# Synthesizing Practical Transfer Functions in Dataflow Analysis

**Yuyou Fan**, Xuanyu Peng, Dominic Kennedy, Ben Greenman, John Regehr, Loris D'Antoni

# Dataflow Analysis Recap

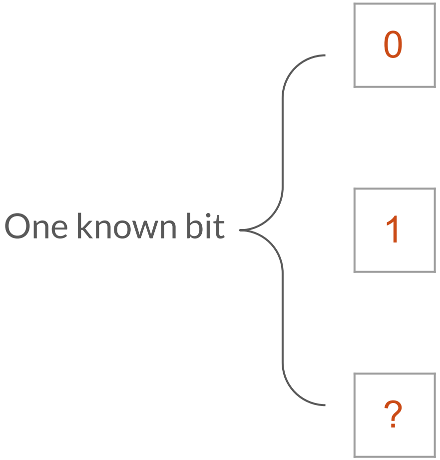
# What Is Dataflow Analysis?

Information/Properties that holds among all executions

```
define i4 @func(i4 %arg0) {
  %and3 = and i4 %arg0, 3
  %and1 = and i4 %arg0, 1
  %xor = xor i4 %and3, %and1
  ret i4 %xor
}
```

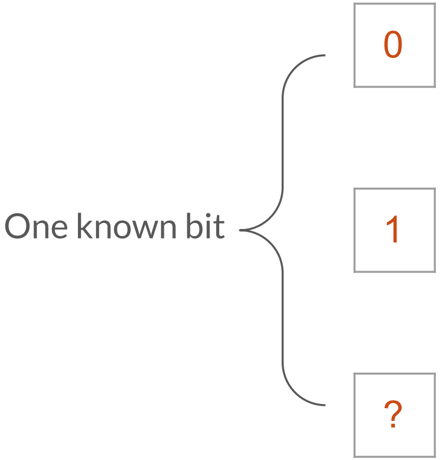
# Known Bits

Known Bits Information tries to determine if a certain bit is always one or zero among all executions.



# Known Bits

Known Bits Information tries to determine if a certain bit is always one or zero among all executions.




Possible runtime values

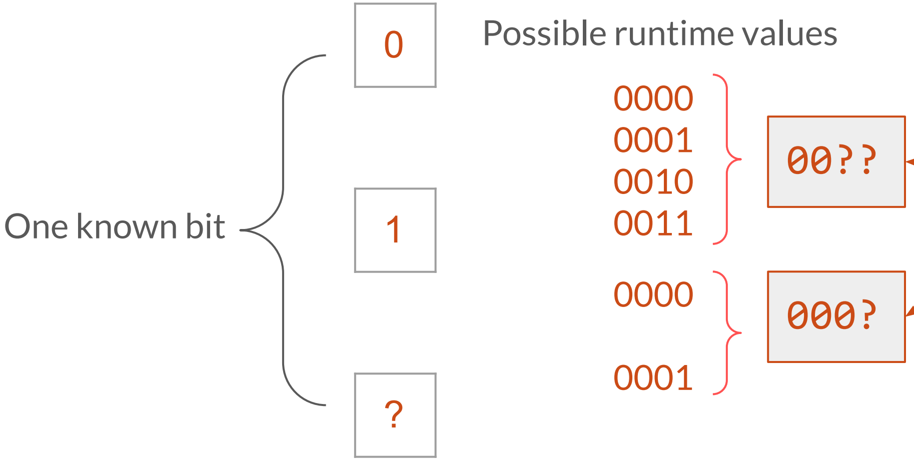
0000
0001
0010
0011

00??

```
define i4 @func(i4 %arg0) {
  %and3 = and i4 %arg0, 3
  %and1 = and i4 %arg0, 1
  %xor = xor i4 %and3, %and1
  ret i4 %xor
}
```

# Known Bits

Known Bits Information tries to determine if a certain bit is always one or zero among all executions.




```
define i4 @func(i4 %arg0) {
  %and3 = and i4 %arg0, 3
  %and1 = and i4 %arg0, 1
  %xor = xor i4 %and3, %and1
  ret i4 %xor
}
```

# Implementation of Known Bits in LLVM

KnownBits → %knownZeros: integer
KnownBits → %knownOnes : integer

00?? → %knownZeros= 1100
00?? → %knownOnes= 0000

11?? → %knownZeros= 0000
11?? → %knownOnes= 1100

# Compute Known Bits

```
define i4 @func(i4 %arg0) {
  ......
  %xor = xor i4 %and3, %and1
}
```

00?? 000?

Xor Truth Table on Known Bits

| xor | 0 | 1 | ? |
|---|---|---|---|
| 0 | 0 | 1 | ? |
| 1 | 1 | 0 | ? |
| ? | ? | ? | ? |

# Compute Known Bits Fact by Transfer Function

```
define i4 @func(i4 %arg0) {
  ......
  %xor = xor i4 %and, %and1
}
```

| %xor | %and | %and1 |
|---|---|---|
| 00?? | 00?? | 000? |

Xor Truth Table on Known Bits

| xor | 0 | 1 | ? |
|---|---|---|---|
| 0 | 0 | 1 | ? |
| 1 | 1 | 0 | ? |
| ? | ? | ? | ? |

```
KnownBits &KnownBits::operator^=(const KnownBits &RHS) {
  // Result bit is 0 if both operand bits are 0 or both are 1.
  APInt Z = (Zero & RHS.Zero) | (One & RHS.One);
  // Result bit is 1 if one operand bit is 0 and the other is 1.
  One = (Zero & RHS.One) | (One & RHS.Zero);
  Zero = std::move(Z);
  return *this;
}
```

# Compute Known Bits Fact by Transfer Function

LLVM implements transfer functions for operations on Known Bits, and some operations are really complicated.

```
KnownBits computeForAddCarry(
    KnownBits &LHS, KnownBits &RHS,
    bool CarryZero, bool CarryOne) {
  APInt PossibleSumZero = LHS.getMaxValue() + RHS.getMaxValue() + !CarryZero;
  APInt PossibleSumOne = LHS.getMinValue() + RHS.getMinValue() + CarryOne;
  APInt CarryKnownZero = ~(PossibleSumZero ^ LHS.Zero ^ RHS.Zero);
  APInt CarryKnownOne = PossibleSumOne ^ LHS.One ^ RHS.One;
  APInt LHSKnownUnion = LHS.Zero | LHS.One;
  APInt RHSKnownUnion = RHS.Zero | RHS.One;
  APInt CarryKnownUnion = CarryKnownZero | CarryKnownOne;
  APInt Known = LHSKnownUnion & RHSKnownUnion & CarryKnownUnion;
  KnownBits KnownOut;
  KnownOut.Zero = ~PossibleSumZero & Known;
  KnownOut.One = PossibleSumOne & Known;
  return KnownOut;
}
```

## Summary

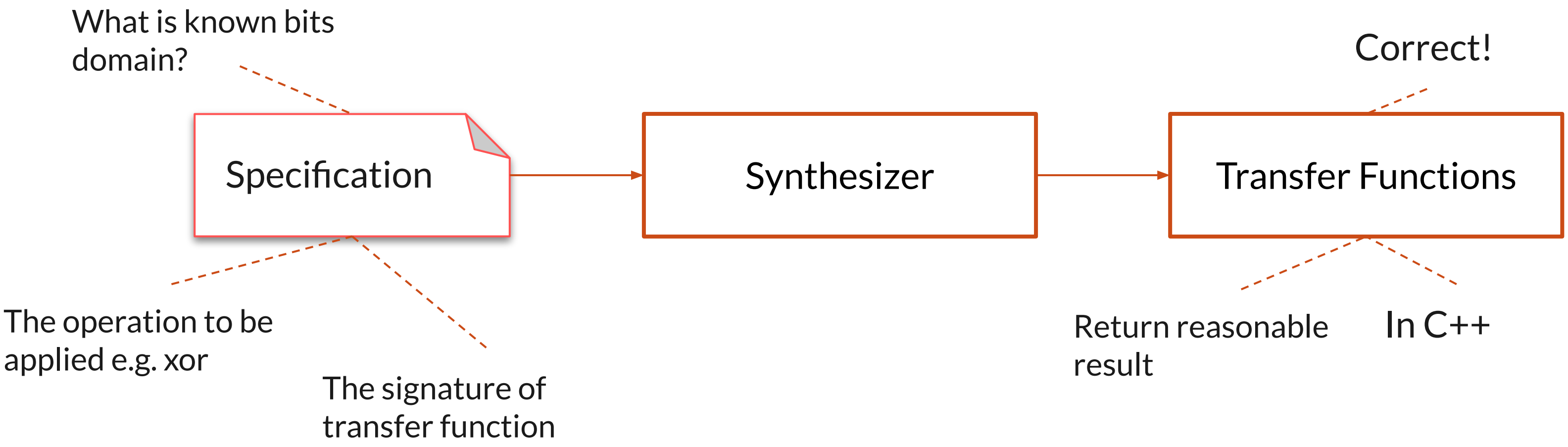
Known Bits is a domain provided in LLVM, there are other domains provided in LLVM and MLIR too.

They implement transfer functions for operations in the specification on all domains.

# The Need for a Synthesizer

# What can our synthesizer do?

In short, it reads a specification and generates *correct* and *not bad* transfer functions automatically.



# Generate more transfer functions

Provide transfer functions for unimplemented operations.

| Target Platform | Number of Intrinsics Implemented | Total Number of Intrinsics |
|---|---|---|
| X86 | 30 | 1713 |
| AArch64 | 5 | 1673 |
| RISCV | 2 | 737 |

LLVM only implements a small number of intrinsics compared to the total number of intrinsics

# Generate more transfer functions

Provide transfer functions for unimplemented operations.

| Dialect | Domain | Number of implemented transfer functions | Number of integer operations |
|---|---|---|---|
| Comb | Known Bits | 7 | 20 |
| Arith | Known Bits | N/A | N/A |

Other dialects can get benefit from the synthesizer such as \`wasmssa\`, \`emitc\`, \`index\`...

# Difficulties in reusing existing transfer functions

Semantics between two operations might be different.

```
// A constant has all bits known!
if (auto constant = dyn_cast<hw::ConstantOp>(op))
  return KnownBits::makeConstant(constant.getValue());
```

```
%res0 = comb.shl %arg0, 5 : i4 ——→  0 : i4

%res1 = llvm.shl %arg0, 5 : i4 ——→  poison
```

# Difficulties in reusing existing transfer functions

Utils

CMakeLists.txt

InferIntRangeCommon.cpp

Arith InferIntegerRange

Index InferIntegerRange

```
void arith::AndIOp::inferResultRanges(ArrayRef<ConstantIntRanges> argRanges,
                                      SetIntRangeFn setResultRange) {
  setResultRange(getResult(), inferAnd(argRanges));
void AndOp::inferResultRanges(ArrayRef<ConstantIntRanges> argRanges,
                              SetIntRangeFn setResultRange) {
  setResultRange(getResult(),
                 inferIndexOp(inferAnd, argRanges, CmpMode::Unsigned));
}
```

# Prove correctness of transfer functions.

Soundness: The analysis result covers all runtime values.

unsound

doesn't cover 0001

0000

```
define i4 @func(i4 %arg0) {
  ......
  %and1 = and i4 %arg0, %c1
}
```

# Prove correctness of transfer functions.

Soundness: The analysis result covers all runtime values.

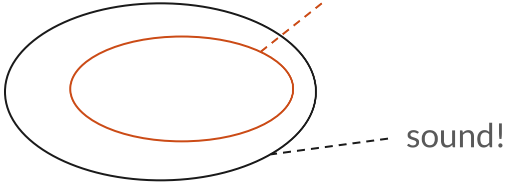
unsound

doesn’t cover 0001

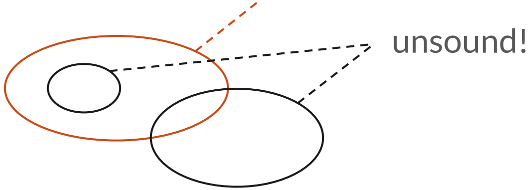
0000

```
define i4 @func(i4 %arg0) {
  ......
  %and1 = and i4 %arg0, %c1
}
```

The set of all runtime values



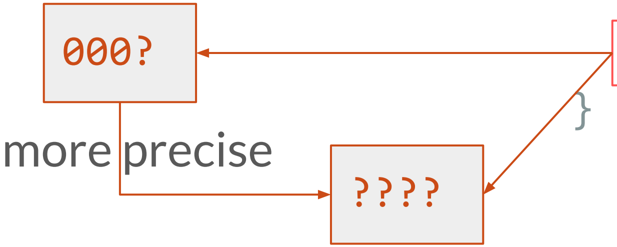

The set of all runtime values



# Prove correctness of transfer functions.

Precision: How many values that never occur at runtime included by the analysis result.

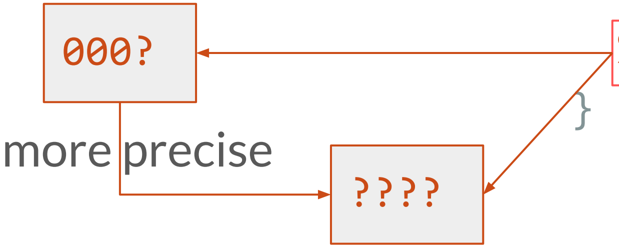
```
define i4 @func(i4 %arg0) {
  ......
  %and1 = and i4 %arg0, %c1
}
```

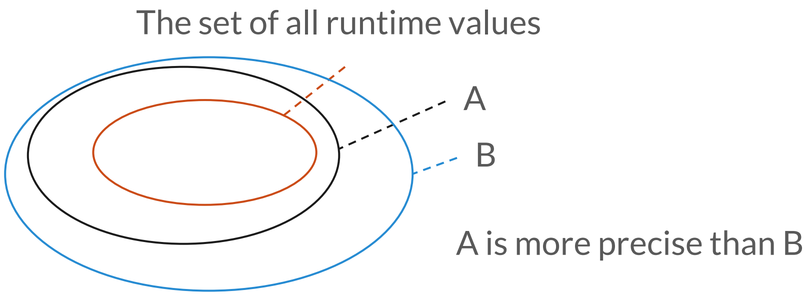


# Prove correctness of transfer functions.

Precision: How many values that never occur at runtime included by the analysis result.

```
define i4 @func(i4 %arg0) {
  ......
  %and1 = and i4 %arg0, %c1
}
```



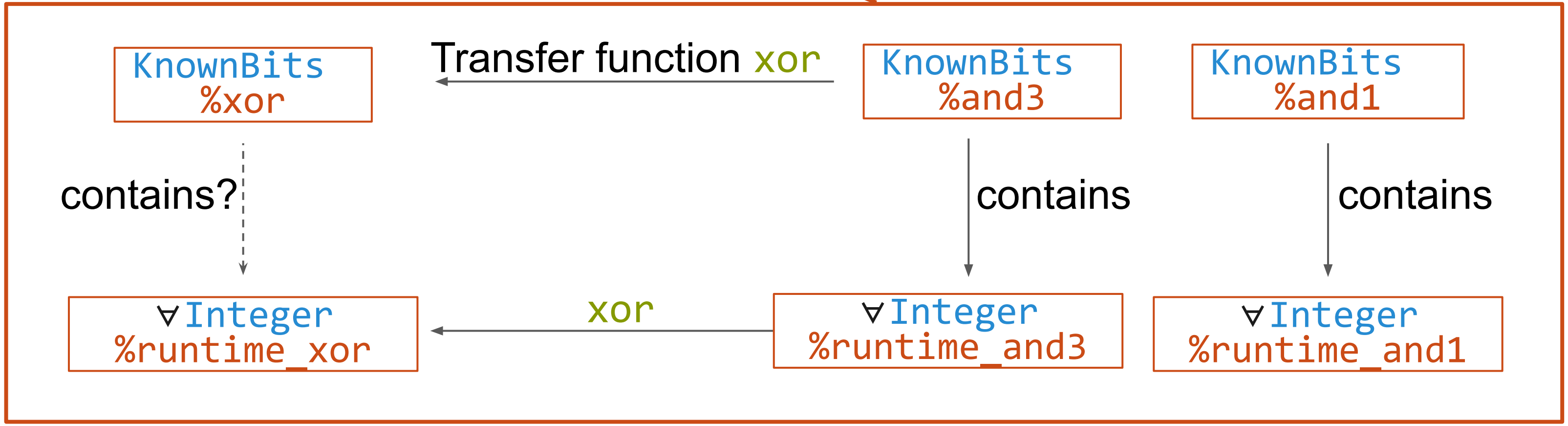
# How LLVM tests a transfer function

Let’s see how LLVM test soundness of transfer functions.

```
unsigned Bits = 4;
ForeachKnownBits(Bits, [&](const KnownBits &Known1) {
  ForeachKnownBits(Bits, [&](const KnownBits &Known2) {
    ForeachKnownBits(1, [&](const KnownBits &KnownCarry) {
      ......
      ForeachNumInKnownBits(Known1, [&](const APInt &N1) {
        ForeachNumInKnownBits(Known2, [&](const APInt &N2) {
          ForeachNumInKnownBits(KnownCarry, [&](const APInt &Carry) {
            ......
```

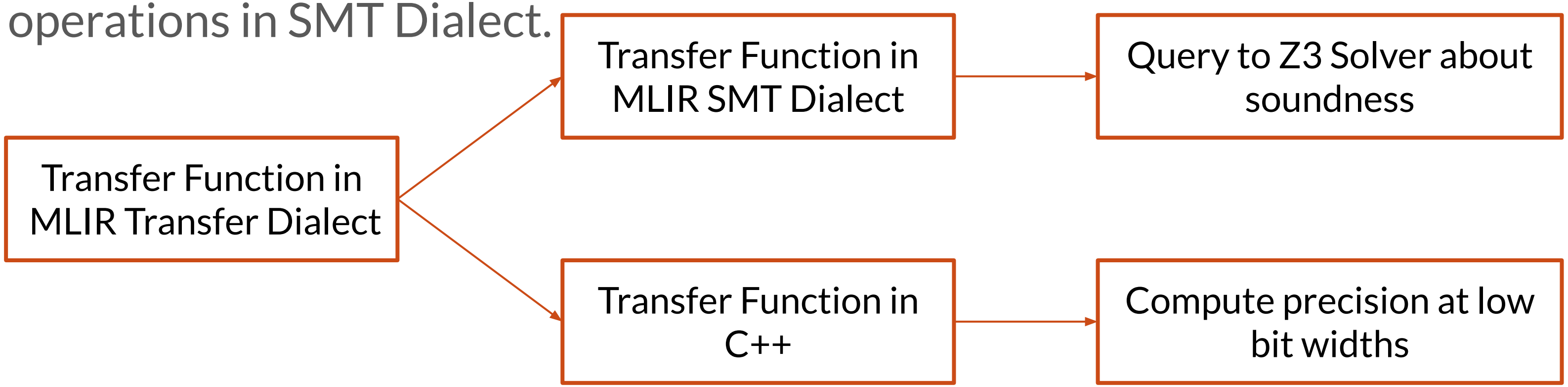
# How the synthesizer verifies a transfer function

With SMT semantics of the operation and transfer dialect, the synthesizer checks soundness as a SMT query:



# Our previous work

We defined a Transfer Dialect in MLIR and it encodes LLVM APInt operations in SMT Dialect.




Our Previous Work:

[First-Class Verification Dialects for MLIR]

[2023 LLVM Dev Mtg - An SMT dialect for assigning semantics to MLIR dialects]

# Summary

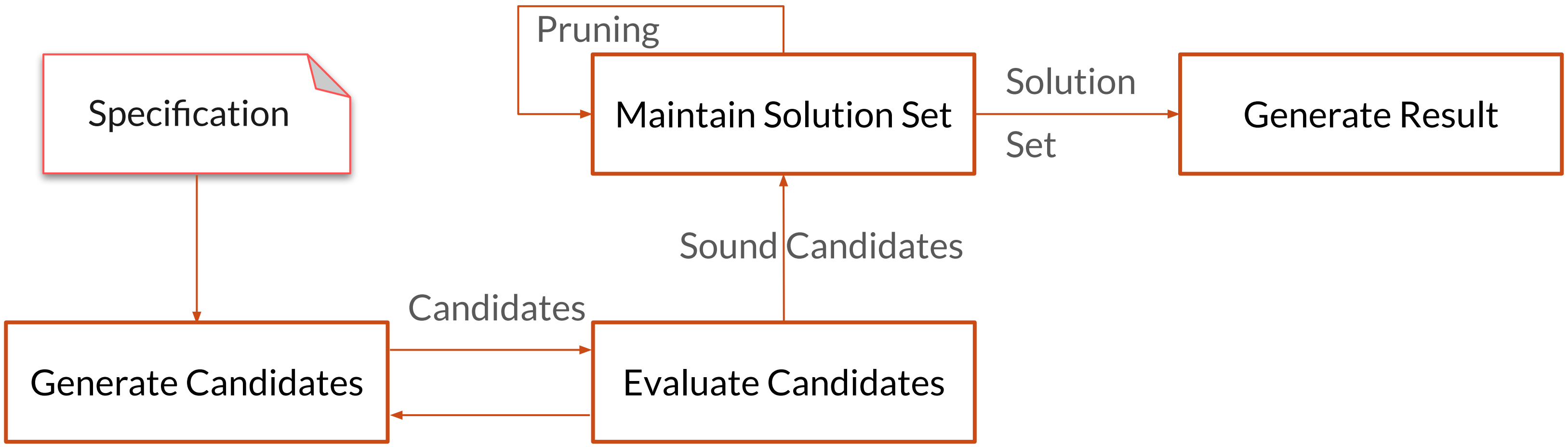
Our goal is not to beta LLVM transfer functions.

Our goal is to provide sound and precise transfer functions in C++ for new operations or domains.

# Design of the Synthesizer

# Big Picture

Our synthesizer comes with a synthesis loop



# Input Specification

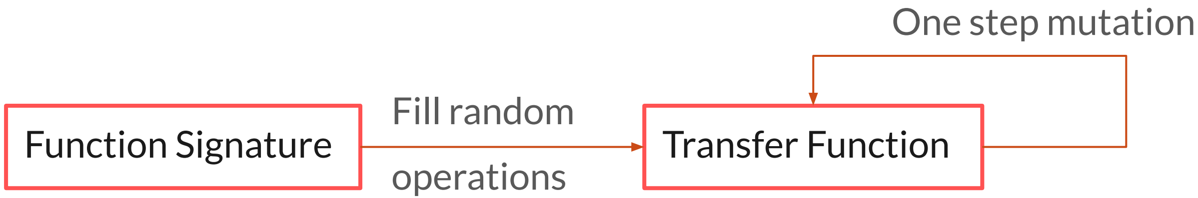
Our input specification is defined in MLIR.

Specification

- Definition of the domain
  - Domain members
  - Domain Constraints
  - Constructors (top, meet)
- Definition of the operation
  - Operation with SMT semantics
  - Operation constraints
- Transfer function signature

# Find candidates by stochastic search

Because the search space is extremely large, we adopt a stochastic search strategy to explore candidate transfer functions.



# Find candidates by stochastic search

Because the search space is extremely large, we adopt a stochastic search strategy to explore candidate transfer functions.

```
APInt autogen9 = autogen3 * autogen7;
```

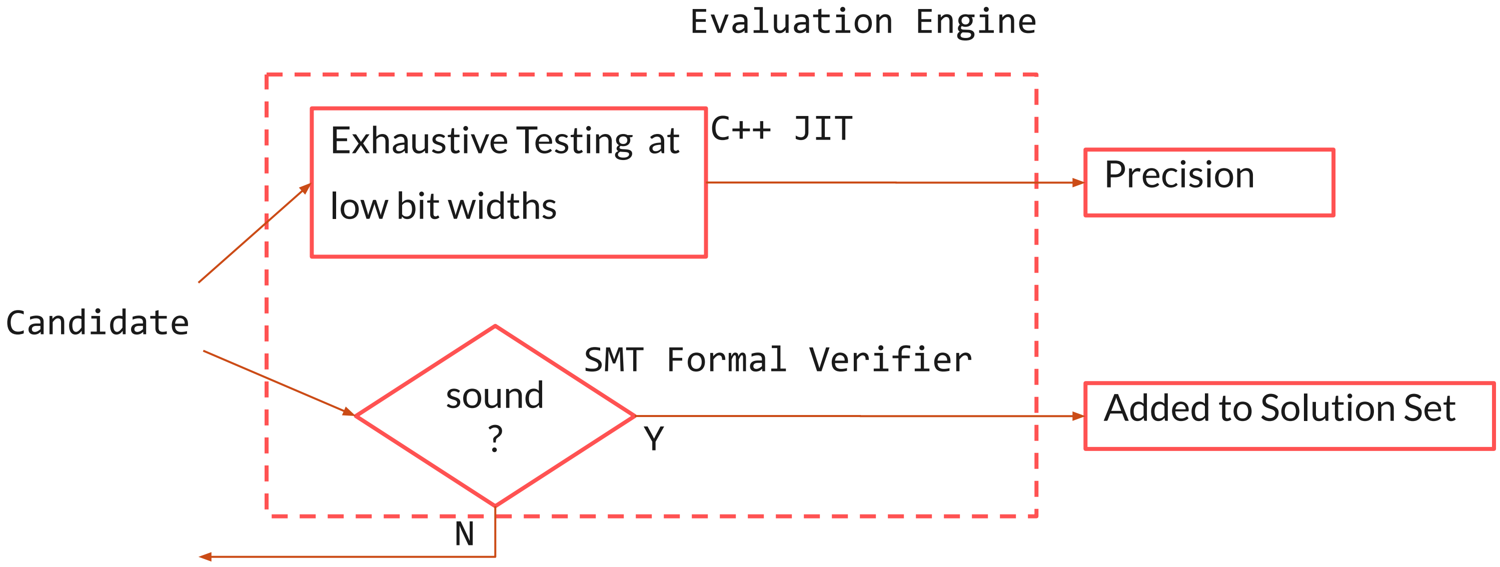
```
......
APInt autogen9 = autogen3 & autogen7;
APInt autogen10 = smax(autogen6, autogen4);
APInt autogen11 = umin(autogen1, autogen8);
......
```

Both might happen

```
APInt autogen10 = smax(autogen9, autogen4);
```

# Evaluate candidates

Evaluation Engine verifies the soundness of a transfer function and produce a precision score by the cost function.



# What candidates does the synthesizer keep?

It saves two kinds of candidates.

Solution set

Sound candidates and with high precision

# What candidates does the synthesizer keep?

It saves two kinds of candidates.

`Solution set`

Sound candidates and with high precision

Candidates only sound conditionally and with high precision

# What candidates does the synthesizer keep?

It saves two kinds of candidates.

Solution set

Sound candidates and with high precision

Candidates only sound conditionally and with high precision

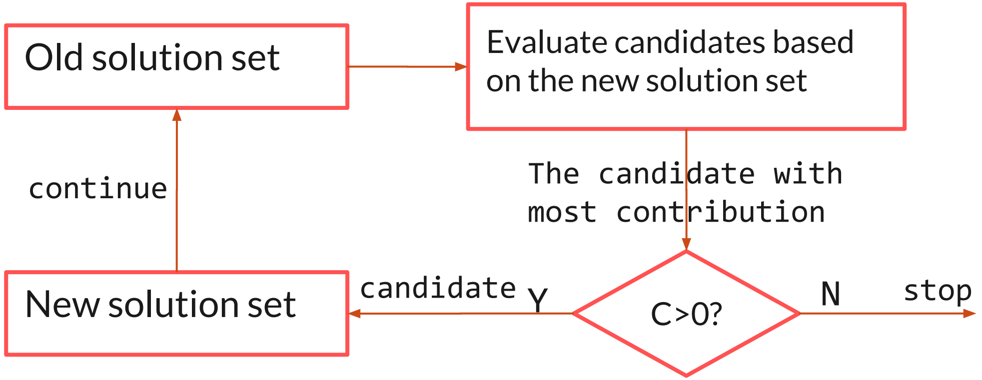
```
domain candidate(LHS, RHS);
```

```
if(candidate_guard(LHS, RHS))
  return candidate(LHS, RHS);
return top();
```

# Maintain Solution Set

The synthesizer maintains a dynamic pool of candidate transfer functions.

Less effective ones are pruned to prevent the solution set from growing excessively.



# Generate result

The synthesizer generates the final solution by the meet of all transfer functions.

```
Vec<2> xor_solution(Vec<2> autogen0,Vec<2> autogen1){
  Vec<2> autogen2 = xor_partial_solution_0(autogen0,autogen1);
  Vec<2> autogen3 = xor_partial_solution_1(autogen0,autogen1);
  Vec<2> autogen4 = meet(autogen2,autogen3);
  return autogen4;
}
```

# Generate result

The synthesizer generates the final solution by the meet of all transfer functions.

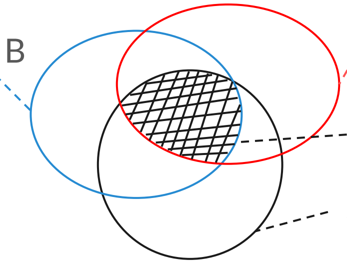
```
Vec<2> xor_solution(Vec<2> autogen0,Vec<2> autogen1){
  Vec<2> autogen2 = xor_partial_solution_0(autogen0,autogen1);
  Vec<2> autogen3 = xor_partial_solution_1(autogen0,autogen1);
  Vec<2> autogen4 = meet(autogen2,autogen3);
  return autogen4;
}
```

```
extern "C" Vec<2> sub_solution(Vec<2> autogen0,Vec<2> autogen1){
  Vec<2> autogen2 = sub_partial_solution_0(autogen0,autogen1);
  Vec<2> autogen3 = sub_partial_solution_1(autogen0,autogen1);
  Vec<2> autogen4 = sub_partial_solution_2(autogen0,autogen1);
  Vec<2> autogen5 = sub_partial_solution_3(autogen0,autogen1);
  Vec<2> autogen6 = sub_partial_solution_4(autogen0,autogen1);
  Vec<2> autogen7 = sub_partial_solution_5(autogen0,autogen1);
  Vec<2> autogen8 = sub_partial_solution_6(autogen0,autogen1);
  Vec<2> autogen9 = sub_partial_solution_7(autogen0,autogen1);
  Vec<2> autogen10 = sub_partial_solution_8(autogen0,autogen1);
  Vec<2> autogen11 = sub_partial_solution_9(autogen0,autogen1);
  Vec<2> autogen12 = sub_partial_solution_10(autogen0,autogen1);
  Vec<2> autogen13 = sub_partial_solution_11(autogen0,autogen1);
  Vec<2> autogen14 = sub_partial_solution_12(autogen0,autogen1);
  Vec<2> autogen15 = sub_partial_solution_13(autogen0,autogen1);
  Vec<2> autogen16 = sub_partial_solution_14(autogen0,autogen1);
  Vec<2> autogen17 = sub_partial_solution_15(autogen0,autogen1);
  Vec<2> autogen18 = meet(autogen2,autogen3);
  Vec<2> autogen19 = meet(autogen18,autogen4);
  Vec<2> autogen20 = meet(autogen19,autogen5);
  Vec<2> autogen21 = meet(autogen20,autogen6);
  Vec<2> autogen22 = meet(autogen21,autogen7);
  Vec<2> autogen23 = meet(autogen22,autogen8);
  Vec<2> autogen24 = meet(autogen23,autogen9);
  Vec<2> autogen25 = meet(autogen24,autogen10);
  Vec<2> autogen26 = meet(autogen25,autogen11);
  Vec<2> autogen27 = meet(autogen26,autogen12);
  Vec<2> autogen28 = meet(autogen27,autogen13);
  Vec<2> autogen29 = meet(autogen28,autogen14);
  Vec<2> autogen30 = meet(autogen29,autogen15);
  Vec<2> autogen31 = meet(autogen30,autogen16);
  Vec<2> autogen32 = meet(autogen31,autogen17);
  return autogen32;
}
```

# Generate result

The synthesizer generates the final solution by the meet of all transfer functions.

```
Vec<2> xor_solution(Vec<2> autogen0,Vec<2> autogen1){
  Vec<2> autogen2 = xor_partial_solution_0(autogen0,autogen1);
  Vec<2> autogen3 = xor_partial_solution_1(autogen0,autogen1);
  Vec<2> autogen4 = meet(autogen2,autogen3);
  return autogen4;
}
```




```
extern "C" Vec<2> sub_solution(Vec<2> autogen0,Vec<2> autogen1){
  Vec<2> autogen2 = sub_partial_solution_0(autogen0,autogen1);
  Vec<2> autogen3 = sub_partial_solution_1(autogen0,autogen1);
  Vec<2> autogen4 = sub_partial_solution_2(autogen0,autogen1);
  Vec<2> autogen5 = sub_partial_solution_3(autogen0,autogen1);
  Vec<2> autogen6 = sub_partial_solution_4(autogen0,autogen1);
  Vec<2> autogen7 = sub_partial_solution_5(autogen0,autogen1);
  Vec<2> autogen8 = sub_partial_solution_6(autogen0,autogen1);
  Vec<2> autogen9 = sub_partial_solution_7(autogen0,autogen1);
  Vec<2> autogen10 = sub_partial_solution_8(autogen0,autogen1);
  Vec<2> autogen11 = sub_partial_solution_9(autogen0,autogen1);
  Vec<2> autogen12 = sub_partial_solution_10(autogen0,autogen1);
  Vec<2> autogen13 = sub_partial_solution_11(autogen0,autogen1);
  Vec<2> autogen14 = sub_partial_solution_12(autogen0,autogen1);
  Vec<2> autogen15 = sub_partial_solution_13(autogen0,autogen1);
  Vec<2> autogen16 = sub_partial_solution_14(autogen0,autogen1);
  Vec<2> autogen17 = sub_partial_solution_15(autogen0,autogen1);
  Vec<2> autogen18 = meet(autogen2,autogen3);
  Vec<2> autogen19 = meet(autogen18,autogen4);
  Vec<2> autogen20 = meet(autogen19,autogen5);
  Vec<2> autogen21 = meet(autogen20,autogen6);
  Vec<2> autogen22 = meet(autogen21,autogen7);
  Vec<2> autogen23 = meet(autogen22,autogen8);
  Vec<2> autogen24 = meet(autogen23,autogen9);
  Vec<2> autogen25 = meet(autogen24,autogen10);
  Vec<2> autogen26 = meet(autogen25,autogen11);
  Vec<2> autogen27 = meet(autogen26,autogen12);
  Vec<2> autogen28 = meet(autogen27,autogen13);
  Vec<2> autogen29 = meet(autogen28,autogen14);
  Vec<2> autogen30 = meet(autogen29,autogen15);
  Vec<2> autogen31 = meet(autogen30,autogen16);
  Vec<2> autogen32 = meet(autogen31,autogen17);
  return autogen32;
}
```

# Experimental Results

# Synthesis Result

Here is the partial result synthesizing 39 operations on KnownBits Domain.
Transfer functions are tested on random 8-bit inputs.

| Operation | Ours | LLVM |
|---|---|---|
| And/Or/Xor | 100% | 100% |
| Modu | 59% | 52.7% |
| UShlSat | 96.6% | N/A |
| Mul | 60.6% | 73.2% |

The results are sensitive to random factors, leading to variability between runs.

| Operation | Ours | LLVM |
|---|---|---|
| AvgFloorS | 39.3% | 100% |
| Shl | 56.9% | 96.5% |
| Abds | 60.1% | 100% |
| Add | 58.70% | 100% |
| Sub | 60.6% | 100% |

# Evaluation on SPEC 2017 CPU

We also test transfer functions on SPEC 2017 CPU benchmarks. The table below compares Known Bits found by LLVM and ours.

| Project | perlbench | gcc | mcf | omnetpp | xalancbmk | x264 | deepsjeng | leela | xz |
|---|---|---|---|---|---|---|---|---|---|
| KLOC | 362 | 1,304 | 3 | 134 | 520 | 96 | 10 | 21 | 33 |
| LLVM | 1,356,555 | 4,272,154 | 910 | 62,251 | 475,736 | 247,344 | 15,578 | 25,207 | 76,907 |
| Ours | 1,305,537 | 4,195,918 | 910 | 62,102 | 442,838 | 218,171 | 14,780 | 19,353 | 72,090 |
| Precision Loss | 3.76% | **1.78%** | 0.00% | 0.24% | 6.29% | 11.79% | 5.12% | 23.22% | 6.26% |

# Summary

Our goal is not to beat LLVM, but generate correct and precise transfer functions when it comes to a new dialect or new domain.

By giving the specification and the operation with SMT semantics, the synthesizer can generate usable and sound transfer functions used in dataflow analysis.

# Thanks for listening!

Yuyou Fan
Fifth-year PhD student
yuyou.fan@utah.edu
Looking for full-time job around May 2026
<- My resume.