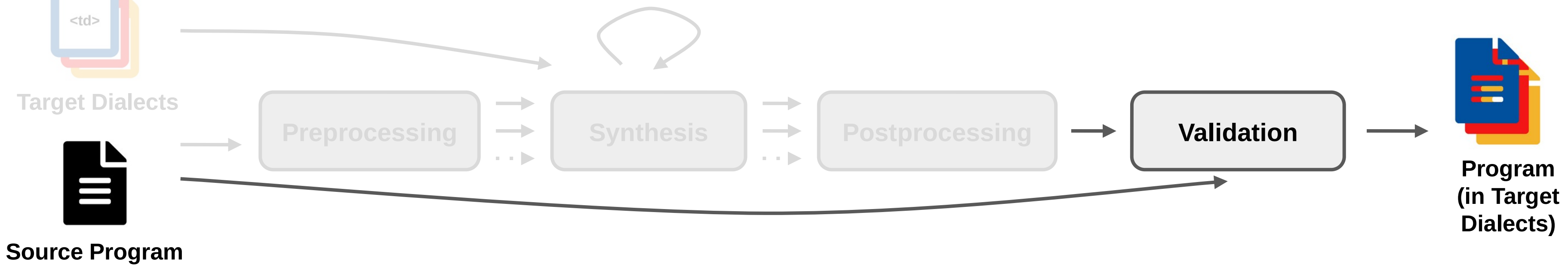


```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
        -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  %1 = op(%0, %arg1) : memref<3xf64>
  return %1 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
        -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  return %0 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
        %arg2: memref<3xf64>)
        -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
        : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
        : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```

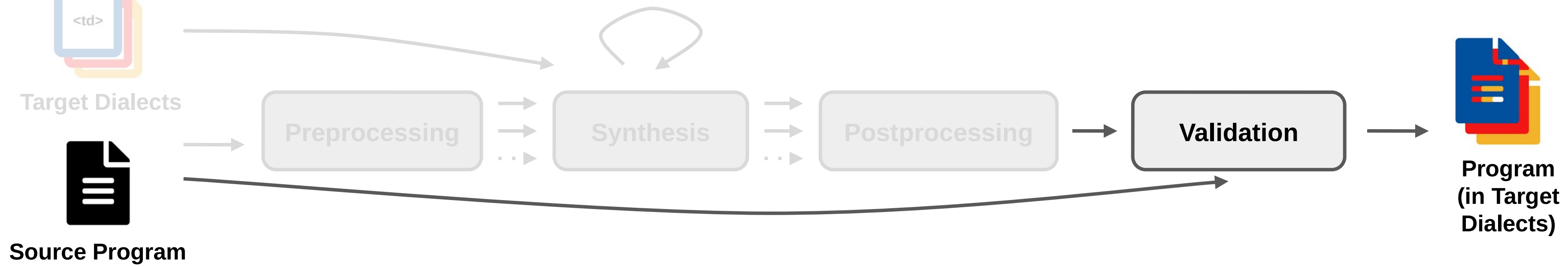
Inline

Resize

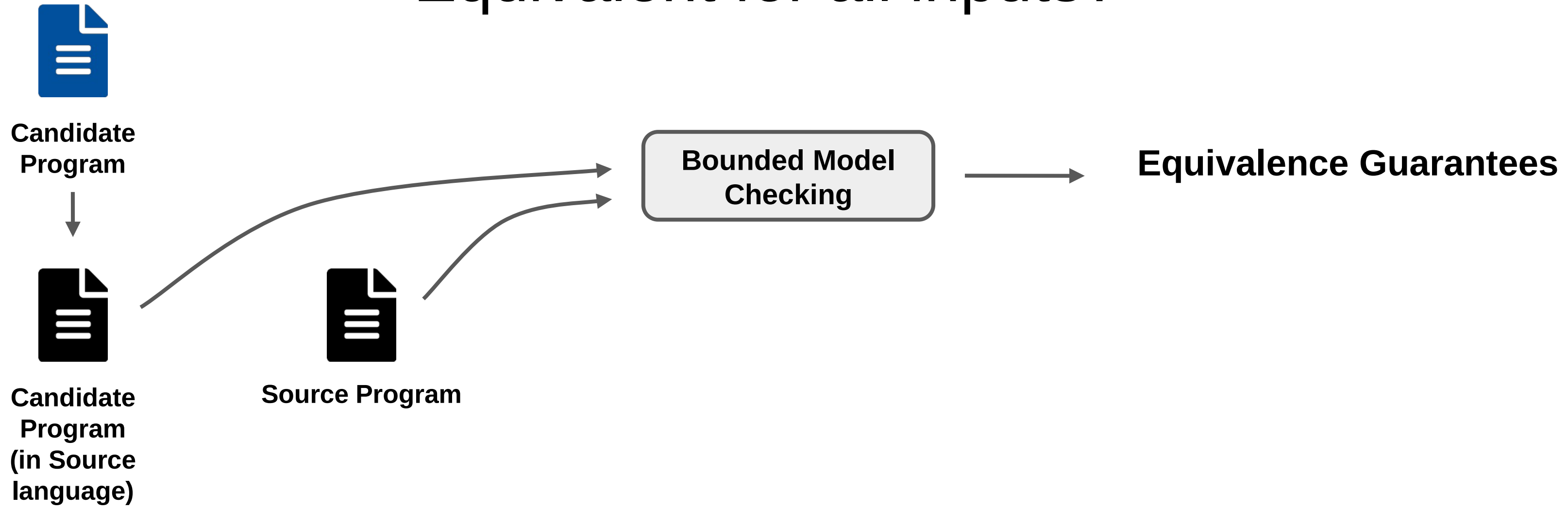
```
func.func @kernel(%arg0: f64,
        %arg1: memref<1400x1200xf64>,
        %arg2: memref<1200xf64>)
        -> memref<1200xf64> {

  // fn_0
  %0 = op(%arg2, %arg1) : memref<1200xf64>
  %1 = op(%0, %arg1) : memref<1200xf64>

  // fn_1
  %2 = op(%arg1, %1) : memref<1200xf64>

  return %2 : memref<1200xf64>
}
```

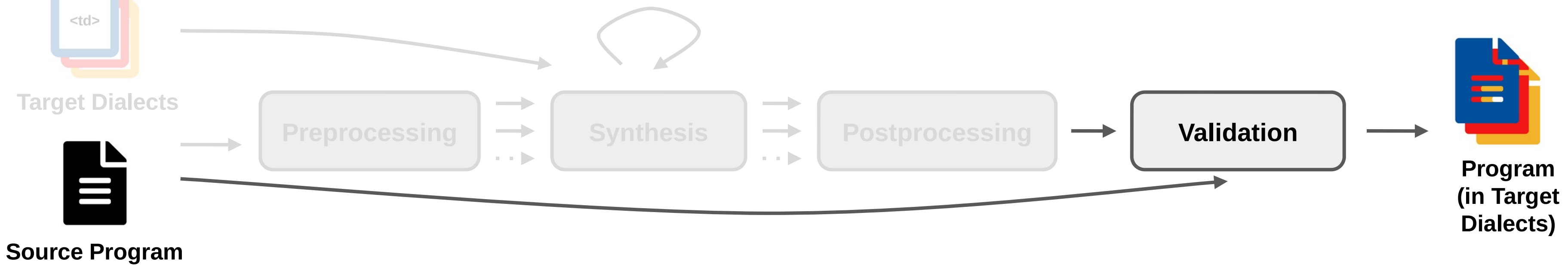


Equivalent for all inputs?

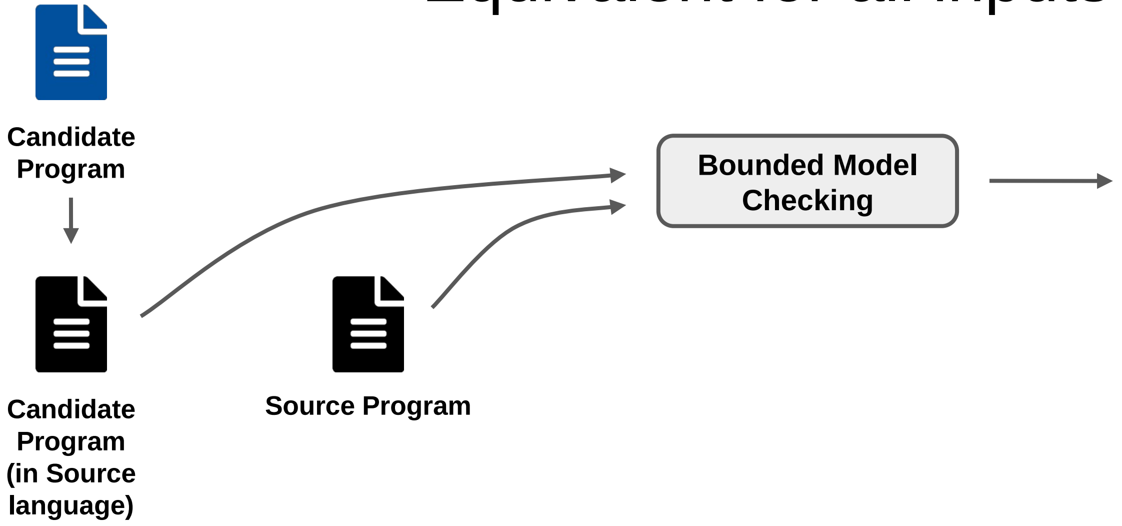


Equivalent for all inputs?



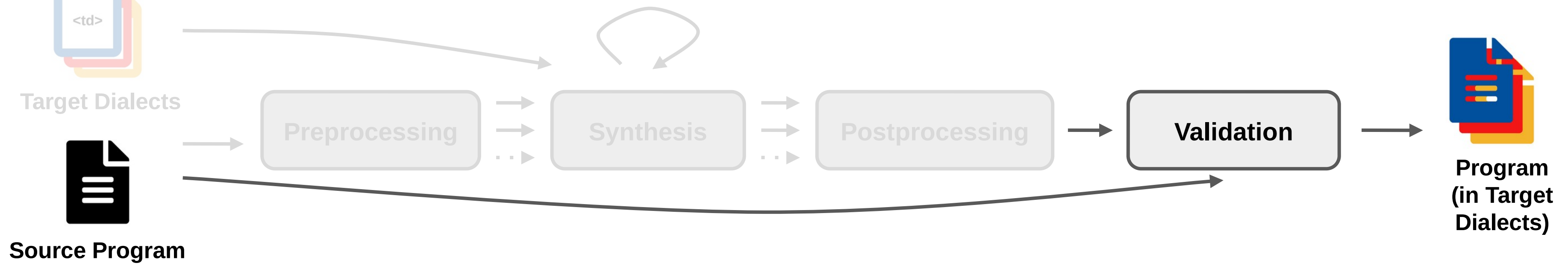


Equivalent for all inputs?

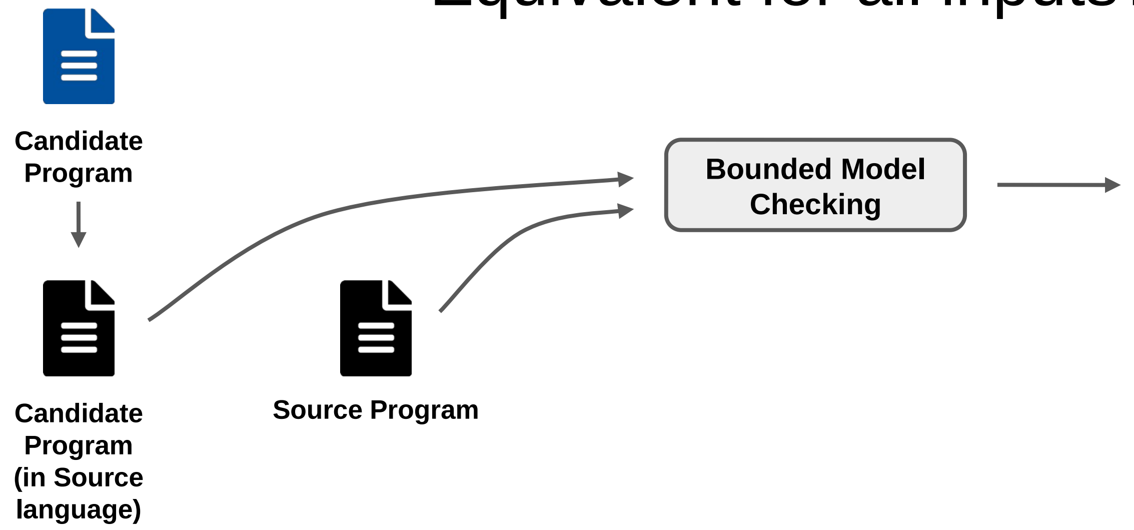


Equivalence Guarantees

a) Float arithmetic

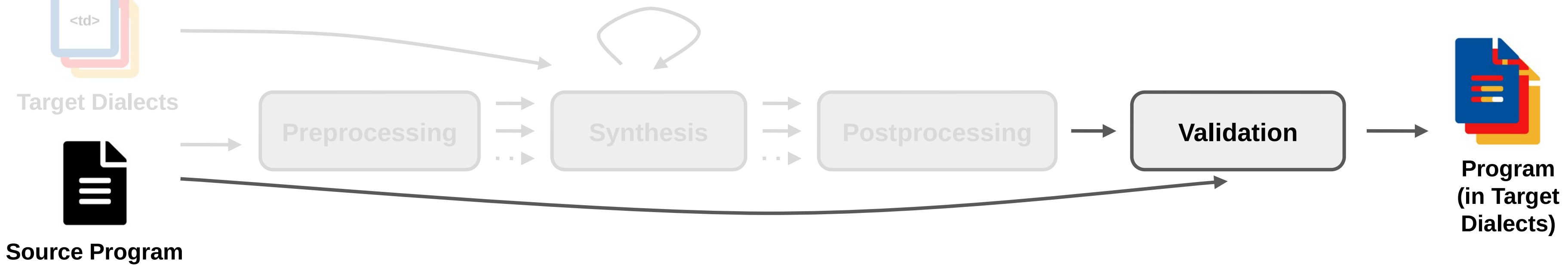


Equivalent for all inputs?

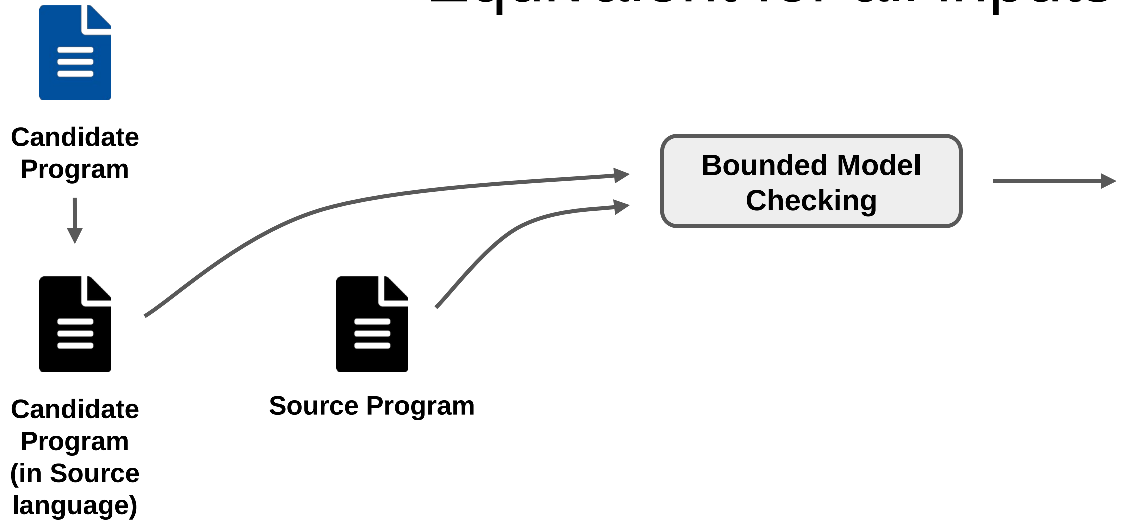


Equivalence Guarantees

a) Float arithmetic
b) Float arithmetic, permitting small δ

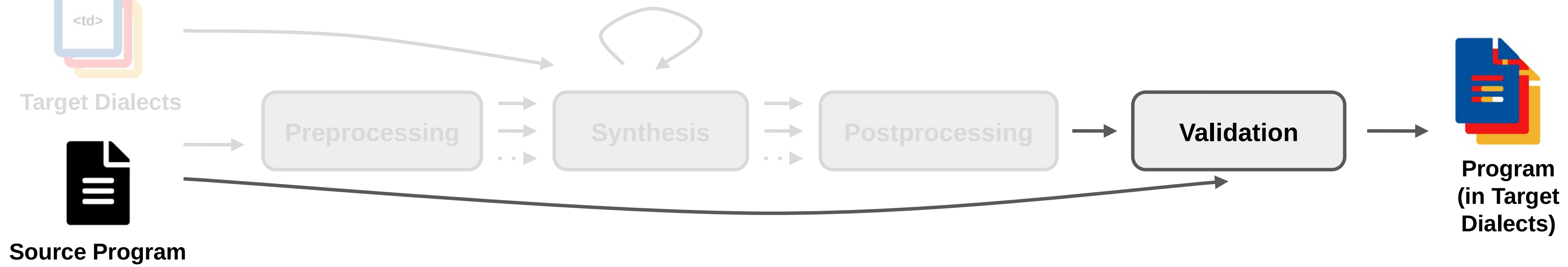


Equivalent for all inputs?

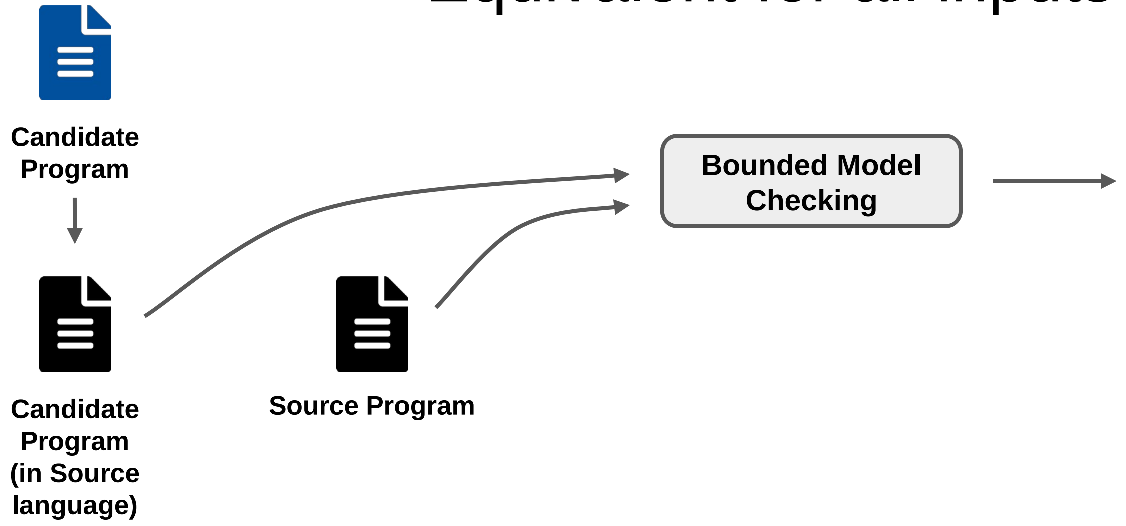


Equivalence Guarantees

a) Float arithmetic
b) Float arithmetic, permitting small δ
c) Integer arithmetic



Equivalent for all inputs?



Equivalence Guarantees

a) Float arithmetic
b) Float arithmetic, permitting small δ
c) Integer arithmetic
d) Testing I/O equivalence

Results

Coverage

Benchmark: PolyBench

- Solvers
- Data Mining
- Linear Algebra BLAS
- Linear Algebra Kernels
- Stencils
- Medley

→Total: 15 Programs

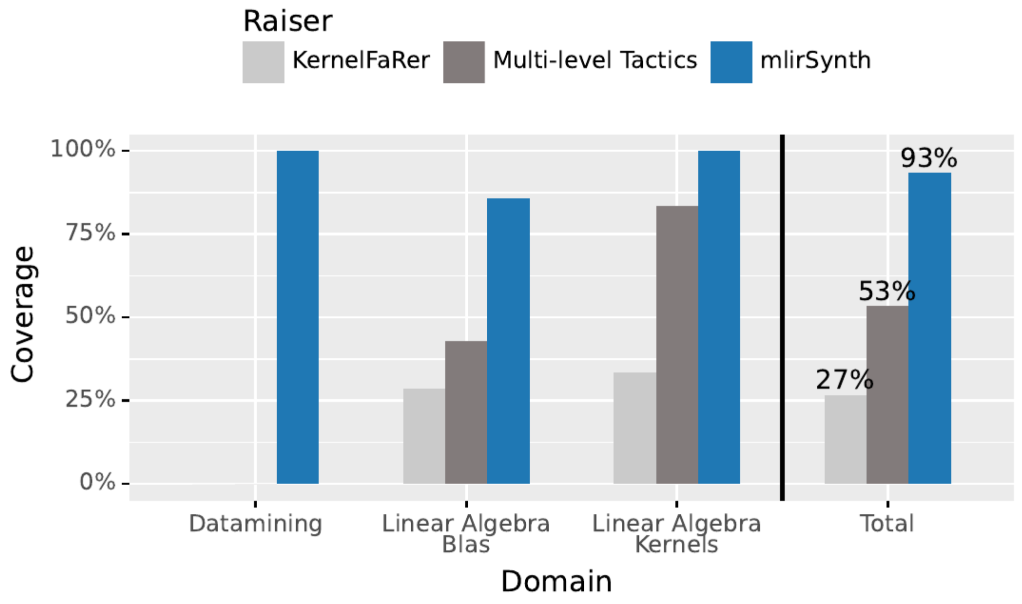
Results

Coverage

Benchmark: PolyBench

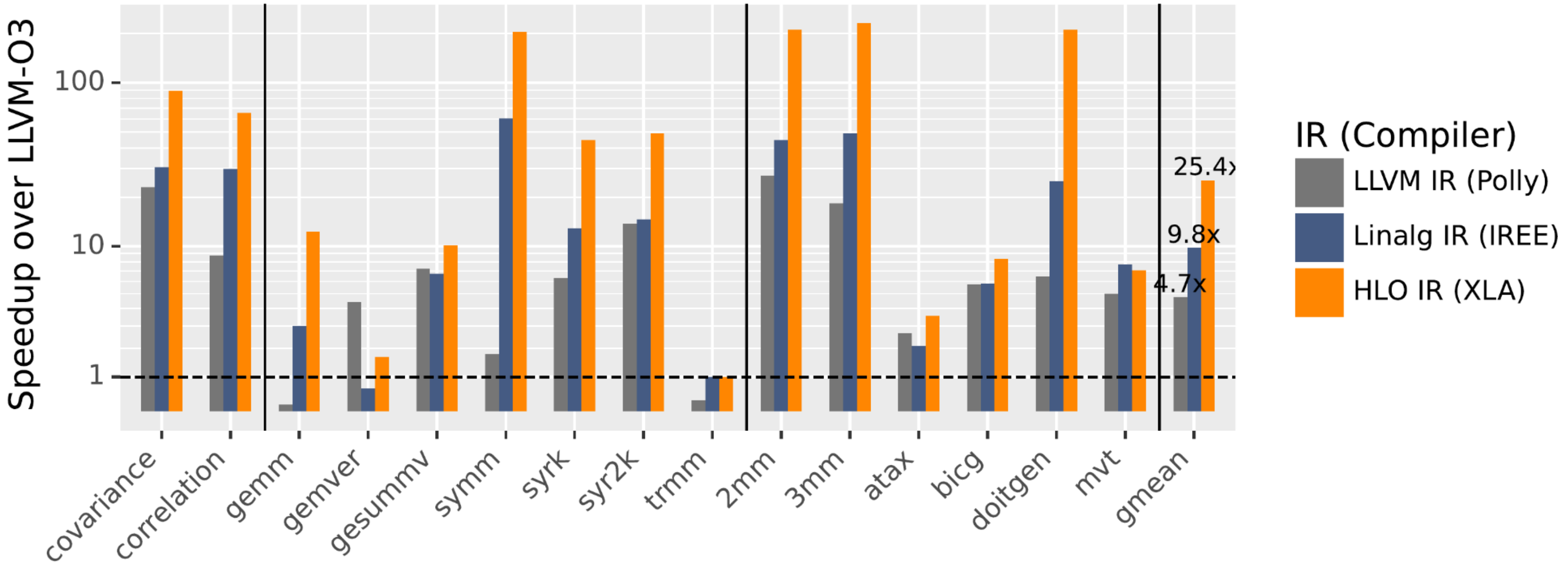
- Solvers
- Data Mining
- Linear Algebra BLAS
- Linear Algebra Kernels
- Stencils
- Medley

→ Total: 15 Programs



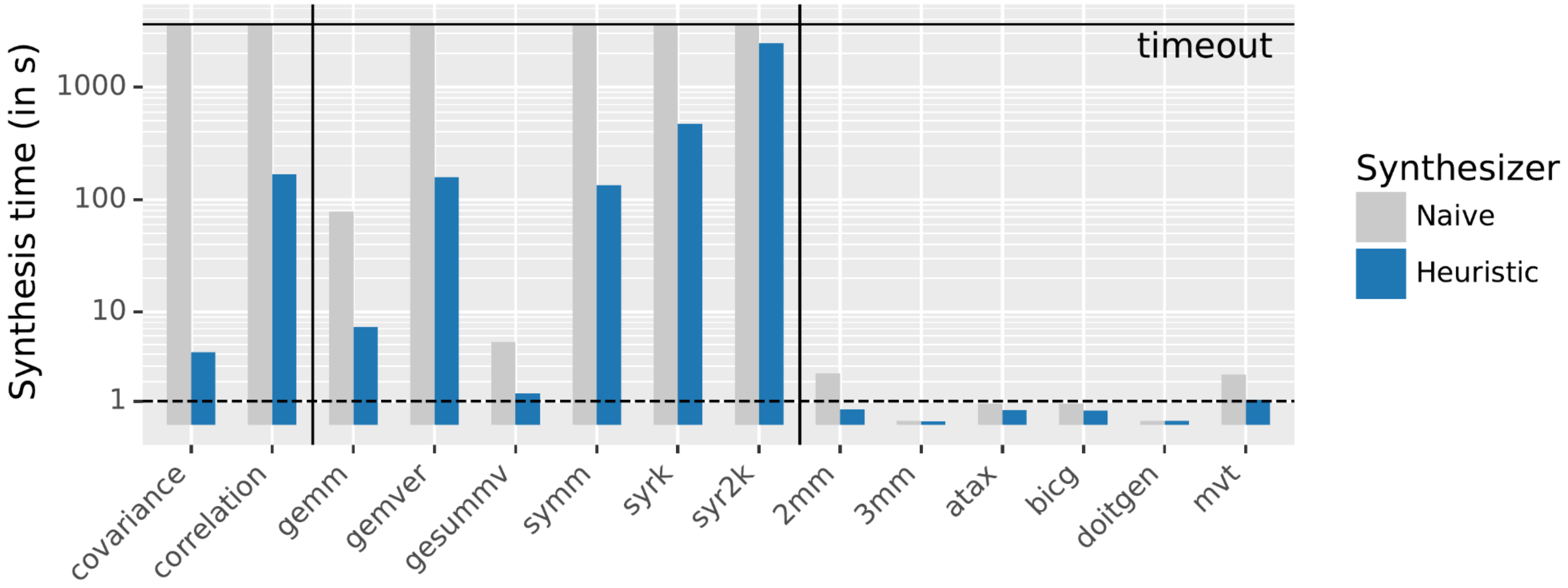
Results

Performance

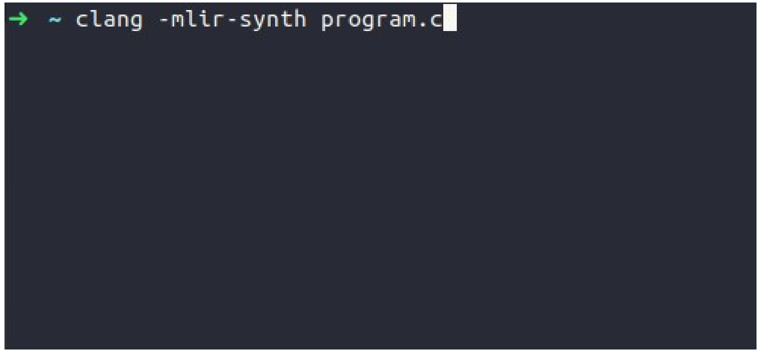


CPU: AMD Ryzen 9 3900X

Synthesis Time



Future Work

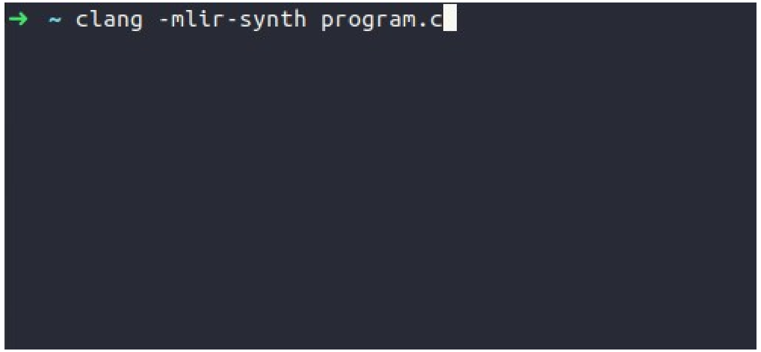


```
→ ~ clang -mlir-synth program.c
```

Method

Applicability

Future Work



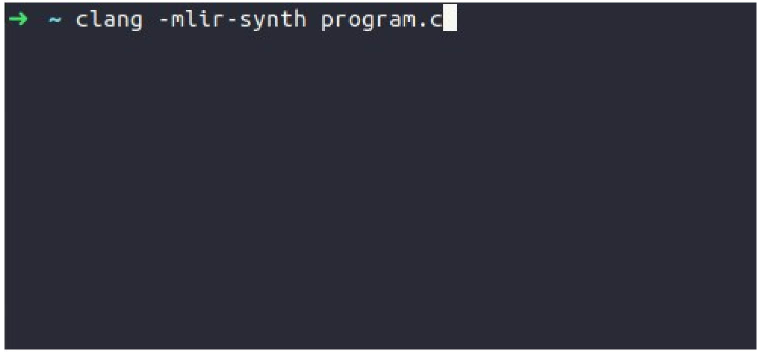
```
➜  ~ clang -mlir-synth program.c
```

Method

- Detection of raisable code regions + their dialect

Applicability

Future Work



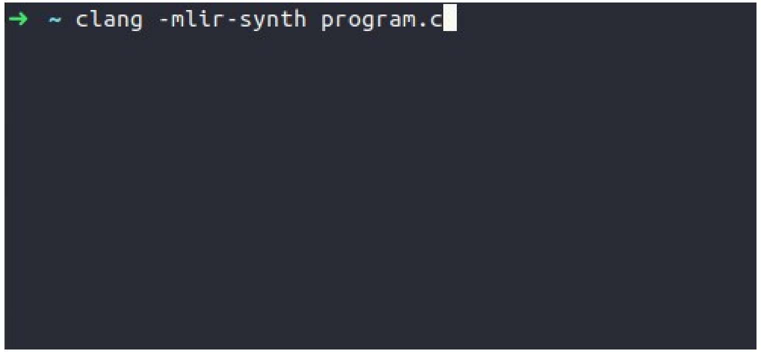
```
→ ~ clang -mlir-synth program.c
```

Method

- Detection of raisable code regions + their dialect
- Speed up search

Applicability

Future Work



```
➜  ~ clang -mlir-synth program.c
```

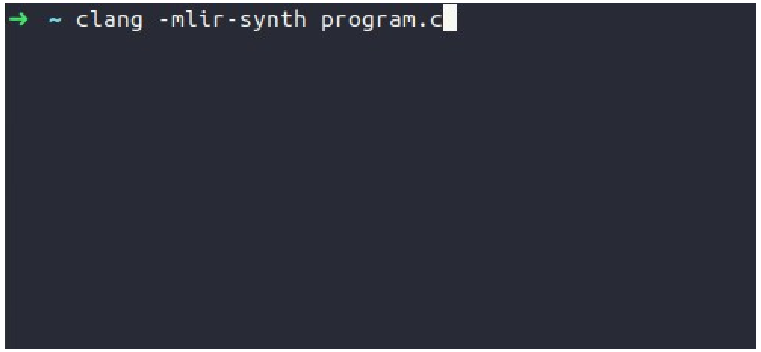
Method

- Detection of raisable code regions + their dialect
- Speed up search

Applicability

- More source dialects

Future Work



```
➜  ~ clang -mlir-synth program.c
```

Method

- Detection of raisable code regions + their dialect
- Speed up search

Applicability

- More source dialects
- More target dialects

Takeaways

Takeaways

- mlirSynth raises programs to high-level dialects using program synthesis
  - Robust ✅
  - Automatic ✅
  - Fast ❌

Takeaways

- mlirSynth raises programs to high-level dialects using program synthesis
  - Robust ✅
  - Automatic ✅
  - Fast ❌

- Raised programs lead to significantly higher performance
  - DSL compilers
  - Hardware accelerators

Takeaways

- mlirSynth raises programs to high-level dialects using program synthesis
    - Robust ✅
    - Automatic ✅
    - Fast ❌

- Raised programs lead to significantly higher performance
    - DSL compilers
    - Hardware accelerators

- Several interesting directions!

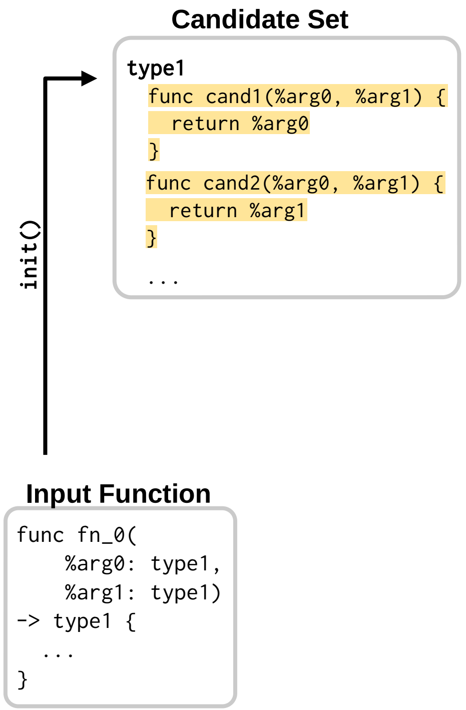
alexander.brauckmann@ed.ac.uk

Synthesis example

Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Synthesis example



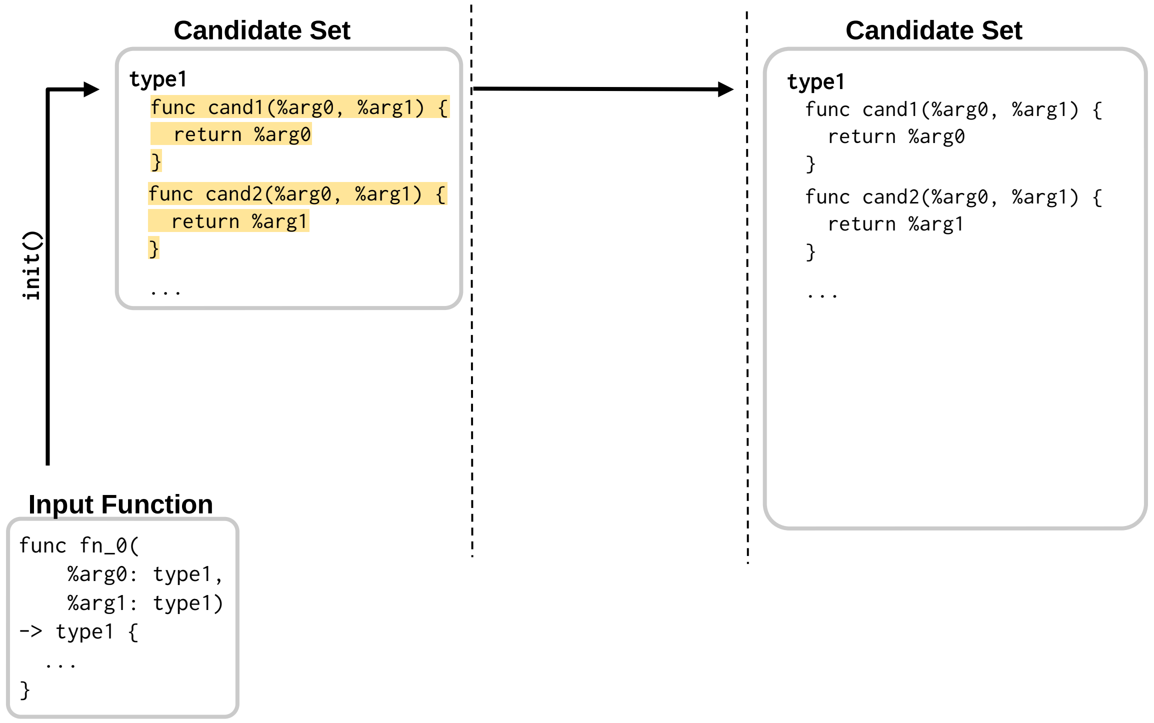
Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }

  ...
```

Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Synthesis example

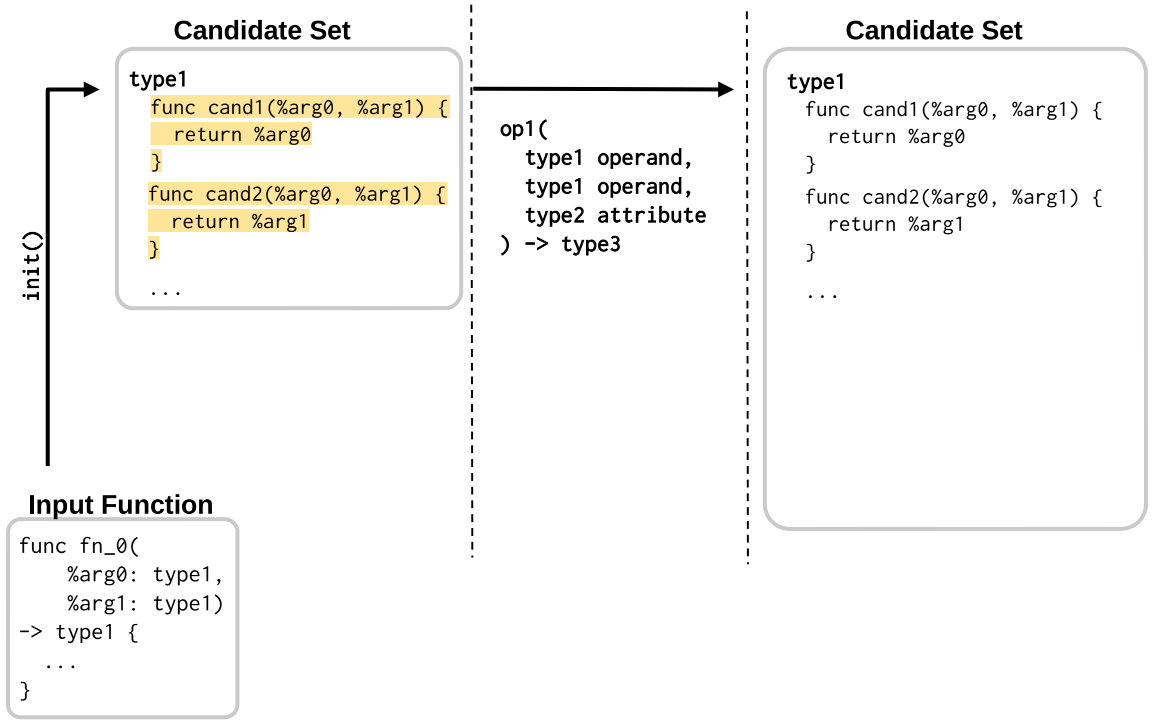


```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Synthesis example



Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Candidate Set

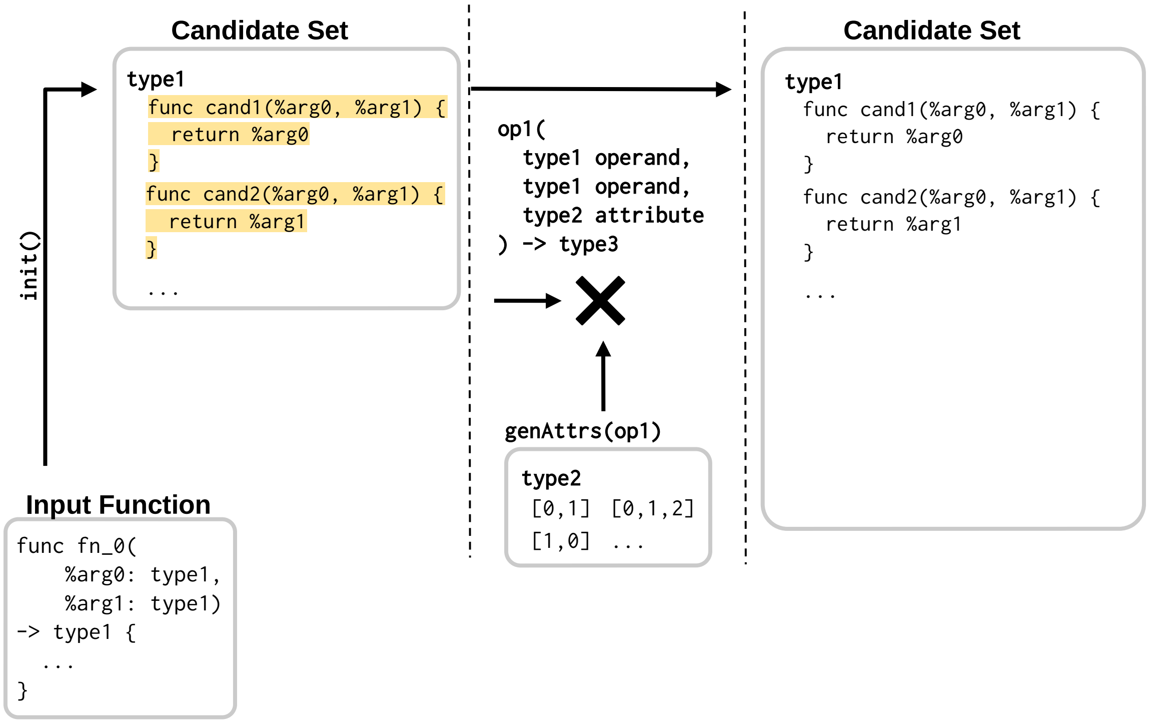
```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

Synthesis example



```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

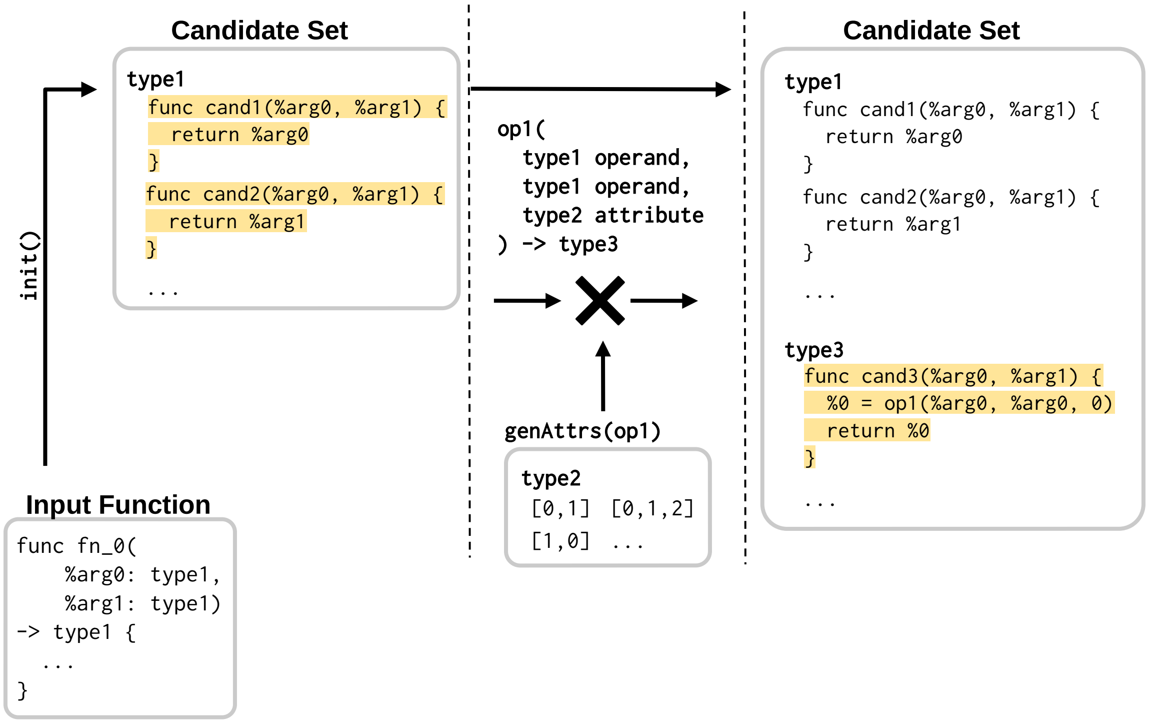
```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Synthesis example



```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

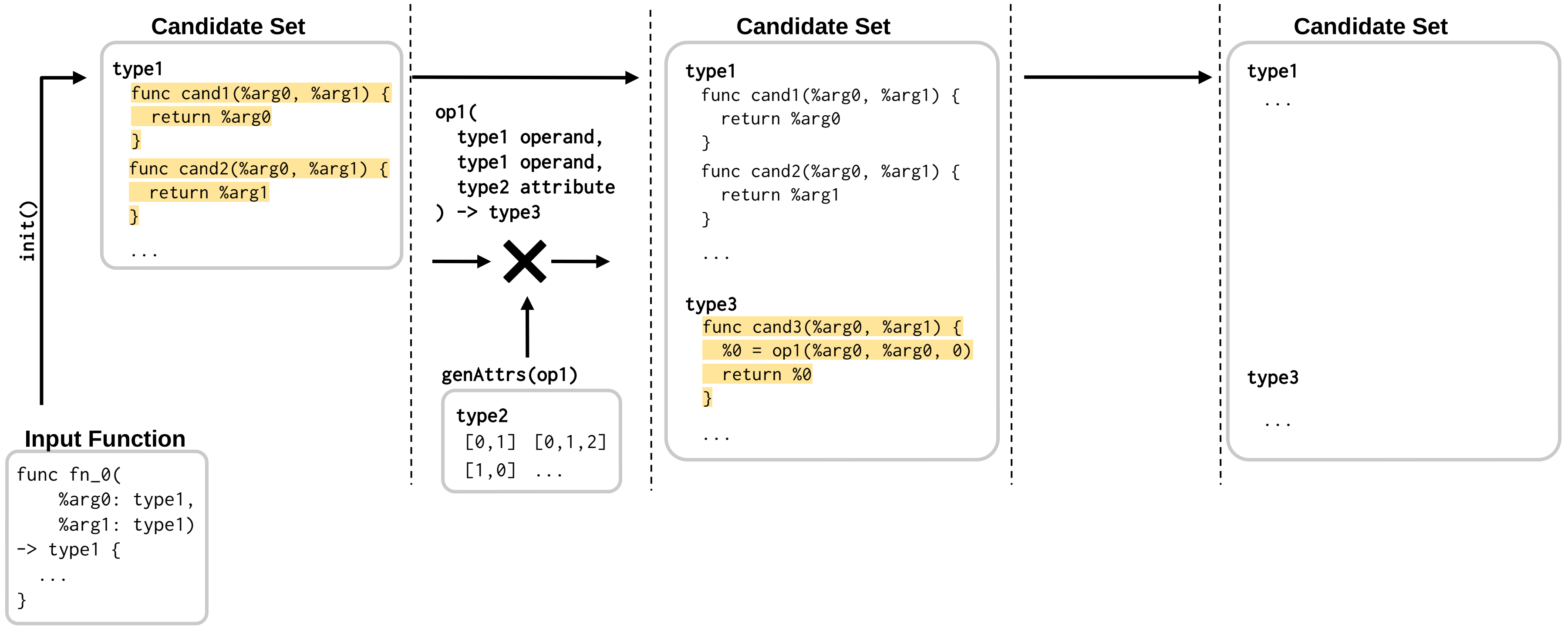
```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
op1(
   type1 operand,
   type1 operand,
   type2 attribute
) -> type3
```

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }
  ...
```

Synthesis example



```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

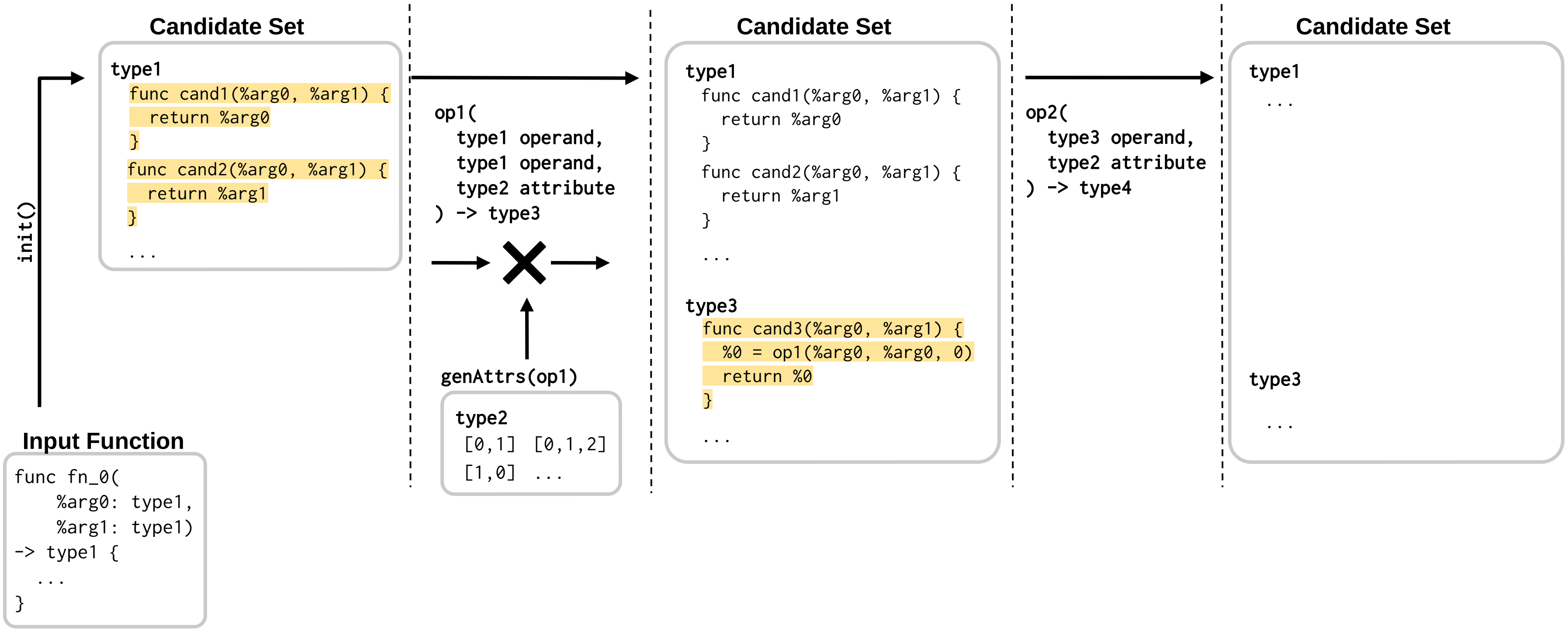
```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }

  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }
  ...
```

```
type1
  ...

type3
  ...
```

Synthesis example



Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }

  ...
```

```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

genAttrs(op1)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

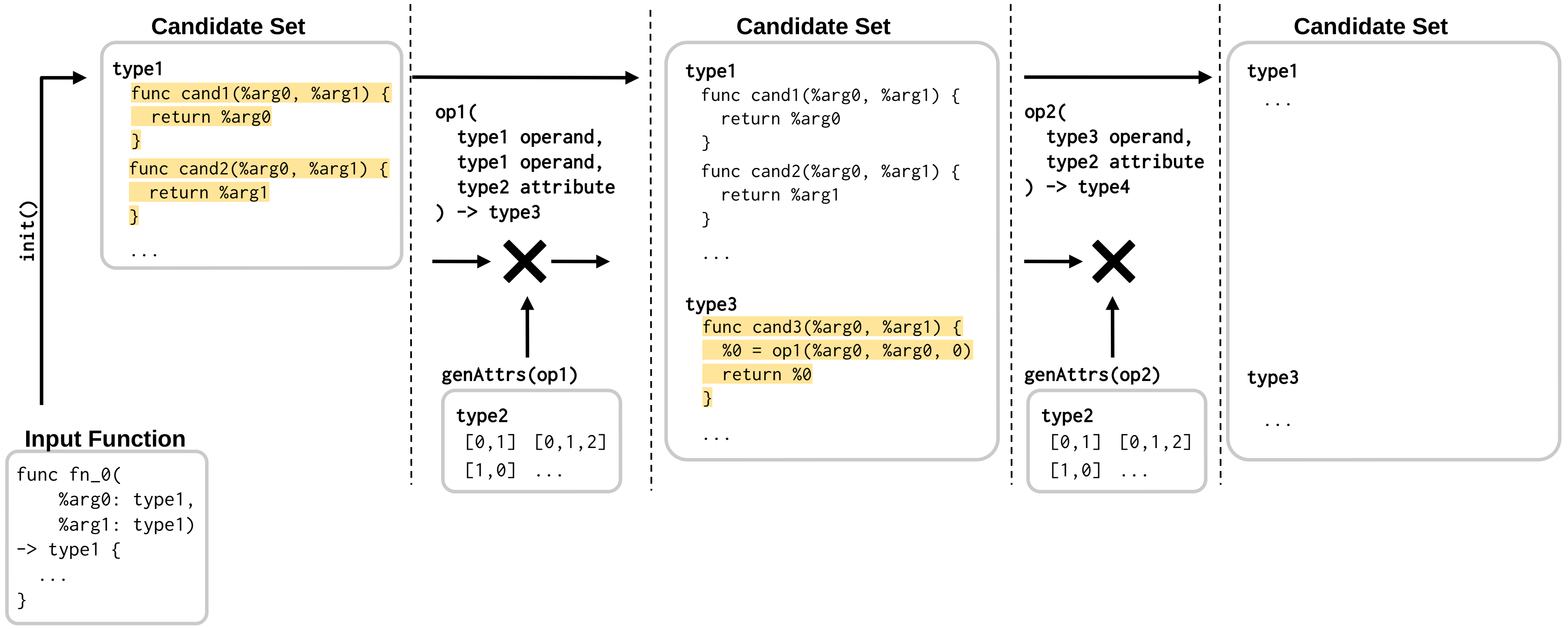
```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }

  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }

  ...
```

```
op2(
  type3 operand,
  type2 attribute
) -> type4
```

Candidate Set

```
type1
  ...

type3
  ...
```

Synthesis example



Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

genAttrs(op1)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }
  ...
```

```
op2(
  type3 operand,
  type2 attribute
) -> type4
```

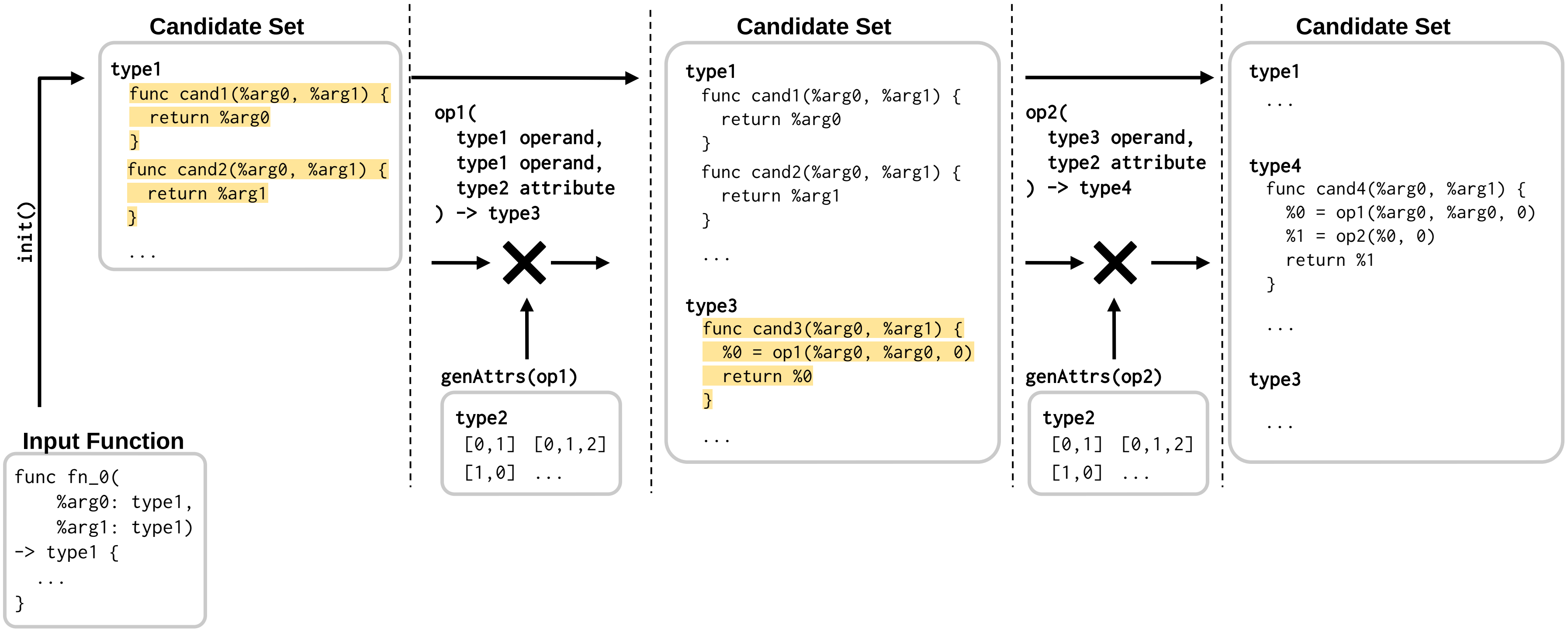
genAttrs(op2)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  ...

type3
  ...
```

Synthesis example



Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

genAttrs(op1)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }
  ...
```

```
op2(
  type3 operand,
  type2 attribute
) -> type4
```

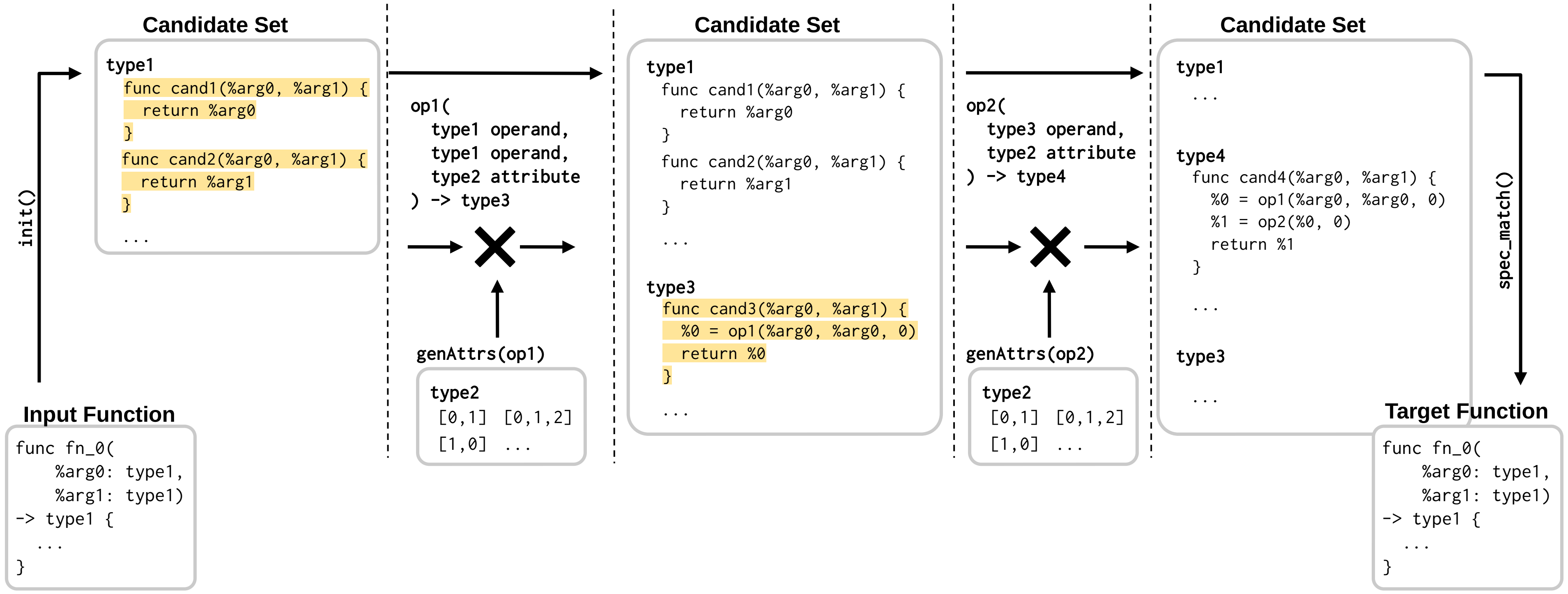
genAttrs(op2)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  ...

type4
  func cand4(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    %1 = op2(%0, 0)
    return %1
  }
  ...

type3
  ...
```

Synthesis example



Input Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...
```

```
op1(
  type1 operand,
  type1 operand,
  type2 attribute
) -> type3
```

genAttrs(op1)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  func cand1(%arg0, %arg1) {
    return %arg0
  }
  func cand2(%arg0, %arg1) {
    return %arg1
  }
  ...

type3
  func cand3(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    return %0
  }
  ...
```

```
op2(
  type3 operand,
  type2 attribute
) -> type4
```

genAttrs(op2)

```
type2
 [0,1]  [0,1,2]
 [1,0]  ...
```

Candidate Set

```
type1
  ...

type4
  func cand4(%arg0, %arg1) {
    %0 = op1(%arg0, %arg0, 0)
    %1 = op2(%0, 0)
    return %1
  }
  ...

type3
  ...
```

Target Function

```
func fn_0(
    %arg0: type1,
    %arg1: type1)
-> type1 {
  ...
}
```