Nice to Meet You: Synthesizing Practical MLIR Abstract Transformers

XUANYU PENG*, University of California San Diego, USA
DOMINIC KENNEDY*, University of Utah, USA
YUYOU FAN, University of Utah, USA
BEN GREENMAN, University of Utah, USA
JOHN REGEHR, University of Utah, USA
LORIS D'ANTONI, University of California San Diego, USA

Static analyses play a fundamental role during compilation: they discover facts that are true in all executions of the code being compiled, and then these facts are used to justify optimizations and diagnostics. Each static analysis is based on a collection of *abstract transformers* that provide abstract semantics for the concrete instructions that make up a program. It can be challenging to implement abstract transformers that are sound, precise, and efficient—and in fact both LLVM and GCC have suffered from miscompilations caused by unsound abstract transformers. Moreover, even after more than 20 years of development, LLVM lacks abstract transformers for hundreds of instructions in its intermediate representation (IR).

We developed `NiceToMeetYou`: a program synthesis framework for abstract transformers that are aimed at the kinds of non-relational integer abstract domains that are heavily used by today's production compilers. It exploits a simple but novel technique for breaking the synthesis problem into parts: each of our transformers is the meet of a collection of simpler, sound transformers that are synthesized such that each new piece fills a gap in the precision of the final transformer. Our design point is bulk automation: no sketches are required. Transformers are verified by lowering to a previously-created SMT dialect of MLIR. Each of our synthesized transformers is provably sound and some (17 %) are more precise than those provided by LLVM.

CCS Concepts: • **Software and its engineering → Compilers**; **Automated static analysis**; **Automatic programming**; • **Theory of computation → Abstraction**; **Program analysis**.

Additional Key Words and Phrases: Abstract Interpretation, Program Synthesis, LLVM

ACM Reference Format:
Xuanyu Peng, Dominic Kennedy, Yuyou Fan, Ben Greenman, John Regehr, and Loris D'Antoni. 2026. Nice to Meet You: Synthesizing Practical MLIR Abstract Transformers. *Proc. ACM Program. Lang.* 10, POPL, Article 80 (January 2026), 32 pages. https://doi.org/10.1145/3776722

1 Introduction

A modern, highly optimizing compiler runs numerous dataflow analyses on the code that is being compiled; the results of the analyses are used to justify optimizations and diagnostics. For example, LLVM relies heavily on a "`KnownBits`" analysis that attempts to prove that individual bits of SSA values are either zero or one in every execution of the program being compiled.

*Equal contribution

Authors' Contact Information: Xuanyu Peng, University of California San Diego, USA, xup002@ucsd.edu; Dominic Kennedy, University of Utah, USA, dominicmkennedy@gmail.com; Yuyou Fan, University of Utah, USA, yuyou.fan@utah.edu; Ben Greenman, University of Utah, USA, benjamin.l.greenman@gmail.com; John Regehr, University of Utah, USA, regehr@cs.utah.edu; Loris D'Antoni, University of California San Diego, USA, ldantoni@ucsd.edu.

ACM 2475-1421/2026/1-ART80
https://doi.org/10.1145/3776722

Empirically, the process of engineering a dataflow-driven compiler works as follows. First, engineers recognize the need for dataflow results and implement the basic analysis structure within the compiler, which is initially highly imprecise because not enough, and not-precise-enough, *abstract transformers* have been written. Then, optimizations and diagnostics driven by analysis results are added, typically alongside improvements to analysis precision that are necessary to make the compiler operate robustly. In a compiler like LLVM, where the IR (intermediate representation) instruction set is large (over 400 target-independent instructions and intrinsics), this process takes an enormous amount of time and energy. Across 17 different LLVM backends, only four have any abstract transformers at all for LLVM instructions representing target-specific intrinsics, and even those four have poor coverage: 30 out of 1713 intrinsics for x86-64, 2 out of 726 for RISC-V, 2 out of 1286 for AMD GPUs, and 5 out of 1673 for AArch64. Operations that lack abstract transformers must be analyzed conservatively: they return *top*, the unknown value. This low coverage can lead to unpredictable compilation effects where, for example, developers who substitute intrinsics for portable code while chasing performance can see degraded dataflow-driven optimizations in nearby code because the portable code could be analyzed but the intrinsics cannot. Moreover, bugs in unverified dataflow analyses have led to miscompilation errors in both GCC and LLVM [18, 26].

Our work attacks the problem of synthesizing abstract transformers from concrete instruction semantics that are formally specified using an MLIR dialect [10]. Our goal is to develop technologies that can rapidly provide compiler developers with reasonable initial implementations. We validate our prototype by synthesizing transformers for three non-relational, compiler-friendly abstract domains (`KnownBits`, signed and unsigned `ConstantRange`) for 39 instructions that are present both in LLVM and in MLIR's Arith dialect. These are formally verified to be sound, and in some cases are more precise than those that are part of LLVM's implementation, which has been tweaked for precision by numerous compiler developers over the last 20 years.

Given a concrete operation f and an abstract domain (e.g., `KnownBits`), our goal is to synthesize a corresponding abstract transformer $f^\#$. To be sound on a given abstract input, $f^\#$ must return an abstract value that over-approximates the set of all possible outputs produced by applying f to any concrete inputs described by its abstract inputs. The smaller this over-approximation, the more precise the abstract transformer. Formally, when the power set of concrete values $\mathcal{P}(C)$ is related to the set of abstract values (i.e., abstract domain) $\mathcal{A}$ by a Galois connection $\mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, there is a *best abstract transformer* $\widehat{f^\#}$ that is defined as $\widehat{f^\#} = \alpha \circ \tilde{f} \circ \gamma$, where $\tilde{f}$ runs f on a set of concrete values and produces their corresponding concrete outputs [5]. However, this transformer definition does not directly lead to a usable implementation: it requires taking the meet of a set of abstract values whose size is exponential in the bitwidth of the concrete values being analyzed.

Since it does not seem generally practical to synthesize best abstract transformers, previous research efforts have focused on finding efficient approximations of them. Some approaches, such as Scherpelz et al. [30] and Elder et al. [9], have targeted specific abstract domains. Kalita et al. [15] provide a synthesis framework that is applicable to arbitrary domains, but it depends on user-provided program sketches (see Section 6.6 and Appendix A).

Our work addresses the following research questions in the context of finite, Galois-connection-based abstract domains (c.f. [5, 7]):

- **Practicality and Generality:** Using existing formally specified concrete instruction semantics, can we automatically synthesize abstract transformers for multiple domains that are used in real-world compilers? In particular, can we generate functions that compiler developers can adopt—i.e., ones that are free of external dependencies, performant enough for production use, and sensitive to IR-level subtleties such as undefined behavior?

- **Soundness and Precision:** How precisely can our synthesized transformers approximate the ideal transformer, $\widehat{f^{\#}}$, while being provably sound?
- **Automation:** Can our synthesis procedure navigate the search space without meaningful help from users? In other words, can we succeed without requiring sketches?

We treat synthesis as an optimization problem, where the objective is to find a sound transformer $f^{\#}$ that minimizes the user-given norm $\left\|f^{\#}\right\|$ that measures the imprecision for every possible pair of abstract inputs (note that such a function is easy to implement for given abstract domains). Since the functions that we wish to synthesize are (empirically) out of direct reach for enumeration or CEGIS, and since we do not wish to rely on user-provided sketches, we use stochastic search techniques inspired by Stoke (Schkufza et al. [31]), where candidate transformers evolve through a sequence of random modifications guided by the cost function induced by our objective.

We have observed two implementation patterns in code produced by compiler developers writing highly precise abstract transformers for GCC and LLVM. Both of these ended up leading to key aspects of our approach:

Pattern 1: Splitting the input space. Practical abstract transformers often gain precision by making a case split to separately handle different parts of the input space. For example, LLVM's transformer for bitvector truncation on integer ranges[1] begins with special cases for $\top$ and $\bot$ and then subsequently splits on whether the incoming integer range wraps around the `UINT_MAX` / 0 boundary. We observed that the logic at the finest granularity was often reasonably simple, but that the overall transfer functions that we wanted to synthesize appeared to be significantly more complicated than anything we could reliably generate without a sketch.

The insight that allowed us to make progress here was that the meet of a collection of sound abstract transformers is still sound. Thus, we can synthesize an abstract transformer in parts, and then assemble the parts at the end: $\mathcal{F}_{\sqcap} = f_1^{\#} \sqcap \cdots \sqcap f_n^{\#}$. If we simply synthesized a pile of transformers and glued them together, they would be likely to all cover similar parts of the input space (because, e.g., some parts of the input space are easier to cover than others). We discourage this behavior by *dynamically adapting* our fitness function: each new abstract transformer is rated by its precision on parts of the input space not covered by transformers that have previously been synthesized. This adaptive strategy steers the synthesis process away from inputs that are already handled precisely and toward those that require new cases. This decomposition not only makes it easier to synthesize precise transformers with high precision, but also allows users to control the number and size of components in the meet—providing an easy knob for tuning efficiency vs. precision.

Pattern 2: Separate transformers for separate jobs. Beyond splitting up the input space, we noticed that realistic transfer functions gain precision by exploiting information that is present in the IR. For example, the LLVM compiler's abstract transformer for integer multiply begins with a large special case for the "nsw" or "no unsigned wrap" flag[2]—this flag can be exploited to increase analysis precision, because signed overflows become undefined. Then, immediately inside the nsw case, the code splits again to handle the case of computing the square of a value, which again affords additional precision. Our observation is that it is unnecessary and even undesirable to entangle the implementation of the $y * y$ case with the $x * y$ case and the $x *_{\text{nsw}} y$ case: these are actually distinct transfer functions that need to be handled separately during testing or formal verification. They happen to be handled by overlapping code only because LLVM's developers decomposed their code that way. Program synthesis, on the other hand, changes the basic software engineering economics,

[1] https://github.com/llvm/llvm-project/blob/release/20.x/llvm/lib/IR/ConstantRange.cpp#L864-L915
[2] https://github.com/llvm/llvm-project/blob/release/20.x/llvm/lib/Analysis/ValueTracking.cpp#L383-L437

making it cheap to create a large number of abstract transformers, including a specialized version for every IR-level condition that can be exploited to increase precision.

Implementation in MLIR. We get formal semantics for instructions from previous work on the SMT dialect for MLIR [10], which supports lowering operations to both LLVM IR and SMT formulae. The LLVM IR lowering, and subsequent JIT compilation, enables fast evaluation of candidate transformers during synthesis, and allow us to leverage LLVM's powerful optimizers to improve the performance of the final synthesized transformers. The SMT lowering allows us to verify the soundness and precision of the synthesized transformers, although in practice we generally measure precision via testing rather than solving because we are interested in giving our stochastic optimizer a hill to climb, rather than a binary result (a model counting solver would be an alternative way to get a hill to climb, but in our experience they do not scale to jobs like this one).

Evaluation. We evaluate our approach by synthesizing abstract transformers for the abstract domains and operations used in the LLVM IR. The results show that `NiceToMeetYou` complements the precision of LLVM transformers (when measured on 8-bit and 64-bit integers) of 7/47 transformers in the `KnownBits` domain and 19/47 in `ConstantRange`. With the addition of a handwritten reduced-product operator for combining synthesized transformers across different abstract domains, our synthesized transformers exceed LLVM's precision on 22/47 operators.

Contributions. Our work makes the following contributions:

- We propose a framework for synthesizing abstract transformers that leverages existing formal semantics for instructions, and is not limited to specific abstract domains and does not require program templates (Section 2).
- We design an algorithm that incrementally synthesizes the meet of multiple abstract transformers (Section 3), which enables an MCMC-based search procedure that can discover individual smaller transformers that can be added to the meet (Section 4).
- We implement the algorithm in `NiceToMeetYou`, a tool that effectively balances MCMC-based exploration with SMT-based verification. We apply `NiceToMeetYou` on the bread-and-butter abstract domains from LLVM, namely `KnownBits` and `ConstantRange` (Section 5).
- We conduct an evaluation showing how `NiceToMeetYou` can synthesize abstract transformers for real LLVM operators. Our transformers are often complementary to LLVM's, and in some cases exceed the precision of LLVM's hand-tuned transformers (Section 6).

2 Problem Definition and Overview of the Approach

In this section, we define the problem addressed by our framework using a toy example (Section 2.1). We then provide an excerpt of a real transformer from MLIR, `urem` over known bits, to illustrate how our problem setting and approach leads to practical gains (Section 2.2).

2.1 The Transformer Synthesis Problem

Throughout this section, we use a running example in which the goal is to synthesize transformers for the integer maximum function $f(x, y) = \max(x, y)$ over the domain of intervals.

The user of our framework needs to provide a definition of a concrete domain, an abstract domain over which they are trying to synthesize abstract transformers, and a language of programming constructs the synthesizer can use to synthesize the abstract transformers.

Concrete Domain and Concrete Transformers. A transformer depends on a concrete domain C, and a concrete transformer $f : C^k \to C$. In our example, we let C be integers of bitwidth up to a

certain number (e.g., 32), which can be represented by the `APInt` class in LLVM and MLIR.[3] The concrete transformer in our example is $\max : C^2 \to C := \lambda x \lambda y.\texttt{ite}(x > y, x, y)$.

Abstract Domain. A lattice $\mathcal{A}$ serves as an abstract domain. In this subsection, $\mathcal{A}$ is the interval domain. Each abstract value a is a pair of integers representing an interval $[a.l, a.r]$.[4]

In particular, the user needs to provide the implementations of the following components:

- Concretization function $\gamma : \mathcal{A} \to 2^C$. In our example, $\gamma(a) = \{a.l, a.l + 1 \cdots, a.r\}$.
- Meet function $\sqcap : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. In our example, $\sqcap$ is the intersection of two intervals, i.e., $a \sqcap b = \textbf{if } \max(a.l, b.l) > \min(a.r, b.r) \textbf{ then } \bot \textbf{ else } [\max(a.l, b.l), \min(a.r, b.r)]$.
- Join function $\sqcup : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. In our example, $\sqcup$ is the union of two intervals, i.e., $a \sqcup b = [\min(a.l, b.l), \max(a.r, b.r)]$.
- Single value abstraction function $\beta : C \to \mathcal{A}$. In our example, $\beta(x) = [x, x]$.

The general abstraction function $\alpha : 2^C \to \mathcal{A}$ that maps a set of concrete elements to their abstract one is defined as $\alpha(C) := \bigsqcup_{x \in C} \beta(x)$. The partial order $\sqsubseteq \in \mathcal{A} \times \mathcal{A}$ that relates abstract elements is defined as follows: $a_1 \sqsubseteq a_2 \iff a_1 \sqcap a_2 = a_1$.

Language. In our setting, a domain-specific language (DSL) $\mathcal{L}$ is a context-free grammar of the form $E := a \mid c \mid op(E_1, \cdots, E_k)$, where a ranges over abstract input variables, c denotes constants drawn from a fixed set C, and op is an operator drawn from a predefined set of function symbols $\mathcal{L}_{op}$ supported by the DSL. In our example, $\mathcal{L}_{op} \cup C$ consists of the set $\{+, -, \&, |, \min, \max, [\cdot, \cdot], \cdot.l, \cdot.r, \texttt{Zero}, \texttt{AllOnes}\}$, where $[\cdot, \cdot]$ denotes interval construction, and $\cdot.l$ and $\cdot.r$ access the left and right endpoints of an interval, respectively. The `Zero` and `AllOnes` constants return all zeros and all ones at the given bitwidth. For instance, the function $f^\#(a, b) := [\min(a.l, \texttt{AllOnes}),\ \max(a.r, \texttt{Zero})]$ is a valid program in the language $\mathcal{L}$.

Soundness and Precision. Our goal is to synthesize a set of *sound* (i.e., valid overapproximations of the function behavior) and *precise* (tight) abstract transformers $\mathcal{F} = \{f_1^\#, f_2^\#, \cdots, f_n^\#\}$ expressed using DSL operators. Because the user of the framework provides the meet operation for the abstract domain as input, we can then compute the meet of all such transformers $\mathcal{F}_\sqcap$ as follows:

Definition 2.1 (Meet of Transformers)**.** *Given two abstract transformers $f_1^\#, f_2^\# : \mathcal{A}^k \to \mathcal{A}$, their* meet *is defined as the transformer $f_1^\# \sqcap f_2^\# : \mathcal{A}^k \to \mathcal{A}$ such that, for all $a_1, \ldots, a_k \in \mathcal{A}$: $(f_1^\# \sqcap f_2^\#)(a_1, \ldots, a_k) = f_1^\#(a_1, \ldots, a_k) \sqcap f_2^\#(a_1, \ldots, a_k)$. We define the meet of a set of transformers $\mathcal{F} \subseteq (\mathcal{A}^k \to \mathcal{A})$ as $\mathcal{F}_\sqcap = \bigsqcap_{f^\# \in \mathcal{F}} f^\#$.*

Intuitively, the meet of two transformers is their pointwise meet in the abstract domain, which is both sound and represents the most precise possible combination of the synthesized transformers.[5]

One can check that a transformer is sound using the concretization function γ as follows:

[3]Note that the set of integers of bitwidths up to w is **not** equivalent to the set of integers in the range $[0, 2^w - 1]$. Instead, it is the union of sets of bitvectors with lengths from 1 to w. In other words, $C = C_1 \cup C_2 \cup \cdots \cup C_w$, where each C_i is a concrete subdomain representing integers with bitwidth w, and these subdomains are disjoint. We always assume the concrete transformer is only defined on inputs within the same subdomain C_i.

[4]As we only consider bounded integers of specific bitwidths, the abstract domain is restricted to closed intervals and is therefore finite. Readers may also notice that $a.l$ and $a.r$ should have the same bitwidth as the concrete values included by that interval. Strictly speaking, given the full concrete domain $C = C_1 \cup C_2 \cup \cdots \cup C_w$, each concrete subdomain C_i has its own abstract domain $\mathcal{A}_i$, each equipped with its own top and bottom elements. The full abstract domain $\mathcal{A}$ is likewise the disjoint union of all abstract subdomains $\mathcal{A}_i$. The top element of $\mathcal{A}$ will only be reached when joining two abstract values from two different $\mathcal{A}_i$, and the bottom element will only be reached when meeting two abstract values from two different $\mathcal{A}_i$. However, in practice, the abstract transformers only need to be defined on inputs from the same $\mathcal{A}_i$.

[5]Ideally, the set of abstract transformers would be a singleton $\{\widehat{f^\#}\}$. However, the theoretical best transformer $\widehat{f^\#}$ might not be expressible in the DSL, or may be too complex and thus computationally expensive for static analysis.

Definition 2.2 (Soundness of Transformers). *A transformer $f^\# : \mathcal{A}^k \to \mathcal{A}$ is* sound *with respect to a concrete function $f : C^k \to C$, denoted by* $\mathsf{sound}(f^\#)$*, if it is sound on all abstract inputs, i.e.,*

$$\mathsf{sound}(f^\#) \stackrel{\text{def}}{=} \forall a_1, \ldots, a_k \in \mathcal{A}. \quad \{f(c_1, \ldots, c_k) \mid c_i \in \gamma(a_i)\} \subseteq \gamma(f^\#(a_1, \ldots, a_k)).$$

For precision, there is no easy way to check that a transformer is the most precise possible among those expressible in a given language (a problem as hard as checking unrealizability in program synthesis [14]). We address this practical problem by introducing a precision measure; namely, we ask the user to provide a norm function $\|\cdot\| : (\mathcal{A}^k \to \mathcal{A}) \to \mathbb{N}$ that quantifies the imprecision of a transformer. The objective of our problem is to minimize the norm of the synthesized transformer $\mathcal{F}_\sqcap$. While a norm function over transformers can be hard to define, in our implementation, we ask the user to provide a size function $|\cdot| : \mathcal{A} \to \mathbb{N}$ on abstract values, and derive the norm via the size function as $\left\|f^\#\right\| = \sum_{a \in \mathcal{A}} \left|f^\#(a)\right|$. In practice, the size $|a|$ can be set as any function that is monotonic with respect to the actual size of the concretization set $\gamma(a)$. For the interval domain, we define the size as the $\log_2$ of its length, e.g., $|[0, 7]| = \log_2(8) = 3$.

Because computing the sum of norms over the entire abstract domain can be expensive, we will often approximate the size by evaluating it over a representative subset $\mathcal{A}' \subseteq \mathcal{A}$. We use $\left\|f^\#\right\|_{\mathcal{A}'} = \sum_{a \in \mathcal{A}'} \left|f^\#(a)\right|$ to denote the approximate norm over the subdomain $\mathcal{A}'$. In our example, the subset $\mathcal{A}'$ can be all abstract values represented by integers up to a smaller bitwidth, or some sampled abstract values represented by integers at a large bitwidth.

Norms vs. Metrics. Initially, we formalized precision using distance metrics rather than norms. This approach turned out to be problematic because one would need to compute the distance between a synthesized transformer $f^\sharp_{synth}$ and the theoretically best transformer $\widehat{f^\#}$ by enumerating all abstract values and their concretizations, and computing the join of $\beta(f(c))$ for each concrete value. This is feasible only at small bitwidths. That being said, prior works use metrics [2, 3, 20] and may provide a way to explore non-integer or infinite abstract domains in the future.

2.1.1 *Problem Definition.*

We are now ready to define the problem solved in this paper:

Definition 2.3 (Transformer Synthesis Problem). *Given a concrete transformer $f : C^k \to C$, an abstract domain $(\mathcal{A}, \top, \gamma, \sqcap, \sqcup, \beta)$, a norm function $\|\cdot\| : (\mathcal{A}^k \to \mathcal{A}) \to \mathbb{N}$, and a DSL $\mathcal{L}$, the transformer synthesis problem is to find a set of transformers $\mathcal{F} = \{f_1^\#, f_2^\#, \cdots, f_n^\#\}$ in $\mathcal{L}$ such that*

- *Their meet $\mathcal{F}_\sqcap$ is sound:* $\mathsf{sound}(\mathcal{F}_\sqcap)$.
- *The norm of $\mathcal{F}_\sqcap$ is minimal, i.e., there is no sound set of transformers $\mathcal{G}$ such that $\|\mathcal{G}_\sqcap\| < \|\mathcal{F}_\sqcap\|$.*
- *No $f_i^\# \in \mathcal{F}$ is redundant: $\forall f_i^\# \in \mathcal{F}, \exists \vec{a} \in \mathcal{A}^k, \left(\bigsqcap_{f^\# \in \mathcal{F} \setminus \{f_i^\#\}} f^\#(\vec{a})\right) \not\sqsubseteq f_i^\#(\vec{a})$.*

The first requirement, that the meet of the n transformers is sound, is satisfied if all n transformers are sound. The second requirement asks that the final meet be as precise as possible, rather than requiring each individual transformer to be. The last requirement ensures every transformer in the final solution contributes to improving the precision.[6]

Returning to our running example $f(x, y) = \max(x, y)$, Figure 1 shows the problem inputs and one possible set of output transformers. Smart readers may observe that the best transformer has a succinct representation: $f^\#(a, b) = [\max(a.l, b.l), \max(a.r, b.r)]$. However, the meet of the four output transformers in Figure 1 is also equivalent to this best transformer—since the maximum of their left endpoints equals $\max(a.l, b.l)$ and the minimum of their right endpoints equals $\max(a.r, b.r)$.

[6] Kalita et al. [15] propose a similar definition for the problem of synthesizing *one* most precise abstract transformer in a given DSL. In their setting, precision is defined in absolute terms with respect to the $\sqsubseteq$ partial order. In our setting, precision is defined in terms of a size function over the abstract domain. Furthermore, our definition extends to the set of synthesizing multiple incomparable transformers. We further discuss these implications in Section 7.

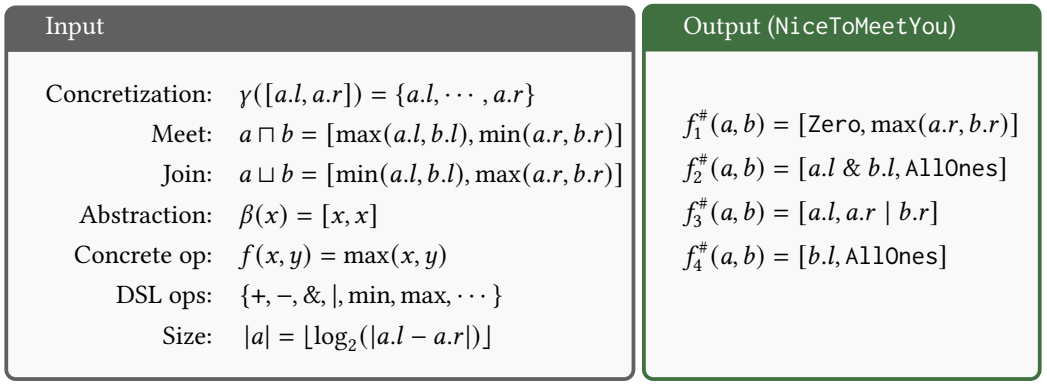

Fig. 1. Input and output for the transformer synthesis problem on a toy example.

```
KnownBits urem(KnownBits L, KnownBits R) {
  // Part 1:
  unsigned RTrailingZeros = R.Zero.countTrailingZero();
  APInt Mask = setLowBits(0, RTrailingZero);
  APInt knownZero = L.Zero & Mask;
  APInt knownOne = L.One & Mask;
  // Part 2:
  if (R.isConstant() && R.getConstant().isPowerOf2()) { ... }
  // Part 3:
  unsigned Leaders = max(L.Zero.countLeadingZero(), R.Zero.countLeadingZero());
  knownZero = setHighBits(knownZero, Leaders);
  return {knownZero, knownOne};
}
```

Fig. 2. The `KnownBits` transformers for `urem` operator in LLVM

Although some individual transformers contain "unnecessary fragments" such as $a.l\&b.r$, each transformer still has a smaller size than the best transformer as a single monolithic expression.

2.2 Case Study: Synthesizing a Precise Transformer for `urem`

To illustrate the practical capabilities of `NiceToMeetYou`, we present a detailed case study drawn from our evaluation: synthesizing a `KnownBits` transformer for the `urem` (unsigned remainder) operation. This example demonstrates how `NiceToMeetYou` synthesizes precise and non-trivial transformers that both match and exceed hand-written implementations in LLVM.

2.2.1 Background: The `KnownBits` *Domain and the* `urem` *Operator.* The `KnownBits` abstract domain models partial bit-level knowledge of integer values using two disjoint bitvectors: `Zero` and `One`. A bit is definitely zero if set in `Zero`, definitely one if set in `One`, and unknown if unset in both. This domain is widely used in compiler optimization passes such as those in LLVM.

We focus on the unsigned remainder operation `urem`, which computes L mod R for unsigned integers L (dividend) and R (divisor). Optimizing the transformer for `urem` is challenging due to its non-linear behavior and the many edge cases involving known bits.

2.2.2 Reference Implementation: LLVM's Hand-Written Transformer. LLVM includes a hand-written `KnownBits` transformer for `urem`, shown in Figure 2. It is implemented using utility functions from the `APInt` library, such as: (*i*) `countTrailingZero(x)` and `countLeadingZero(x)`: compute the number of trailing or leading zeros, and (*ii*) `setHighBits(x, k)` and `setLowBits(x, k)`: construct bitvectors with the highest or lowest k bits set to 1.

This transformer "applies" three heuristics: (*i*) If the divisor R is a multiple of 2^k (i.e., it has k trailing zeroes), then the lowest k bits of the result are equal to those of L. (*ii*) If the divisor `R` is known to be a constant, some special-case handling is applied. (*iii*) Since $L \bmod R < \min(L, R)$, the number of leading zeros in the result must be at least as large as in both L and R.

2.2.3 Synthesized Transformer in MLIR: Matching and Extending LLVM. Figure 3 shows the transformer synthesized by `NiceToMeetYou` in MLIR form. The solution `@solution` computes the meet of nine independently synthesized transformer candidates `@f1` through `@f9`.

Upon manual inspection, we find that 8/9 synthesized components recover key heuristics from the LLVM implementation. The remaining one is a new heuristic.

Recovering Existing Heuristics. As an example, transformer `@f1` matches LLVM's third heuristic. It computes the number of leading zeros in the dividend `%L`, uses that to set the highest bits in a bitvector, and constructs a `KnownBits` value accordingly. This sound heuristic (also used by LLVM) encodes the fact that the result of `urem` must have at least as many leading zeros as the dividend.

Discovering New Heuristics. Transformer `@f2` illustrates synthesis that eludes human intuition. It defines a condition `@f2_cond` and a guarded body `@f2_body`, returning `ite(@f2_cond, @f2_body, %top)` so the body applies only when the condition holds. The body `@f2_body` returns the dividend as the result. This is unsound in general, however, the synthesizer simultaneously generates a guard `@f2_cond` that ensures soundness: the transformer is only used when the dividend's maximum possible value is less than the divisor's minimum possible value. In that case, the remainder equals the dividend, and the transformer is sound.

This heuristic—"if the dividend is provably less than the divisor, then `urem(L, R)` equals `L`"—is absent from the LLVM transformer, highlighting the power of synthesis in discovering useful but overlooked cases. That synthesis outputs the code realizing this case is the cherry on top.

Our synthesized transformer is more precise than the LLVM implementation—according to our precision metric—but also complementary. That is, the LLVM and synthesized transformers are incomparable: neither subsumes the other. Their meet yields a strictly more precise transformer.

This case study highlights three key outcomes:

(1) `NiceToMeetYou` recovers hand-crafted compiler heuristics automatically.
(2) `NiceToMeetYou` discovers new, sound heuristics that are absent in existing implementations.
(3) The transformers synthesized by `NiceToMeetYou` can offer strictly better precision when combined with human-crafted ones.

This example demonstrates how `NiceToMeetYou` can serve as a practical, drop-in synthesis engine for compiler frameworks such as MLIR, producing transformers that are not only correct and efficient but also competitive with and complementary to those written by domain experts.

3 An Ideal Algorithm for the Transformer Synthesis Problem

In this section, we present an idealized synthesis algorithm for solving the transformer synthesis problem (Definition 2.3). We will provide a practical instantiation using MCMC search in Section 4.

Our algorithm draws inspiration from recent approaches for synthesizing the most precise conjunctive specifications [23, 24]. Rather than generating an entire conjunction in one step, these methods iteratively synthesize individual conjuncts, ensuring that each new conjunct strictly

```
func.func @f1(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %1 = countLeadingZero(%L.zero)
    %knownZero = setHighBits(0, %1)
    return makeKnownBits(%knownZero, 0)
}
func.func @f2_cond(%L: KnownBits, %R: KnownBits) -> bool {
    %Lmax = negate(%L.zero)
    %Rmin = %R.one
    %cond = unsignedLessThan(%Lmax, %Rmin)
    return %cond
}
func.func @f2_body(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return %L
}
func.func @f2(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return ite(@f2_cond(%L, %R), @f2_body(%L, %R), %top)
}
...
func.func @solution(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return meet(@f1(%L, %R), ... , @f9(%L, %R))
}
```

Fig. 3. Our synthesized KnownBits transformers for urem, written in MLIR.

improves the overall precision. We adopt a similar strategy, tailored to the transformer setting, by incrementally synthesizing individual transformers that refine precision.

3.1 Synthesizing One Transformer at a Time

In our setting, meet operations over transformers play the role of conjunctions, and individual transformers correspond to conjuncts. Algorithm 1 maintains a set of sound, incomparable transformers $\mathcal{F}^s$, initialized to the empty set, representing the most imprecise transformer $\top$. The algorithm iteratively synthesizes new transformers that, when combined via meet, minimize the imprecision measured by the norm function. It loops as long as it can find a precision improvement:

Algorithm 1: IdealSynthesizeTransformers(prob)

```
1 Input: prob — An instance of the Transformer Synthesis Problem.
2 Output: F^s — A set of synthesized transformers solving prob.
3 F^s ← ∅                                          // Initialize to most imprecise transformer set
4 while true do
5     f ← SynthesizeTransformer(F^s, prob)         // Synthesize transformer maximizing precision gain
6     if ||f ⊓ F^s_⊓|| = ||F^s_⊓|| then
7         return RemoveRedundant(F^s)              // Termination: remove transformers, preserving norm
                                                   // (compute a set greedily; it may not be unique)
8     F^s ← F^s ∪ {f}                              // update set of synthesized transformers
```

A key advantage of this iterative approach is the reduction of the synthesis problem to repeatedly generating individual transformers. Specifically, rather than directly synthesizing a full set $\mathcal{F}^s$ minimizing the norm $\left\|\mathcal{F}^s_\sqcap\right\|$, each iteration seeks a transformer f that minimizes the norm when added to the set of transformers $\mathcal{F}^s$ we already have synthesized:

$$\underset{f \in \mathcal{L},\ \mathsf{sound}(f)}{\text{minimize}} \quad \left\| f \sqcap \mathcal{F}^s_{\sqcap} \right\|. \tag{1}$$

This structure simplifies the search space and modularizes synthesis, as each `SynthesizeTransformer`$(\mathcal{F}^s, prob)$ call focuses solely on the next incremental improvement.

Theorem 3.1 (Soundness). *If* `SynthesizeTransformer` *synthesizes a single sound transformer that is a solution to Equation* (1) *and there exists a finite solution to the transformer synthesis problem, then Algorithm 1 returns a solution to the transformer synthesis problem.*

Proof. Let $\mathcal{F}$ be a solution to the synthesis problem such that for every $\|\mathcal{F}\| = k$. Because every call to `SynthesizeTransformer` reduces the norm (and the norm is an integer), the algorithm terminates. Furthermore, the final set satisfies all three conditions: all transformers are sound, the precision cannot be improved further, and no transformer is redundant. □

Note that any intermediate result of the algorithm is a valid transformer, though (probably) sub-optimal. This property is important for our MCMC-based approach in Section 4.

3.2 Focusing Precision on Relevant Inputs

A further benefit of this approach is the ability to focus synthesis efforts on the inputs that matter, that is, inputs where the current transformer set $\mathcal{F}^s$ is still imprecise.

While soundness requires that synthesized transformers behave correctly on all inputs, precision only needs to improve where existing transformers leave room for refinement. Thus, each `SynthesizeTransformer` call can restrict its optimization to the subset of "imprecise" inputs:

$$\mathcal{A}_{\text{imprecise}} = \mathcal{A} \setminus \{a \mid \mathcal{F}^s_{\sqcap}(a) = \widehat{f^{\#}}(a))\}. \tag{2}$$

We can equivalently rewrite the objective for the next transformer f as:

$$\underset{f \in \mathcal{L}}{\text{minimize}} \quad \left\| f \sqcap \mathcal{F}^s_{\sqcap} \right\|_{\mathcal{A}_{\text{imprecise}}} \tag{3}$$

Here, $\|\cdot\|_{\mathcal{A}_{\text{imprecise}}}$ computes norm considering only inputs in $\mathcal{A}_{\text{imprecise}}$. This refinement preserves correctness while avoiding wasted effort on already-precise regions of the input space. Formally, $\underset{f \in \mathcal{L}}{\text{minimize}} \ \left\| f \sqcap \mathcal{F}^s_{\sqcap} \right\| \iff \underset{f \in \mathcal{L}}{\text{minimize}} \ \left\| f \sqcap \mathcal{F}^s_{\sqcap} \right\|_{\mathcal{A}_{\text{imprecise}}}$.

This selective focus is efficient (it allowing us to compute the norm on fewer inputs) and also aligns with standard optimization principles: prioritize areas of maximum potential gain.

4 Randomly Searching for Abstract Transformers using MCMC

The ideal algorithm presented in Section 3 incrementally synthesizes abstract transformers that collectively form the solution to the transformer synthesis problem (Definition 2.3). To make this process practical, we implement the core routine `SynthesizeTransformer`$(\mathcal{F}^s, prob)$ using a stochastic search procedure `MCMCSynthesizeTransformer`$(\mathcal{F}^s, prob)$ based on Markov Chain Monte Carlo (MCMC). Our approach is inspired by Stoke [31] and contributes a novel cost function and abductive refinement strategy.

The goal of this random search, presented in Algorithm 2, is to synthesize a transformer that minimizes the cost function defined in Line 3. The cost combines two objectives: maximizing soundness and minimizing norm. `Soundness`(f) returns a number between 0 and 1 representing the fraction of inputs on which the transformer f is sound. `Improvement`(f, g) returns a number between 0 and 1 representing how much $f \sqcap g$ improves the precision of g. Formally, we define a predicate `soundAt`$(f, a) := \widehat{f^{\#}}(a) \sqsubseteq f(a)$ indicating transformer f is sound at an abstract input

Algorithm 2: `MCMCSynthesizeTransformer`$(\mathcal{F}^s, prob)$

1 ***Input:*** $\mathcal{F}^s$ — *Current set of synthesized transformers;*
2 ***Output:*** *A new sound transformer* $f \in \mathcal{L}$ *that (in the limit) minimizes cost.*
3 **fun** `Cost`$(f^\#)$:
4 ⌊ **return** $\lambda(1-\texttt{Soundness}(f^\#)) + \kappa(1-\texttt{Improvement}(f^\#, \mathcal{F}^s))$ `// reward soundness, precision improv.`
5 $f \leftarrow$ `initialize()` `// random initial program`
6 **for** $i \leftarrow 1$ **to** N_{step} **do**
7 ⌊ $f' \leftarrow$ `mutate`(f) `// mutate current candidate`
8 ⌊ $p \sim \mathcal{U}(0,1)$ `// sample acceptance threshold`
9 ⌊ **if** `Cost`(f) − `Cost`$(f') > T \cdot \log(p)$ **then**
10 ⌊ ⌊ $f \leftarrow f'$ `// accept proposed candidate`
11 **if** `Soundness`$(f) < 1$ **then**
12 ⌊ **return** $\top$ `// return trivial top transformer if no sound one found`
13 **return** f `// return the lowest cost sound transformer found`

a. As we target only finite abstract domains, `Soundness` and `Improvement` can be defined as in Equation (4) and Equation (5), respectively:

$$\texttt{Soundness}(f) := \left(\sum_{a \in \mathcal{A}} \mathbf{1}[\texttt{soundAt}(f, a)]\right) / |\mathcal{A}| \tag{4}$$

$$\texttt{Improvement}(f, g) := \left(\sum_{a \in \mathcal{A}} \mathbf{1}[\texttt{soundAt}(f, a)] \cdot (|g(a)| - |f(a) \sqcap g(a)|)\right) / \|g\|_{\mathcal{A}} \tag{5}$$

Note that `Improvement` counts precision gain only on sound inputs. If f is sound over the entire abstract domain, we have $\texttt{Improvement}(f, g) = 1 - \|f \sqcap g\|_{\mathcal{A}} / \|g\|_{\mathcal{A}}$ because $\Sigma_{a \in \mathcal{A}} |g(a)| = \|g\|_{\mathcal{A}}$.

The algorithm begins by sampling a random candidate program f from the DSL $\mathcal{L}$ (Line 5). It then performs N_{step} iterations of local search, where in each iteration a syntactic mutation produces a new candidate f' (Line 7). If f' has a lower cost than the current candidate, it is accepted; otherwise, it is accepted with a probability determined by the difference in cost and temperature parameter T, following the standard Metropolis-Hastings acceptance rule (Line 9).

If no sound transformer is found after the search, the algorithm returns the trivial transformer $\top$, representing the most imprecise but sound abstraction (Line 11). Otherwise, the most precise sound transformer discovered is returned.

The use of MCMC for synthesizing transformers provides important asymptotic guarantees' assuming ergodicity (discussed below). In the limit, the search procedure samples transformers according to a distribution biased toward lower-cost candidates. As the number of iterations tends to infinity, the probability of synthesizing a sound transformer that is sound and minimizes norm approaches one. The following corollary of the standard convergence guarantees for Metropolis-Hastings MCMC [12] captures this result:

Corollary 4.1 (Asymptotic Optimality of MCMC Search). *Assume the proposal distribution used by* `mutate`$(\cdot)$ *is ergodic over the space of programs expressible in* $\mathcal{L}$*. Then, as* $N_{step} \to \infty$*, with probability approaching 1,* `MCMCSynthesizeTransformer`$(\mathcal{F}^s, prob)$ *returns* f *satisfying:*

$$\left\|f \sqcap \mathcal{F}^s_\sqcap\right\| = \min_{\substack{f' \in \mathcal{L} \\ \texttt{Soundness}(f')=1}} \left\|f' \sqcap \mathcal{F}^s_\sqcap\right\|.$$

Algorithm 3: randomly initializing and randomly mutating transformer operations

```
1  fun initializeNormal() :
2      l ← maximum number of lines
3      for i ← 1 to l do
4          op ← sample(operator in ℒ)
5          a ← sample(0, i − 1)        // only use earlier variable indices or the input variable x_0
6          b ← sample(0, i − 1)
7          s[i] ← (x_i = op(x_a, x_b))
8      return s[1] ··· s[l]
9  fun mutateNormal(f) :
10     x_i = op(x_a, x_b) ← sample(line in f)          // sample a line of the program to mutate
11     op′ ← sample(operator in ℒ)
12     a′ ← sample(0, i − 1)
13     b′ ← sample(0, i − 1)
14     return f[x_i = op(x_a, x_b)/x_i = op′(x_a′, x_b′)]      // replace i-th line with randomly sampled one
```

In short, repeated or sufficiently long runs of `MCMCSynthesizeTransformer` should yield transformers that are sound and highly precise in terms of the norm function. Though each invocation of the algorithm is approximate, in expectation the overall result tends to maximum precision.

On Ergodicity. Our setup is designed to provide an ergodic search space. First, the space is in a simple format parameterized by the operation set, and further constrained by the program representation: each operation takes two abstract inputs, consists of a sequence of SSA instructions, and produces one abstract output. Only the instructions in the middle are subject to mutation. Second, our mutation strategy ensures the ergodicity of the Markov chain because the two possible mutations are invertible. Thus, there is a positive probability of transitioning from any program to any other. Overall, our strategy is similar to the one employed by Stoke [31].

4.1 Two Strategies for Randomly Sampling Programs

We now describe two complementary strategies that we can use in Algorithm 2 to synthesize transformers via random search. The strategies correspond to different implementations of the `initialize` and `mutate` procedures in Algorithm 2 (Lines 5 and 7).

4.1.1 Randomly Mutating Transformer Operations. The first strategy performs random syntactic mutations to the current transformer by altering its operations or structure (Algorithm 3). This corresponds to traditional MCMC-based synthesis, which operates directly over the DSL $\mathcal{L}$ [31].

For simplicity, the formalization assumes all operations are binary and that x_0 is the input variable to the transformer. In practice there are more possible inputs as well as constants that can be used in the transformer. Algorithm 3 defines two core functions: `initializeNormal` and `mutateNormal`, which build and mutate transformers as sequences of statements. The initialization function creates a transformer with l statements (this is a parameter in our implementation). Each statement $s[i]$ assigns a variable x_i by applying a randomly selected operator from $\mathcal{L}$ to two variables previously defined variables at lower indices. This ensures well-formed data dependencies.

The mutation function (`mutateNormal`) selects a random statement, sampled using a user-given probability distribution, and replaces it with a newly sampled statement, again selecting operator and operands consistent with variable dependencies.

This setup supports incremental local modifications during MCMC sampling, enabling the search to explore transformer candidates efficiently and effectively.

Algorithm 4: `initialize` and `mutate` for condition abduction

```
1 fun initializeAbd():
2     C^P ← history of precise but unsound transformers encountered so far
3     f_u ← random element from C^P                    // select unsound but precise transformer
4     c ← initializeNormal()   // sample a random condition expressible in ℒ using Algorithm 3
5     return ite(c, f_u, ⊤)
6 fun mutateAbd(ite(c, f_u, ⊤)):
7     c' ← mutateNormal(c)      // randomly mutate c using DSL operators in ℒ using Algorithm 3
8     return ite(c', f_u, ⊤)
```

For example, the following transformer from Figure 3 is a valid initial transformer (we avoid using variable indices for readability and instead use actual variable names):

```
func.func @f1(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %1 = countLeadingZero(%L.zero)
    %knownZero = setHighBits(0, %1)
    return makeKnownBits(%knownZero, 0)
}
```

That same transformer could be mutated into the following one by replacing the argument of `countLeadingZero` with a different input:

```
func.func @f1(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %1 = countLeadingZero(%R.zero)  // mutated
    %knownZero = setHighBits(0, %1)
    return makeKnownBits(%knownZero, 0)
}
```

4.1.2 Randomly Adding Conditions to Unsound but Precise Transformers. Throughout a typical random search in which Algorithm 2 uses the mutation strategy from Section 4.1.1, the algorithm discovers transformers that significantly improve precision but are not sound. Such transformers potentially contain valuable information, but cannot be used directly. To address this issue, we use our MCMC approach to implement abductive synthesis, which constructs guards that identify subsets of the input space. Given an unsound transformer $f^\#$, we aim to synthesize a guard c (written in the DSL $\mathcal{L}$) and thereby construct a sound transformer $f_c^\#(a) := \texttt{ite}(c,\ f^\#(a),\ \top)$. We denote a conditional transformer as a term $\texttt{ite}(c,\ f^\#,\ \top)$.

The `initializeAbd` and `mutateAbd` operations in Algorithm 4 implement this abductive synthesis by reusing the initialization and mutation operations in Algorithm 3 to explore the space of possible conditions c starting from an unsound transformer. The remaining structure of Algorithm 2 is untouched as the cost function and overall structure of the algorithm are exactly the same regardless of what type of transformer we decide to synthesize. For example, the following transformer `f_2` from Figure 3 is a valid initial transformer for Algorithm 4, assuming again proper renaming of variables and that the last variable is being returned:

```
func.func @f2_cond(%L: KnownBits, %R: KnownBits) -> bool {
    %Lmax = negate(%L.zero)
    %Rmin = %R.one
    %cond = unsignedLessThan(%Lmax, %Rmin)
    return %cond
}
func.func @f2_body(%L : KnownBits, %R : KnownBits) -> KnownBits { return %L }
```

```
func.func @f2(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return ite(@f2_cond(%L, %R), @f2_body(%L, %R), %top)
}
```

That same transformer could be mutated by modifying `f2_cond`, e.g., by changing the second argument of `unsignedLessThan`, but the body `f2_body` would remain unchanged:

```
func.func @f2_cond(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %Lmax = negate(%L.zero)
    %Rmin = %R.one
    %cond = unsignedLessThan(%Lmax, %R.zero)  // mutated
    return %cond
}
```

5 Implementation

We have implemented `NiceToMeetYou` as a modular and extensible synthesis framework that supports several finite non-relational integer abstract domains and concrete operations. Figure 4 presents a high-level overview of the system architecture. This section first describes how we instantiate the core synthesis algorithm to support different domains and instruction semantics (Section 5.1), and then outlines key engineering optimizations that enable efficient large-scale synthesis (Section 5.2). The implementation is publicly available [13].

5.1 The Basic Ingredients

Synthesis of an approximation of the ideal transformer $\widehat{f^{\#}}$ takes place through a two-loop process: (*i*) A slow *outer loop* performs bounded model checking (via z3) to validate soundness, and (*ii*) A fast *inner loop* generates new candidates and aggressively tests their soundness and precision by sampling the space of abstract values. In practice, we run 1,000 inner-loop iterations per outer-loop iteration. After each outer iteration, the framework discards provably unsound candidates, adds sound candidates to the solution set $\mathcal{F}^s$, and updates weights in the probability distribution used to sample MCMC mutations.

While the ultimate goal is to produce a set of sound abstract transformers $\mathcal{F}^s$, our implementation also maintains a set of unsound-but-precise functions, $\mathcal{F}^p$, as well. The purpose of $\mathcal{F}^p$, as outlined in Section 4.1.2, is to discover sound functions in a bottom-up way from a precise function $f^{\#} \in \mathcal{F}^p$ by synthesizing a condition c that narrows the input space to a subset on which $f^{\#}$ is sound. Concretely, the goal is to find c such that $f^{\#}_c(x) := \texttt{ite}(c(x),\ f^{\#}(x),\ \top)$ is sound.

Accordingly, the system performs two interleaved forms of MCMC-guided synthesis. In each round of the inner loop:

- When discovering new sound transformers starting from a transformer $f^{\#}$, synthesis mutates $f^{\#}$ itself.
- When discovering guard conditions for existing transformers $f^{\#} \in \mathcal{F}^p$, synthesis keeps $f^{\#}$ fixed and mutates c in $f^{\#}_c(x) := \texttt{ite}(c(x),\ f^{\#}(x),\ \top)$.

When the process converges or time runs out, we return the meet of all sound candidates $\mathcal{F}^s_{\sqcap}$. The precise set $\mathcal{F}^p$ is discarded.

5.1.1 Input Language. Our target DSL $\mathcal{L}$ for synthesis is the MLIR dialect `xdsl-smt` [4]. This language has several important characteristics: (1) it includes basic numeric operations (+, /, `ite`), making it suitable to express a variety of transformers; (2) it uses SSA form, which narrows the scope of possible mutations; (3) it implements SMT-Lib [1] and has a direct mapping to z3; and (4) it has a straightforward mapping to `C++`, enabling fast testing of transformers in the inner loop.

To instantiate our framework, users must provide the semantics of the concrete operation for which a transformer is to be synthesized, and the definition of the abstract domain over which the transformer operates. `NiceToMeetYou` targets one concrete operation at a time in its outer loop (hence the multi-arrow in Figure 4). Multiple loops can of course run in parallel.

For each concrete operation, the user provides both a z3 implementation (for verification in the outer loop) and a corresponding `C++` implementation (for fast evaluation in the inner loop). Users must ensure these are in agreement. Operations may additionally specify input constraints prohibiting abstract values—e.g., integer division excludes zero denominators. For each abstract domain, the user supplies a top element, a meet operation, a concretization function (technically, only a membership test: the function is given a concrete value v and an abstract value A, and must determine whether $v \in \gamma(A)$), a size function to guide precision, and an optional well-formedness constraint to rule out syntactically valid but semantically invalid abstract values (e.g., for `KnownBits`, the two bitvectors representing known-zero and known-one must not overlap). Information about concrete operations can be reused across abstract domains. Information about abstract domains can potentially be reused across different sets of concrete operations.

Turning back to our toy example from Section 2, the following elements suffice to instantiate `NiceToMeetYou` for `max` on the interval domain. Each piece must be defined in MLIR:

- Concrete op: $f(x, y) = \max(x, y)$
- Input constraint: None
- Top element: $[0, \texttt{UINT\_MAX}]$
- Meet: $a \sqcap b = [\max(a.l, b.l), \min(a.r, b.r)]$
- Concretization: $\gamma([a.l, a.r]) = \{a.l, \cdots, a.r\}$
- Size function: $|a| = \lfloor \log_2(|a.l - a.r|) \rfloor$
- Well-formedness constraint: $\forall\ a.\ a.l \leq a.r$

5.1.2 Initializing MCMC in the Inner Loop. Each iteration of the inner MCMC synthesis loop begins by generating a fresh pool of candidate transformer functions and candidate guard conditions. Candidate functions are initialized with approximately 30 randomly generated instructions, drawn from the DSL described in Section 5.1.1. These functions are intended to represent full transformer logic. Candidate conditions, which are used for condition abduction (Section 4.1.2), are initialized with around 6 instructions, reflecting their role as lightweight guards. In both cases, mutation attempts to morph these placeholder candidates into codes that outperform the current members of $\mathcal{F}^s$. We chose the constants 30 and 6 empirically; similar values work as well.

The scoring function used to evaluate candidates balances two objectives: soundness and precision. Two tunable parameters, κ and λ (from Algorithm 2), govern this tradeoff. During the early stages of synthesis, we prioritize exploratory behavior by assigning a higher weight to precision for functions (λ) and to soundness for conditions (κ). These weights encourage exploration; later on, the verifier prunes unsound functions and the scoring function de-emphasizes overly precise conditions. The relative value of these parameters is important; the exact value is not so much.

5.1.3 Mutation. Mutation is the simplest component of the implementation. It operates directly on MLIR programs expressed in the `xdsl-smt` dialect. Each candidate is a sequence of SSA assignments of the form $x_i = op(x_a, \ldots)$ where op is an `xdsl-smt` operator, i is the current line number, and a is some index preceding i. There are two possible mutations: replace one variable reference x_a with another, or replace the operator with another op' (with randomly selected arguments). Our DSL consists of 29 operators in total, each of which corresponds to a function in `APInt` library.

Variables are always selected uniformly at random. Each index in the range $[0, i)$ has equal probability. Operators are initially chosen uniformly at random, but their weights get adjusted after each update to the sound set $\mathcal{F}^s$. The operators that are most common across the sound functions have the highest chance of selection. Concretely, $\mathit{weight}(op) := 1 + \mathit{frequency}(op, \mathcal{F}^s)$, where $\mathit{frequency}(op, F)$ is the total number of op appearances in all transformers in F.

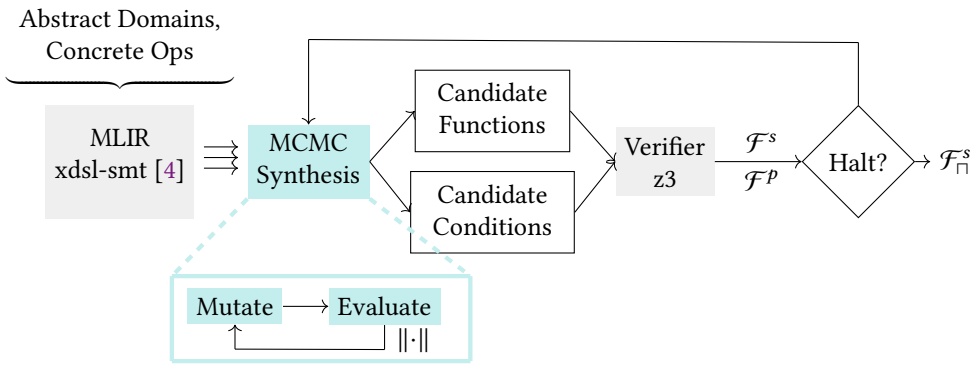

Fig. 4. Implementation overview. The outer loop (above) checks the soundness of promising candidates. The inner loop (below) generates new candidates via mutation and evaluates candidates' soundness and precision.

Weights never decrease below a positive minimum, ensuring all operators maintain a nonzero probability of selection and preserving exploration diversity throughout synthesis.

5.1.4 Evaluation Engine. After mutation generates candidate functions and conditions, a fast evaluation engine implemented in `C++` tests their soundness and precision by executing the semantics of the candidates. This engine implements the core procedure outlined in Algorithm 2. We choose `C++` for its fast performance and good integration with LLVM infrastructure—most notably, with LLVM's type `APInt` for arbitrary-precision integers.

Since our transformers operate over integer domains, the evaluation engine is tailored to efficiently test integer values in several ways:

- **Bitvector representation:** The bitvectors that we use for static analysis are not faithfully represented by standard `C++` datatypes such as `uint64_t`. We use LLVM's `APInt` numbers instead. Crucially, an `APInt` can be instantiated at any bitwidth and comes with numerous helper functions, e.g., for overflow checking.
- **Test generation by bitwidth:** Integers can be enumerated in a straightforward way. This property guides test generation:
  - For small bitwidths (range $[1, 4]$), evaluation tests all possible abstract values and concrete inputs. Candidate functions are thus guaranteed to be sound on small bitwidths. *Examples:* For `KnownBits`, there are 120 small abstract values. For `ConstantRange`, there are 185 abstract values. To test a transformer $f^\sharp$, we compute $a = f^\sharp(a_0, a_1)$ for all pairs of abstract values a_0, a_1, and, unless a is bottom, we check that $f(n_0, n_1) \in a$ for all pairs of concrete numbers.
  - For mid-size bitwidths ($[5, 8]$), evaluation samples abstract values a and exhaustively tests the concretization of a. Candidate functions are sound over sampled tests, but may be unsound over the entire abstract domain; z3 verification is there to catch mistakes.
  - For large bitwidths ($[9, 64]$), evaluation samples abstract values a and samples elements $c \in \gamma(a)$. Due to this incomplete concretization, candidate functions could be unsound even on sampled abstract values at this stage.

The extraction of an MLIR candidate to `C++` requires some care beyond the use of `APInt` values. Operations that can lead to undefined behavior in `C++`, such as division, must be guarded with a check for invalid input and must return a default value that matches the behavior of the z3 verifier. For division, invalid input is a zero denominator and the z3 default is to return zero. Our goal here

is simply to keep the evaluation engine and the verifier in sync. If static analysis discovers that a program definitely divides by zero, other passes in LLVM will take appropriate action.

5.1.5 Size Functions. Each abstract domain has a size function $|\cdot|$ to quantify precision:

- For the `KnownBits` domain, the size measures the number of unknown bits. For example, the `KnownBits` abstract value `01?1?00?` has size 3.
- For the `ConstantRange` domain, the size is computed as the $\lfloor \log_2 \rfloor$ of the absolute difference between the lower and upper bounds of the abstract intervals. For example a `ConstantRange` abstract value of `[15, 45]` has a size of $\lfloor \log_2(30) \rfloor = 4$

5.1.6 Integrating MCMC Results in the Outer Loop. At each outer-loop iteration, the inner MCMC loop generates a collection of candidate transformers and guard conditions, ranked by their size. These candidates are then filtered and integrated as follows:

- Sound candidates that are not subsumed by any existing member of the solution set $\mathcal{F}^s$ are added to it. Subsumption is determined using the abstract domain's ordering (e.g., inclusion).
- Precise but unsound candidates are selectively retained in $\mathcal{F}^p$, the pool used for condition abduction (Section 4.1.2). To control memory and evaluation cost, $\mathcal{F}^p$ is capped at a fixed size (15 in our implementation), and retains only the top-scoring elements.

This strategy ensures that $\mathcal{F}^s$ grows monotonically with respect to both soundness and variety, while $\mathcal{F}^p$ remains a focused source of potentially useful components for future synthesis.

5.1.7 Verifier. Soundness is established via z3. For each candidate produced by the inner loop, we check it across all possible inputs for bitwidths ranging from 1 to 64. This validation ensures that the candidate respects the transformer specification over the full concrete input space. If z3 times out, we simply assume the transformer is unsound and discard it. To perform this check, we use the lowering to z3 that is provided by the `xdsl-smt` dialect. We use this lowering directly, though we did fix several bugs in it along the way [21, 22]. Candidates that pass verification are admitted to $\mathcal{F}^s$ and considered sound in subsequent synthesis and scoring phases.

5.2 Gotta Go Fast

`NiceToMeetYou` can find promising transformers only because it can quickly explore a huge number of candidates. Most candidates are garbage. Sifting through the trash in a reasonable time is possible thanks to details of its engineering, described below.

Parallel MCMC. The MCMC synthesis in our inner loop is a slow, step-by-step process. To accelerate converge and improve the likelihood of high coverage, we run this inner loop in parallel (currently, 100 times). Operationally, each time we enter the inner loop with a set of candidate functions, we spawn several copies of MCMC that independently mutate and evaluate transformers. At the end of the inner loop, the highest-scoring candidates from each process are selected as candidates for the outer loop to verify. MCMC splits its focus between sound candidates ($\mathcal{F}^s$) and precise candidates ($\mathcal{F}^p$) on a 70/30 basis: 70 % of the parallel searches explore sound candidates, and 30 % of the searches perform condition abduction.

LLVM JIT. Our evaluation engine uses the LLVM JIT, `LLJIT` [19], to quickly compile and test candidate functions. We create a pre-compiled binary that packages the abstract domain, the `APInt` API, our scoring function, and LLJIT, and use it to lower candidates written in MLIR to executable code. Compared to our initial approach of invoking `clang`, linking with LLVM, and compiling the candidates, JIT-ing improves performance by roughly two orders of magnitude. Testing a batch of roughly 100 candidates dropped from seconds to milliseconds with LLJIT.

5.3 Generalizability of the Approach

We have focused on `KnownBits` and `ConstantRange` domains because these are the main abstract domains employed by LLVM. There are no fundamental limitations to adapting `NiceToMeetYou` to other abstract domains:

- *Non-relational abstract domains:* `NiceToMeetYou` can be directly applied to non-relational domains where each abstract value consists of a finite number of integers, as in `KnownBits` or non-wrapping interval domains. Users must provide the components described above (Section 5.1.1) and a soundness verifier. Reuse is made simple because the template of mutated programs begins by deconstructing each input abstract value into integers and ends by reconstructing the output abstract value from several integers, and only the operations in the middle are mutated. To support non-relational abstract domains that are not composed of integers, significant engineering is needed to define suitable templates and mutation strategies.
- *Relational abstract domains:* We conjecture that `NiceToMeetYou` can synthesize transformers for relational domains, such as octagons and polyhedra. We have deferred investigation because we suspect these domains will lead to prohibitively high compilation overhead. In contrast to a non-relational domain, where program variables can be tracked independently, a relational domain requires all variables to be tracked simultaneously. A transformer must operate over entire program states rather than individual variables, which complicates reuse across operators.
- *Non-Galois abstract domains:* Our abstract domains are characterized by a Galois connection. This means there is a unique most precise abstraction for each concrete set of integers, a fact that our inner MCMC loop leverages to speed up evaluation. To adapt `NiceToMeetYou` to a non-Galois-connection-based abstract domain, the framework would need a method for choosing among multiple sound abstract values. Take the wrapped interval domain as an example [11]. There is no most precise abstraction for integers in a circle; any number can be the left endpoint.

`NiceToMeetYou` targets MLIR, but there is nothing fundamental about our choice to use it. MLIR is convenient because it has an SMT dialect [10], a dataflow analysis framework with pluggable transfer functions, and a clear relation with LLVM to facilitate comparisons.

6 Evaluation

Our evaluation consists of several parts: a direct comparison between LLVM's `KnownBits` and `ConstantRange` transformers and ours (Section 6.1); an evaluation of our known bits transfer functions vs. LLVM's, in terms of precision and compilation time, when compiling an appropriate subset of the SPEC 2017 benchmark suite (Section 6.2); evidence that synthesis facilitates exploration on composite transfer functions and a reduced product domain (Section 6.3); ablation studies of DSL choice (Section 6.4) and condition abduction (Section 6.5); and, a detailed comparison to Amurth [15] (Section 6.6). Note that, although we compare against LLVM to show that our work is practical beating LLVM is not a goal in and of itself. `NiceToMeetYou` is a complement to handcrafted analyses.

6.1 Comparison to LLVM's Abstract Transformers

LLVM provides abstract transformers for a handful of abstract domains, notably `KnownBits` and `ConstantRange`. These transformers are handwritten and have been fine-tuned over the years. Currently, LLVM supports 44 of the 47 binary operators for `KnownBits` and 37 of 47 operators for `ConstantRange`. To evaluate the effectiveness of our synthesis approach, we use our tool to automatically generate abstract transformers for operators in both domains.

Table 1. KnownBits results for 8-bit and 64-bit integers. The 8-bit results report the percentage of tests where the transformer matches the best transformer $\widehat{f^{\#}}$. The 64-bit results report the norm of the transformer (normalized by the number of **Tests**) which reflects its imprecision. Higher (↑) is better for **exact**, while lower (↓) is better for **norm**. $\#f^{\#}$ is the number of transformers in $\mathcal{F}^{s}$. $\#c$ counts those with conditions. #inst is the total number of MLIR instructions used by those transformers. **Tests** is the number of sampled test inputs. ⊤, synth, and LLVM present results for three transformers: a trivial one that always returns top, our synthesized transformer, and LLVM's manually-implemented transformer. The meet column is for the transformer synth ⊓ LLVM. **Bolded** numbers denote cases where the meet of the synthesized transformer and the LLVM one has higher precision than the LLVM one. If no LLVM implementation is available (N/A) we bold cases in which the synthesized transformer is better than ⊤. Some 64-bit rows contain a dash (-) because random sampling failed to produce any valid pairs of inputs. These operators have strict constraints on their inputs.

ConcreteOp				Tests	8-bit exact (%) ↑				Tests	64-bit precision (norm) ↓			
	$\#f^{\#}$	$\#c$	#inst		⊤	synth	llvm	meet		⊤	synth	llvm	meet
Abds	10	3	189	1000	33.90	60.10	100.00	100.00	10000	0.059	0.050	0.000	0.000
Abdu	16	4	259	1000	33.10	59.40	100.00	100.00	10000	0.059	0.050	0.000	0.000
Add	17	2	306	1000	29.60	58.70	100.00	100.00	10000	0.140	0.082	0.000	0.000
AddNsw	13	2	205	1000	24.50	42.00	100.00	100.00	9674	0.147	0.136	0.000	0.000
AddNswNuw	14	3	220	1000	7.40	45.50	100.00	100.00	7479	0.160	0.136	0.000	0.000
AddNuw	17	4	291	1000	15.20	53.90	100.00	100.00	8305	0.152	0.103	0.000	0.000
And	1	0	14	1000	0.10	100.00	100.00	100.00	10000	0.625	0.000	0.000	0.000
Ashr	5	3	94	1000	31.10	65.50	85.70	85.70	0	-	-	-	-
AshrExact	7	2	123	1000	14.70	40.10	100.00	100.00	0	-	-	-	-
AvgCeilS	8	4	157	1000	31.80	38.80	100.00	100.00	10000	0.139	0.136	0.000	0.000
AvgCeilU	8	6	153	1000	31.60	38.60	100.00	100.00	10000	0.139	0.136	0.000	0.000
AvgFloorS	9	3	163	1000	32.40	39.30	100.00	100.00	10000	0.139	0.136	0.000	0.000
AvgFloorU	7	4	138	1000	32.20	37.70	100.00	100.00	10000	0.139	0.136	0.000	0.000
Lshr	4	1	80	1000	12.30	59.30	96.50	96.50	0	-	-	-	-
LshrExact	6	2	119	1000	12.80	31.40	100.00	100.00	0	-	-	-	-
Mods	12	3	235	1000	41.20	64.70	71.30	**71.50**	10000	0.090	0.078	0.076	0.076
Modu	12	4	207	1000	16.70	59.00	52.70	**70.60**	10000	0.149	0.030	0.132	**0.027**
Mul	10	6	202	1000	25.60	60.60	73.20	**73.30**	10000	0.025	0.010	0.006	**0.006**
Or	1	0	8	1000	0.00	100.00	100.00	100.00	10000	0.624	0.000	0.000	0.000
Sdiv	11	7	248	1000	64.10	64.10	83.40	83.40	10000	0.331	0.331	0.114	0.114
SdivExact	2	2	32	1000	19.30	19.30	37.40	37.40	0	-	-	-	-
Shl	4	1	69	1000	10.50	56.90	96.50	96.50	0	-	-	-	-
ShlNsw	7	1	115	1000	6.80	26.20	100.00	100.00	0	-	-	-	-
ShlNswNuw	7	2	115	1000	5.50	9.80	100.00	100.00	0	-	-	-	-
ShlNuw	7	3	139	1000	10.60	40.50	100.00	100.00	0	-	-	-	-
Smax	9	5	185	1000	6.50	63.80	100.00	100.00	10000	0.348	0.095	0.000	0.000
Smin	6	4	119	1000	6.00	72.80	100.00	100.00	10000	0.349	0.064	0.000	0.000
SshlSat	7	1	124	1000	37.60	72.40	N/A	**72.40**	10000	0.624	0.109	N/A	**0.109**
Sub	12	2	204	1000	28.50	60.60	100.00	100.00	10000	0.140	0.088	0.000	0.000
SubNsw	16	5	287	1000	22.20	47.80	100.00	100.00	9673	0.146	0.103	0.000	0.000
SubNswNuw	14	6	292	1000	7.60	31.80	100.00	100.00	7515	0.160	0.139	0.000	0.000
SubNuw	18	4	323	1000	16.40	47.10	100.00	100.00	8215	0.152	0.099	0.000	0.000
UaddSat	14	5	305	1000	18.30	61.80	100.00	100.00	10000	0.253	0.069	0.000	0.000
Udiv	10	5	191	1000	2.50	80.80	89.80	**90.90**	10000	0.960	0.004	0.001	0.001
UdivExact	3	2	53	1000	2.80	15.30	33.90	**42.20**	0	-	-	-	-
Umax	9	6	199	1000	6.40	90.60	100.00	100.00	10000	0.351	0.002	0.000	0.000
Umin	6	1	105	1000	6.40	92.90	100.00	100.00	10000	0.351	0.001	0.000	0.000
UshlSat	2	1	41	1000	3.60	96.60	N/A	**96.60**	10000	1.000	0.000	N/A	0.000
UsubSat	10	6	232	1000	19.00	52.10	100.00	100.00	10000	0.254	0.071	0.000	0.000
Xor	3	0	39	1000	2.30	100.00	100.00	100.00	10000	0.390	0.000	0.000	0.000

Table 2. ConstantRange results for 8-bit and 64-bit integers. See the Table 1 caption for descriptions of the columns, **exact**, and **norm**. Higher (↑) is better for **exact**. Lower (↓) is better for **norm**. Operators marked by a * use synthesized $\mathtt{CR}_s$ transformers, others use $\mathtt{CR}_u$ transformers.

ConcreteOp				Tests	8-bit exact (%) ↑				Tests	64-bit precision (norm) ↓			
	$\#f^{\#}$	#c	#inst		⊤	synth	llvm	meet		⊤	synth	llvm	meet
Abds*	3	2	70	1000	59.80	59.80	N/A	**59.80**	10000	0.917	0.915	N/A	**0.915**
Abdu	20	6	344	1000	0.00	75.00	N/A	**75.00**	10000	0.990	0.908	N/A	**0.908**
Add	2	2	45	509	36.54	100.00	100.00	100.00	4991	0.949	0.887	0.887	0.887
AddNsw*	8	4	148	1000	7.10	100.00	100.00	100.00	9770	0.982	0.905	0.905	0.905
AddNswNuw	10	2	172	1000	0.00	70.70	84.80	**88.60**	8190	0.994	0.921	0.912	**0.910**
AddNuw	14	5	243	1000	0.00	94.00	100.00	100.00	8267	0.993	0.910	0.906	0.906
And	10	5	193	1000	0.00	82.30	83.10	**83.30**	10000	0.990	0.886	0.883	0.883
Ashr*	8	3	141	1000	0.00	98.00	98.50	**99.20**	0	-	-	-	-
AshrExact*	4	2	73	1000	0.00	81.70	N/A	**81.70**	0	-	-	-	-
AvgCeilS*	26	2	444	1000	0.00	0.30	N/A	**0.30**	10000	0.998	0.953	N/A	**0.953**
AvgCeilU	17	3	303	1000	0.00	1.80	N/A	**1.80**	10000	0.998	0.947	N/A	**0.947**
AvgFloorS*	20	1	332	1000	0.00	1.80	N/A	**1.80**	10000	0.998	0.955	N/A	**0.955**
AvgFloorU	17	3	276	1000	0.00	30.00	N/A	**30.00**	10000	0.998	0.932	N/A	**0.932**
Lshr	1	0	14	1000	0.00	100.00	100.00	100.00	0	-	-	-	-
LshrExact	4	1	78	1000	0.00	85.20	N/A	**85.20**	0	-	-	-	-
Mods*	6	2	118	1000	0.00	43.50	95.60	95.60	10000	0.997	0.909	0.875	0.875
Modu	8	3	164	1000	0.00	89.00	88.90	**89.00**	10000	0.989	0.877	0.877	**0.877**
Mul	17	6	371	998	90.78	90.78	90.88	90.88	10000	0.861	0.861	0.861	0.861
Or	14	8	275	1000	0.00	82.40	85.60	**86.20**	10000	0.990	0.889	0.882	0.882
Sdiv*	8	3	146	998	0.50	12.32	100.00	100.00	10000	1.000	0.997	0.501	0.501
SdivExact*	8	8	202	1000	0.70	0.70	N/A	**0.70**	0	-	-	-	-
Shl	8	4	140	1000	0.00	83.00	0.30	**83.00**	0	-	-	-	-
ShlNsw*	4	3	81	1000	0.70	18.00	99.20	**99.40**	0	-	-	-	-
ShlNswNuw	7	1	116	1000	0.00	88.30	49.80	**100.00**	0	-	-	-	-
ShlNuw	7	4	156	1000	0.00	76.40	99.60	99.60	0	-	-	-	-
Smax*	1	0	14	1000	0.00	100.00	100.00	100.00	10000	0.996	0.838	0.838	0.838
Smin*	1	0	13	1000	0.00	100.00	100.00	100.00	10000	0.996	0.839	0.839	0.839
SshlSat*	7	3	132	1000	50.90	75.50	100.00	100.00	10000	0.497	0.486	0.243	0.243
Sub	2	2	39	526	35.36	100.00	100.00	100.00	4997	0.948	0.887	0.887	0.887
SubNsw*	5	4	106	1000	7.00	100.00	100.00	100.00	9773	0.981	0.904	0.904	0.904
SubNswNuw	3	1	59	1000	0.00	74.50	84.00	84.00	8152	0.993	0.917	0.912	0.912
SubNuw	14	8	259	1000	0.00	94.50	100.00	100.00	8304	0.993	0.910	0.906	0.906
UaddSat	6	1	93	1000	0.00	99.20	100.00	100.00	10000	0.995	0.756	0.755	0.755
Udiv	7	1	126	1000	0.00	31.70	100.00	100.00	10000	1.000	0.033	0.006	0.006
UdivExact	5	2	95	1000	0.00	34.30	N/A	**34.30**	0	-	-	-	-
Umax	1	0	13	1000	0.00	100.00	100.00	100.00	10000	0.996	0.839	0.839	0.839
Umin	1	0	15	1000	0.00	100.00	100.00	100.00	10000	0.996	0.839	0.839	0.839
UshlSat	5	1	85	1000	0.00	100.00	100.00	100.00	10000	1.000	0.000	0.000	0.000
UsubSat	2	0	31	1000	0.00	100.00	100.00	100.00	10000	0.995	0.753	0.753	0.753
Xor	13	2	232	1000	50.90	56.10	66.70	**68.20**	10000	0.921	0.908	0.890	**0.887**

Our evaluation reports data for 40 of the 47 operators. The seven omitted operators—MulNsw, UMulSat, MulNuw, SAddSat, MulNswNuw, SMulSat, and SSubSat—require overflow checks that we have not yet supported in our dalect.

6.1.1 Evaluation Setting. We evaluate the precision of NiceToMeetYou synthesized transformers, LLVM built-in transformers, and the meet of both. We measure precision for 8-bit and 64-bit bitvectors on a set of randomly sampled inputs, denoted as $\mathcal{A}_{\text{test}}$. All concrete operators in our

benchmarks are binary, so each input consists of a pair of abstract values (a_1, a_2). Generally speaking, the generation of test inputs for evaluation is the same as the test generation described in Section 5.1.4, but using different random seeds.

In the 8-bit setting, the concrete domain is small (i.e., 2^8), making it feasible to compute for each sampled input the theoretically best result $\widehat{f^\#}(a_1, a_2)$ by exhaustively enumerating all concrete value pairs (c_1, c_2) such that $c_1 \in \gamma(a_1)$ and $c_2 \in \gamma(a_2)$. We therefore report the percentage of inputs for which the evaluated transformer produces exactly the abstract output $\widehat{f^\#}(a_1, a_2)$. In the 64-bit setting, exhaustive enumeration is no longer feasible as each abstract value could concretize to as many as 2^{64} concrete values. Therefore, we report the norm over the sampled test inputs $\left\|f^\#\right\|_{\mathcal{A}_{\text{test}}}$.

Specifically, for `KnownBits` and `ConstantRange` domain, we both sample 1,000 (and 10,000) pairs of abstract values for the 8-bit (and 64-bit) evaluations. For each abstract pair, we enumerate all concrete value pairs in the 8-bit setting, and sample 10,000 concrete value pairs in the 64-bit setting.

Constraints on Concrete Operators. As mentioned in Section 5.1.1, some concrete operators impose constraints on their operands. For example, shift operators require the second operand to lie within the range $[0, \text{bitwidth}]$. In such cases, a randomly sampled abstract input pair may not contain any concrete value pair that satisfies the constraint. For 8-bit inputs, we apply rejection sampling to ensure only non-empty inputs—those that contain at least one valid concrete value pair—are included. However, for 64-bit inputs, non-empty inputs could be rare, and rejection sampling becomes inefficient. Therefore, we simply skip empty inputs and exclude them from evaluation at 64 bits. As a result, the **Tests** count in some rows of the right columns in Tables 1 and 2 is less than 10,000. In extreme cases where all 10,000 sampled inputs are empty (due to how few valid inputs there might be for a certain operator), we omit the corresponding data.

Comparing to LLVM's wrapped `ConstantRange`. While the LLVM `KnownBits` domain cleanly fits into our framework, the LLVM `ConstantRange` domain presents some friction, as it consists of wrapped intervals[11]. Specifically, each element in this LLVM domain is either $\bot$ (the empty set), $\top$ (the set of all w-bit integers), or represented by $[a, b)$, where a, b are w-bit bitvectors such that $a \neq b$. This domain is a sign-agnostic domain. The concretization function for LLVM `ConstantRange` domain is defined as follows, where $<_l$ is lexicographic ordering on bit-vectors:

$$\gamma([a,b)) = \begin{cases} \{a, \ldots, b-1\} & \text{if } a <_l b \\ \{0^w, \ldots, b-1\} \cup \{a, \ldots, 1^w\} & \text{otherwise.} \end{cases}$$

The LLVM `ConstantRange` domain is a sound but non-Galois abstract domain—i.e., it does not form a Galois connection with the concrete bit-vector domain. Its abstraction function α can be understood as returning a most precise wrapped interval that covers the given set of bitvectors—i.e., no subinterval of it does. However, such a most precise interval is not guaranteed to be unique.

`NiceToMeetYou` only supports Galois-connection-based abstract domains where a best abstraction is unique. So to enable a comparison to this non-well-defined abstract domain, we synthesized transformers for the following two segment domains:

- Unsigned Intervals ($\texttt{CR}_u$): Each element is either $\bot$ or from the set $\{[a, b] \mid 0 \leq a \leq b < 2^w\}$.
- Signed Intervals ($\texttt{CR}_s$): Each element is either $\bot$ or from the set $\{[a, b] \mid 2^{w-1} \leq a \leq b < 2^{w-1}\}$.

Because our abstract domains and the not-well-defined abstract domain used by LLVM are technically incomparable, we need to make some compromises when analyzing their precision and mapping elements of one to the other. For unsigned concrete operators (e.g. `umax`) and sign-agnostic ones (e.g. `add`), $\texttt{CR}_u$ transformers are used for comparison. If the LLVM transformer produces a wrapper interval that cannot be represented as an element $\texttt{CR}_u$, we do not use that input in our comparison. For signed operators (e.g., `smax`), we instead use $\texttt{CR}_s$ transformers and similarly skip

cases where LLVM's wrapped interval cannot be captured in $\texttt{CR}_s$. To summarize, we only compare the two domains on inputs that they can both represent. These skipped inputs cause the **Tests** number in both 8-bit and 64-bit in Table 2 to be less than the desired number of sampled inputs.

6.1.2 Evaluation Results. Table 1 and Table 2 summarize `KnownBits` and `ConstantRange` evaluation result respectively. Experiments ran on a server with 2× Intel Xeon Gold 6230 CPUs (40 cores, 80 threads, 2.10GHz). Each benchmark took around 6 hours to finish.

For the `KnownBits` domain, LLVM provides manually written transformers for 37/40 concrete operators. Among these, there are 12 operators for which the LLVM transformers do not achieve the theoretically best result. For 3 out of these 12 cases, `NiceToMeetYou` synthesizes more precise transformers, as measured by the norm function on 64-bit inputs. For another 5 out of 11 cases, the synthesized transformers are less precise than the LLVM ones but still uncover new heuristics that are not present in the LLVM implementations and therefore result in increased precision for the meet of the LLVM and the synthesized transformer. The remaining 28 out of 40 LLVM transformers are already the best, meaning there is no room for improvement via synthesis. That being said, `NiceToMeetYou` is able to match the precision for 3 of these hand-tuned transformers. `NiceToMeetYou` also synthesizes transformers with non-trivial precision for the two operations for which LLVM does not include a transformer.

For the `ConstantRange` domain, LLVM provides manually written transformers for 30/40 concrete operators. `NiceToMeetYou` is able to synthesize transformers for all 40 operators. There are 13 operators where LLVM transformers do not achieve the theoretically best result. For 9 out of these 13 cases, `NiceToMeetYou` synthesizes more precise transformers, as measured by the norm function on 64-bit inputs. For another 6 out of 13 cases, the synthesized transformers are less precise than the LLVM ones but still uncover new heuristics that are not present in the LLVM implementations (again shown by the increased precision obtained when taking the meet of the LLVM and synthesized transformer). The remaining 17 out of 30 LLVM transformers are already the best, meaning there is no room for improvement via synthesis. `NiceToMeetYou` matches existing hand-tuned transformers for 11 of these cases.

6.2 End-to-End Precision and Performance

In this section we evaluate the effect of replacing LLVM's known bits transfer functions with our own. We perform this comparison to show that our transfer functions are reasonable ones, but our larger goal is not to beat LLVM at its own game. Rather, we aim to develop basic technologies that can be used to avoid manual implementation of transfer functions in future compilers.

Replacing LLVM's transfer functions with our own was not entirely a clean software engineering job since LLVM's known bits analysis is implemented in a style that intermixes its transfer functions with a highly ad hoc dataflow framework that simply recurses along backwards dataflow edges until a maximum depth is reached. We left this framework in place, but called out to our own transfer functions, instead of LLVM's, at appropriate points in the code.

We took the nine C/C++, integer programs from the SPEC CPU 2017 benchmark suite and compiled them using an off-the-shelf Clang/LLVM version 21, and then also our modified version. This optimizing compilation (using Clang's `-O3 -march=native` flags) was done on a Linux machine using an AMD 2990WX 32-core CPU, with parallelism disabled in the build system. During an optimizing compile, LLVM uses the known bits analysis results many times, from many different passes. To evaluate precision, we took the final, optimized LLVM IR resulting from optimized compilation and invoked LLVM's known bits analysis—with and without our synthesized transfer functions—on every integer-typed SSA value.

Table 3. Comparison of compilation times and known bits precision between LLVM's own transfer functions and ours, for the nine SPEC CPU 2017 integer benchmarks in C/C++

		perlbench	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	xz
	KLOC	362	1,304	3	134	520	96	10	21	33
Compile time (s)	LLVM	68.39	370.60	1.94	117.82	303.47	42.08	4.78	13.49	11.31
	Ours	69.95	376.80	2.05	117.99	304.61	46.41	5.07	13.66	11.74
	Slowdown	2.27 %	1.67 %	5.49 %	0.14 %	0.38 %	10.29 %	5.89 %	1.22 %	3.80 %
Known bits	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 %	6.92 %	11.79 %	5.12 %	23.22 %	6.26 %

Table 3 shows the results of this experiment. Our transfer functions are neither as fast nor as precise as LLVM's, but the difference is not huge. We have not yet attempted to optimize our transfer functions for runtime performance. One aspect that could be improved is our generated C++, which is not idiomatic: it creates and destroys more temporary objects than does LLVM's hand-written code, and furthermore it performs some translation between MLIR and LLVM data structures that could be elided. Another potential avenue for improvement would be to make execution time part of our fitness function during synthesis.

In addition, we ran a smaller ad-hoc experiment compiling openssl, ffmpeg, and cvc5 using the meet of our synthesized `KnownBits` transformers and LLVM's. The precision improvements are modest: +2 discovered bits in openssl (above the baseline of 1.3M bits discovered by LLVM), +14 bits in ffmpeg (baseline: 3.7M), and +0 bits in cvc5 (baseline: 16M).

6.3 Opportunities Created by `NiceToMeetYou`

We believe that synthesis changes the basic economics of transfer functions, in the sense that we can synthesize more of these than we would want to write by hand.

6.3.1 Precision via Specialization. Production compilers usually provide a wide variety of intrinsic functions that encapsulate higher-level operations such as popcount (Hamming weight) and saturating arithmetic operations. The advantage of intrinsics over open-coded versions of these operations is that the composite versions can readily be lowered to either dedicated machine instructions or optimized library code. However, composite operations have a secondary benefit, which is that a composite transfer function is typically more precise than what we would get by composing the results of the transfer functions for the elementary operations that make up the open-coded version of the operation. To create Table 4, which illustrates this point, we synthesized transfer functions for eight LLVM intrinsics, and then we also measured the precision attained by analyzing an open-coded version of each operation—that is, by composing the most precise transformers that we have been able to synthesize. In all cases, the transfer function for the composed operation is considerably more precise.

6.3.2 Reduced Product. A reduced product domain improves analysis precision by combining the strengths of multiple abstract domains. Each domain captures different aspects of program behavior—for example, `KnownBits` tracks known zero and one bits, while `ConstantRange` tracks possible value ranges. The reduced product coordinates these domains using a reduction operator (σ), which refines each domain's element based on information from the other [6]. This mutual refinement helps eliminate infeasible states and improve the overall precision of the analysis. Since `NiceToMeetYou` synthesizes transformers for all operations in both the `KnownBits` and

Table 4. Results for *top*, *composed*, and *synth* on unary (6,561 test cases) and binary (43,046,721 cases) functions. Exact is reported in % (higher is better), and precision is reported as a normalized count (lower is better).

Category	Concrete Op	Exact (%) ↑			Precision (norm) ↓		
		⊤	composed	synth	⊤	composed	synth
Unary Functions (6,561 test cases)	Abs	1.95	3.92	**100.00**	3372	2552	**0**
	CountRZero	0.00	33.33	**83.63**	5740	3553	**193**
	CountLZero	0.00	0.00	**83.63**	5740	5466	**193**
	PopCount	0.00	0.05	**69.53**	4461	4456	**369**
Binary Functions (43,046,721 test cases)	Smax	4.46	6.33	**56.86**	19,797,600	18,179,000	**5,471,520**
	Smin	4.46	6.04	**70.39**	19,797,600	18,384,100	**4,117,500**
	UaddSat	13.93	22.93	**56.20**	17,358,900	14,111,200	**4,471,530**
	UsubSat	13.93	19.46	**49.11**	17,358,900	15,377,400	**5,369,170**

Table 5. Results for reduced product between `KnownBits` and `ConstantRange` for 8-bit and 64-bit integers. The columns and cells have the same meaning as in Table 1. Only operations for which the reduced product has an improvement over our synthesized `KnownBits` transformer are included.

Concrete Op	Tests	8-bit exact (%) ↑			Tests	64-bit precision (norm) ↓		
		⊤	synth	reduced		⊤	synth	reduced
Abds	1000	7.64	18.06	**28.98**	10000	0.1260	0.1119	**0.1069**
Abdu	1000	7.11	20.76	**71.06**	10000	0.1236	0.1085	**0.0921**
AddNsw	1000	6.68	18.73	**81.65**	10000	0.2820	0.1125	**0.0869**
AddNswNuw	1000	0.22	44.89	**88.20**	10000	0.5592	0.1216	**0.0610**
AddNuw	1000	3.35	31.80	**92.32**	10000	0.4920	0.0930	**0.0557**
AvgCeilS	1000	9.83	18.01	**29.16**	10000	0.1651	0.1573	**0.1503**
AvgFloorS	1000	9.86	17.37	**46.61**	10000	0.1669	0.1598	**0.1412**
AvgFloorU	1000	9.90	19.00	**50.37**	10000	0.1668	0.1481	**0.1336**
Sdiv	1000	17.99	27.85	**45.61**	10000	0.7262	0.2493	**0.2229**
Smax	1000	0.44	59.89	**83.19**	10000	0.4959	0.0926	**0.0822**
Smin	1000	0.43	59.62	**84.23**	10000	0.4954	0.0953	**0.0843**
Srem	1000	13.14	22.41	**26.92**	10000	0.1845	0.1701	**0.1689**
SshlSat	1000	4.08	33.43	**43.35**	10000	0.9542	0.3211	**0.3148**
SubNswNuw	1000	0.33	39.01	**77.20**	10000	0.5617	0.1299	**0.0701**
SubNuw	1000	3.52	36.84	**90.94**	10000	0.4822	0.0763	**0.0466**
UaddSat	1000	4.11	61.03	**83.09**	10000	0.4482	0.0874	**0.0469**
Udiv	1000	0.00	68.66	**75.28**	10000	0.9845	0.0134	**0.0067**
UdivExact	1000	0.02	3.21	**5.78**	10000	1.0000	0.0272	**0.0195**
Umax	1000	0.54	95.28	**99.74**	10000	0.4947	0.0016	**0.0001**
Umin	1000	0.56	92.99	**99.59**	10000	0.4964	0.0023	**0.0003**
Urem	1000	2.12	61.45	**66.53**	10000	0.2677	0.0393	**0.0367**
UsubSat	1000	4.06	56.03	**73.09**	10000	0.4508	0.1106	**0.0700**

`ConstantRange` domains, we can automatically construct reduced product transformers for any concrete operation by manually providing a suitable reduction operator that relates the two domains.

We evaluate the difference in precision between using `KnownBits` and using the reduced product between `KnownBits` and `ConstantRange`. To have a uniform random sampling from this reduced product lattice, we simply sample uniformly from both `KnownBits` and `ConstantRange`, where the reduction between the `KnownBits` and `ConstantRange` abstract values is not bottom.

We compare only against our synthesized transformers, not LLVM's, because our sampling occurs over the product lattice. This means the abstract elements we evaluate are inherently more precise than those representable by `KnownBits` alone, making a direct comparison with LLVM's `KnownBits` transformers unfair. Instead, we focus on identifying which concrete operations benefit from the added precision of the reduced product domain, and by how much, when using our synthesized transformers.

Table 5 presents our results. The experiment finished quickly (90 seconds, on a reasonably fast laptop) because it merely evaluates transformers rather than synthesizing transformers. We report only the transformers for which the reduced product yields a precision improvement over using `KnownBits` alone. On the remaining operations, such as `Xor` (not shown in Table 5), the reduced product offers no additional benefit, as `KnownBits` is already maximally precise. In contrast, operations such as `AddNswNuw` show substantial gains in precision when evaluated within the reduced product domain. There were 22 such concrete operations for which the reduced product was able to improve over the `KnownBits` domain, out of 40 total operations. For the concrete operations where there was an improvement, the reduced product improved by an average of 24.4% points for the tests at 8-bits wide, and improved the average size score by 196.2, on the tests at 64-bits wide. By combining `KnownBits` with `ConstantRange` in a reduced product, we are able to improve precision particularly for operators that are challenging for both our synthesized `KnownBits` transformers and LLVM's. Overall, the reduced product delivers consistently strong performance across the full range of supported operations, thus showcasing one additional benefit enabled by `NiceToMeetYou`.

6.4 Impact of DSL Choice

We evaluate how the choice of operations in DSL affects the performance of `NiceToMeetYou`. The **Full** DSL used in Section 6.1 includes 29 primitives, which can be grouped as follows: (1) *Bitwise:* `and`, `or`, `xor`, `neg`; (2) *Add:* `add`, `sub`. (3) *Max:* `umax`, `umin`, `smax`, `smin`; (4) *Mul:* `mul`, `udiv`, `sdiv`, `urem`, `srem`; (5) *Shift:* `shl`, `ashr`, `lshr`; (6) *BitSet:* `set_high_bits`, `set_low_bits`, `clear_low_bits`, `clear_high_bits`, `set_sign_bit`, `clear_sign_bit`; (7) *BitCount:* `count_left_one`, `count_left_zero`, `count_right_one`, `count_right_zero`. (8) *ITE:* `if_then_else`. We conduct an ablation study on the `KnownBits` domain over two DSL subsets: **Basic** =*Bitwise* ∪ *Add*, contains only addition, subtraction, and bitwise operations. **BitExt** =*Bitwise* ∪ *Add* ∪ *Max* ∪ *Shift* ∪ *BitSet* ∪ *BitCount* ∪ *ITE*, contains all primitives except those in *Mul*.

The **Full** language, **Basic** , and **BitExt** yield the most precise transformers for 16, 5, and 14 operations, respectively. For the remaining 5/40 benchmarks, they all reach the same precision. Detailed results are summarized in Table 6 in Appendix B.1.

The results are in a way expected: if certain operations are known to be irrelevant for a specific transformer, removing them from the DSL can improve the precision—e.g., **Basic** is the smallest of the three languages that can produce an optimal transformer for `add` and `sub` and does well for such benchmarks.

6.5 Impact of Abduction

We now evaluate the impact of condition abduction (Section 4.1.2) by running `NiceToMeetYou` with and without condition abduction. For `KnownBits`, condition abduction improves precision by 6.44% on average (geometric mean), with 19/40 benchmarks showing gains. For `ConstantRange`, condition abduction improves precision by 2.3% on average (geometric mean), with 16/37 benchmarks showing gains. Detailed Results are shown in Tables 7 and 8 in Appendix B.2.

A representative case highlighting the necessity of abduction is the `add` benchmark in the $\mathtt{CR}_u$ domain. Its best transformers produce intervals better than $\top$ only when one knows overflow cannot happen. Using condition abduction, `NiceToMeetYou` synthesizes the best transformer in 2 rounds: first, it discovers a transformer that only works for non-overflow cases, and then it identifies the overflow condition to generate a complete transformer. When abduction is disabled, `NiceToMeetYou` cannot find the best transformer within 5 rounds.

To summarize, overall condition abduction generally improves precision, but because it uses part of the compute budget, it may in certain cases reduce precision. Of course, one can also run `NiceToMeetYou` with and without abduction and output the better result.

6.6 Comparison to Amurth

The only other work we are aware of that tackles the general problem of synthesizing abstract transformers is Amurth [15]. Amurth takes the same inputs as `NiceToMeetYou`—i.e., a DSL, a concrete semantics, and an abstract domain—and synthesizes a provably most precise (up to a given input bound) transformer expressible within the given DSL. To do so, Amurth uses constraint solvers (specifically Sketch [32]) to iteratively synthesize transformers that increase precision on a finite set of abstract inputs until a provably optimal transformer is sound.

The evaluation by Kalita et al. includes transformers for both string operations and integer operations. Because `NiceToMeetYou` currently does not support string operations, we only focus on the latter. Amurth has successfully been used to synthesize transformers for 9 concrete operators for the unsigned and signed interval domains Kalita et al. [15, Table 4]: `add`, `sub`, `mul`, `and`, `or`, `xor`, `shl`, `ashr`, and `lshr`. We therefore focus our evaluation on these 2 domains and 9 operations.

When provided with this DSL consisting of the set of 29 primitive instructions we used in Section 6.4, Amurth could not synthesize any transformer or returned $\top$ within the time limit.

While Amurth cannot synthesize abstract transformers when given a generic DSL, it can do so by providing "hints" to the synthesizer in the form of sketches—i.e., partial programs where only some parts are missing—and custom auxiliary functions. For example, Amurth synthesizes transformers for bitwise operators (`and`, `or`, `xor`), when auxiliary functions such as `minOr`, `maxOr`, `minAnd`, `maxAnd` (which compute the lower/upper bound of the results of bitwise-or/and over 2 intervals) are provided. With those auxiliary functions provided, they further provide program template (that describes a clever way to divide input intervals at 0) to synthesize for signed domains Kalita et al. [15, Figure 16]. Providing hints and templates allows Amurth to synthesize most-precise transformers for very tricky transformers operations, but requires the users of Amurth to provide insights that are quite close to the actual solution. Moreover, one has to provide Amurth with different hints and sketches (i.e., different DSLs) for different concrete operations, even when the underling abstract domain does not change.

To summarize, Amurth cannot solve the problem tackled in this paper—i.e., synthesize abstract transformers for many concrete operators using just *one* given DSL. However, Amurth is well-suited for synthesizing optimal transformers for tricky individual operations, as long as the user is willing to provide hints to the synthesizer in the form of program sketches and auxiliary functions.

7 Other Related Work

Our approach draws on a rich line of work on synthesizing abstract transformers, verification infrastructure, and stochastic program synthesis. We build on the MLIR ecosystem [10], expressing synthesized abstract transformers in a first-class dialect that supports both efficient compilation via LLVM and formal reasoning via SMT encoding, using Z3 [8] for soundness verification. We have already discussed at length how `NiceToMeetYou` relates to its closest related work, Amurth [15, 16], in Section 6.6, and use the rest of the section to discuss other approaches.

Precision-Oriented Synthesis and Domain-Specific Approaches. Several efforts have addressed the problem of deriving best or approximate abstract transformers. A line of work by Reps, Sagiv, Yorsh, and Thakur [28, 29, 33–35] develops methods for automated symbolic abstraction—computing the best abstract transformer for a given input. In these approaches, the transformer is not necessarily an explicit, executable program. Other works have focused on finding executable representations of abstract transformers, but are typically tied to specific abstract domains or specific representations. For example, Regehr and Reid [27] encode transformers using BDDs, which can sometimes be inefficient due to the limited expressivity of BDDs. Elder et al. [9] target conjunctions of bit-vector equalities, Laurel et al. [17] target numerical abstract domains such as linear convex polytope, and Vishwanathan et al. [36] target interval analysis in the eBPF verifier. In contrast, `NiceToMeetYou` supports a wide range of instructions for composing abstract transformers and remains agnostic to specific domains. More recent work proposes sketching-based algorithms for learning disjunctive and conjunctive specifications over program behaviors [23, 24]. While our work shares the idea of synthesizing multiple components and combining them (via meet), we instantiate it in the domain of abstract interpretation with formal guarantees and no sketching.

Stochastic and MCMC-Based Synthesis. Our synthesis algorithm is inspired by stochastic search techniques, in particular the Markov Chain Monte Carlo (MCMC) superoptimization strategy introduced by Stoke [31]. Like Stoke, our framework searches the space of candidate programs guided by a cost function, using probabilistic rewrites. Unlike Stoke, which targets concrete program optimization, we use MCMC to synthesize abstract transformers, and our cost function encodes the precision improvement from existing transformers. Moreover, our work introduces a novel abductive refinement strategy that iteratively improves precision by synthesizing and composing multiple sound transformers. Instantiating this algorithm over the full LLVM instruction set via MLIR requires significant engineering and forms a core contribution of our work.

8 Conclusion

Abstract transformers are a load-bearing component of a modern optimizing compiler: the compiler will miss optimizations if transformers are imprecise, and it will miscompile if they are unsound. We created `NiceToMeetYou`: a framework for synthesizing formally verified abstract transformers from specifications of integer IR instructions and finite non-relational abstract domains. Unlike previous systems, ours does not require any sketches—transformers are synthesized from scratch—and can therefore quickly synthesize transformers for dozens of operators. The insight that made this possible is that transformers can be synthesized piecewise, with each new piece targeting a different part of the input space. The final transformer is simply the meet of its constituent pieces. In our evaluation, `NiceToMeetYou` synthesized transformers for most LLVM operations with precision sometimes comparable to LLVM's manually written transformers. `NiceToMeetYou` also synthesized 26 transformers that are either more precise than LLVM's or can be combined with LLVM ones via a meet operation to yield new transformers with greatly increased precision—i.e., `NiceToMeetYou`'s transformers deal with corner cases that had eluded LLVM developers.

Data-Availability Statement

`NiceToMeetYou` and scripts for reproducing our experiments are available on Zenodo [25].

Acknowledgments

Kennedy was supported by a seed grant from the University of Utah. D'Antoni and Peng are supported in part by a Microsoft Faculty Fellowship; a UCSD JSOE Scholarship; and NSF under grants CCF-2422214, CCF-2506134 and CCF-2446711. Fan and Regehr were supported in part by

the National Science Foundation under grant under Grant No. 1955688. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities. Loris D'Antoni holds concurrent appointments as a Professor at the University of California San Diego and as an Amazon Scholar. This paper describes work performed at the University of California San Diego and is not associated with Amazon.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2010. The SMT-LIB standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories*, Vol. 13. 14. https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-02-05.pdf.

[2] Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. 2023. A Formal Framework to Measure the Incompleteness of Abstract Interpretations. In *SAS*. Springer, 114–138. doi:10.1007/978-3-031-44245-2_7

[3] Ignacio Casso, José F. Morales, Pedro López-García, Roberto Giacobazzi, and Manuel V. Hermenegildo. 2020. Computing Abstract Distances in Logic Programs. In *Logic-Based Program Synthesis and Transformation*. Springer, 57–72. doi:10.1007/978-3-030-45260-5_4

[4] Compiler Research at The University of Cambridge. 2025. xdsl-smt. https://github.com/opencompl/xdsl-smt, Accessed 2025-11-25.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. doi:10.1145/512950.512973

[6] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM, 269–282. doi:10.1145/567752.567778

[7] Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming*. Springer, 269–295. doi:10.1007/3-540-55844-6_142

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340. doi:10.1007/978-3-540-78800-3_24

[9] Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas W. Reps. 2014. Abstract Domains of Affine Relations. *Transactions on Programming Languages and Systems* 36, 4 (2014), 11:1–11:73. doi:10.1145/2651361

[10] Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proceedings of the ACM on Programming Languages* 9, PLDI, Article 206 (2025), 25 pages. doi:10.1145/3729309

[11] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *Transactions on Programming Languages and Systems* 37, 1, Article 1 (2015), 35 pages. doi:10.1145/2651360

[12] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (04 1970), 97–109. doi:10.1093/biomet/57.1.97

[13] Hatsunespica and OpenCompl Contributors. 2025. xdsl-smt: Artifact Branch. https://github.com/Hatsunespica/xdsl-smt/tree/artifact. Accessed: 2025-10-23.

[14] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *CAV*. Springer, 335–352. doi:10.1007/978-3-030-25540-4_18

[15] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas W. Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proceedings of the ACM on Programming Languages* OOPSLA (2022). doi:10.1145/3563334

[16] Pankaj Kumar Kalita, Thomas Reps, and Subhajit Roy. 2024. Synthesizing Abstract Transformers for Reduced-Product Domains. In *SAS*. Springer-Verlag, Berlin, Heidelberg, 147–172. doi:10.1007/978-3-031-74776-2_6

[17] Jacob Laurel, Ignacio Laguna, and Jan Hückelheim. 2025. Synthesizing Sound and Precise Abstract Transformers for Nonlinear Hyperbolic PDE Solvers. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 310 (2025), 29 pages. doi:10.1145/3763088

[18] Nick Lewycky. 2015. Miscompile of % in loop. https://bugs.llvm.org/show_bug.cgi?id=23011, Accessed 2025-11-25.

[19] LLVM Contributors. 2025. LLJIT Class Reference. https://llvm.org/doxygen/classllvm_1_1orc_1_1LLJIT.html, Accessed 2025-11-25.

[20] Francesco Logozzo. 2009. Towards a Quantitative Estimation of Abstract Interpretations. In *Workshop on Quantitative Analysis of Software*. Microsoft. https://www.microsoft.com/en-us/research/publication/towards-a-quantitative-estimation-of-abstract-interpretations/

[21] OpenCompl Contributors. 2025. Pull Request #73: xdsl-smt. https://github.com/opencompl/xdsl-smt/pull/73. Accessed: 2025-10-23.

[22] OpenCompl Contributors. 2025. Pull Request #74: xdsl-smt. https://github.com/opencompl/xdsl-smt/pull/74. Accessed: 2025-10-23.

[23] Kanghee Park, Loris D'Antoni, and Thomas W. Reps. 2023. Synthesizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1787–1816. doi:10.1145/3622861
[24] Kanghee Park, Xuanyu Peng, and Loris D'Antoni. 2025. LOUD: Synthesizing Strongest and Weakest Specifications. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1, Article 114 (2025), 28 pages. doi:10.1145/3720470
[25] Xuanyu Peng, Dominic Kennedy, Yuyou Fan, Ben Greenman, John Regehr, and Loris D'Antoni. 2025. *Artifact for Nice to Meet You: Synthesizing Practical Abstract Transformers for MLIR.* doi:10.5281/zenodo.17371668 Version v2.
[26] John Regehr. 2012. Wrong code bug. https://bugs.llvm.org/show_bug.cgi?id=12541, Accessed 2025-11-25.
[27] John Regehr and Alastair Reid. 2004. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. In *ASPLOS*. ACM, 133–143. doi:10.1145/1024393.1024410
[28] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *VMCAI*. 252–266. doi:10.1007/978-3-540-24622-0_21
[29] Thomas W. Reps and Aditya V. Thakur. 2016. Automating Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 3–40. doi:10.1007/978-3-662-49122-5_1
[30] Erika Rice Scherpelz, Sorin Lerner, and Craig Chambers. 2007. Automatic Inference of Optimizer Flow Functions from Semantic Meanings. In *PLDI*. ACM, 135–145. doi:10.1145/1250734.1250750
[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *ASPLOS*. ACM, 305–316. doi:10.1145/2451116.2451150
[32] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495. doi:10.1007/s10009-012-0249-7
[33] Aditya V. Thakur, Matt Elder, and Thomas W. Reps. 2012. Bilateral Algorithms for Symbolic Abstraction. In *SAS*. 111–128. doi:10.1007/978-3-642-33125-1_10
[34] Aditya V. Thakur, Akash Lal, Junghee Lim, and Thomas W. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* 311 (2015), 15–32. doi:10.1016/j.entcs.2015.02.003
[35] Aditya V. Thakur and Thomas W. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *CAV*. 174–192. doi:10.1007/978-3-642-31424-7_17
[36] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2025. Automatic Synthesis of Abstract Operators for eBPF. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions* (Coimbra, Portugal) *(eBPF '25)*. Association for Computing Machinery, New York, NY, USA, 15–21. doi:10.1145/3748355.3748361

A Detailed Comparison to Amurth

Kalita et al. used Amurth to synthesize transformers for both string domains and fixed-bitwidth integer domains. We focus our comparison on integer domains, as `NiceToMeetYou` currently supports only integer domains. Amurth synthesized transformers for 9 concrete operators for unsigned and signed interval domains: `add`, `sub`, `mul`, `and`, `or`, `xor`, `shl`, `ashr`, and `lshr` [15, Table 4]. These operators are also featured in our benchmark set. Amurth successfully synthesized the best transformers for each of these operators within 30 minutes, whereas `NiceToMeetYou` synthesized the best transformers only for `add`, `sub`, and `ashr`.

However, the reason Amurth performs well is that it requires users to supply program templates and auxiliary functions, which serve as powerful hints to guide synthesis. Moreover, these hints differ across synthesis tasks. When Amurth is restricted to the same setting as `NiceToMeetYou` —i.e., using only the base DSL operators and no templates—it times out on all benchmarks for both unsigned and signed interval domains. In the following case studies, we examine several examples of such hints from the Amurth benchmark suite and illustrate why Amurth is not suited to automate the synthesis of many transformers in a compiler.

Amurth Requires Sketches or Templates. Users of Amurth typically need to provide a program sketch or template of the desired solution, even though the theoretical framework supporting Amurth is parametric in the choice of DSL. The sketches are often hand-derived from existing manual implementations (noted in Kalita et al. [15, Table 2 and Table 3]).

For instance, the template used to synthesize the transformer for `xor` Kalita et al. [15, Figure 16] in the signed domain begins with a helper function `splitAtZero`, which divides each input interval

at 0 if it crosses 0. The core synthesis task then fills in the logic that transforms the resulting (up to 2*2=4) pairs of same-signed intervals, before the template finally joins those outputs into the final interval. While the template is effective, both the `splitAtZero` helper and the looping pattern over split-interval pairs are non-trivial to synthesize from scratch.

In contrast, our candidate program fixes only a minimal structure: it deconstructs each input interval into two integers at the beginning and reconstructs the output interval from two integers at the end. The middle portion is a free-form SSA program, unconstrained in its instruction dependencies.

Additionally, Amurth does not synthesize boundary conditions to handle overflow [15, Section 6.2.3]. It synthesizes the non-overflow case, and such an assumption can be seen as an implicit template. For example, consider the best transformer for addition in the unsigned interval domain, shown in Figure 5a. Our approach synthesized an equivalent transformer. In contrast, Amurth synthesized only the else branch $[a.l + b.l,\ a.r + b.r]$, as it assumes its output will be plugged into a sketch that checks for overflow. Such an assumption is equivalent to $(a.l + b.l > \texttt{MAX\_INT})$ **or** $(a.r + b.r > \texttt{MAX\_INT})$. This condition is weaker than the one in our transformer because it means an overflow for additions of either the left **or** right endpoints (which should be an **xor**). As a result, their no-overflow assumption not only reduces the difficulty of the synthesis, but also loses precision on the overflow cases.

$\text{add}^\sharp(a, b) :=$
  if $(a.l + b.l > \texttt{MAX\_INT})$ **xor** $(a.r + b.r > \texttt{MAX\_INT})$
  then $[0, \texttt{MAX\_INT}]$
  else $[a.l + b.l,\ a.r + b.r]$

(a) The best transformer for the add operator in CR_u domain

```
KnownBits add#(KnownBits L, R) {
  APInt l0 = L.Zero, l1 = L.One;
  APInt r0 = r.Zero, r1 = R.One;
  APInt E = (l0 | l1) & (r0 | r1) &
    (~((~l0 + ~r0) ^ l0 ^ r0) |
    ((l1 + r1) ^ l1 ^ r1));
  APInt known0 = ~(~l0 + ~r0) & E;
  APInt known1 = (l1 + r1) & E;
  return {known0, known1};
}
```

(b) The best transformer for the add operator in the `KnownBits` domain

Fig. 5. Optimal transformers

Amurth *relies on auxiliary functions.* This limitation is reflected in [15, Section 6.2.4] and we confirmed it by examining the Amurth codebase. For example, to synthesize transformers for bitwise operators (`and`, `or`, `xor`), auxiliary functions such as `minOr`, `maxOr`, `minAnd`, `maxAnd` (which compute the lower/upper bound of the results of bitwise-or/and over 2 intervals) are provided.

These auxiliary functions are non-trivial, consisting of about 25 operators and involving branching and loops, and make the synthesis tasks much easier. Even with the hints above, Amurth still needs advanced program sketches for several harder benchmarks.

Hints vary across synthesis tasks. When provided with enough structure and templates, Amurth can directly synthesize optimal transformers. However, the supporting DSL is typically crafted and modified individually for synthesis tasks of each concrete operator. For example, even within the same abstract domain, bitwise operators, arithmetic operators, and shifting operators *each rely on a distinct DSL*. These DSLs consist of nearly disjoint sets of operations.

In contrast, our approach employs a single, unified DSL shared across all concrete operators. It consists of 29 basic numeric operations that can express a wide variety of transformers. Details of the DSL's design and its operations are discussed in Section 6.4.

Table 6. KnownBits results under DSLs with different operation subsets. Each value denotes the percentage of tests where the synthesized transformer matches the best transformer $\widehat{f^{\#}}$ at 8-bit precision. **Bolded** numbers denote the configuration that yields the most precise result

	Abds	Abdu	Add	AddNsw	AddNswNuw	AddNuw	And	Ashr	AshrExact	AvgCeilS	AvgCeilU	AvgFloorS	AvgFloorU	Lshr	LshrExact	Mods	Modu	Mul	Or	Sdiv
Full	60.1	**59.4**	58.7	42.0	45.5	**53.9**	**100.0**	**65.5**	**40.1**	38.8	38.6	**39.3**	37.7	**59.3**	31.4	**64.7**	**59.0**	60.6	**100.0**	**64.1**
Basic	54.7	53.3	**100.0**	**78.0**	12.9	37.9	100.0	31.2	30.0	31.8	31.6	32.4	32.2	12.6	28.1	48.6	39.3	**74.4**	100.0	64.1
BitExt	**60.3**	59.3	51.8	51.3	**49.0**	49.5	100.0	37.8	36.6	**39.4**	**40.9**	39.0	**40.5**	31.0	**31.6**	60.4	58.4	62.1	100.0	64.1

	SdivExact	Shl	ShlNsw	ShlNswNuw	ShlNuw	Smax	Smin	SshlSat	Sub	SubNsw	SubNswNuw	SubNuw	UaddSat	Udiv	UdivExact	Umax	Umin	UshlSat	UsubSat	Xor
Full	19.3	**56.9**	26.2	9.8	40.5	63.8	**72.8**	**72.4**	60.6	47.8	**31.8**	**47.1**	61.8	**80.8**	15.3	90.6	**92.9**	**96.6**	52.1	**100.0**
Basic	19.3	13.2	6.9	5.6	10.7	57.8	51.9	37.6	**100.0**	**72.3**	12.0	42.4	42.5	24.1	2.9	71.4	72.2	3.7	48.2	100.0
BitExt	19.3	56.6	**27.9**	**34.9**	**46.9**	**71.0**	56.1	45.0	61.5	51.4	28.5	45.0	**67.4**	39.9	**15.4**	**93.3**	88.3	96.6	**56.9**	100.0

Summary. To summarize, one cannot use Amurth to solve the problem solved by NiceToMeetYou, i.e., automatically synthesizing transformers for many operators at once without manually tuning the underlying DSL or providing strong hints in the form of sketches. While Amurth does appear to be better than NiceToMeetYou for the task of finding tricky transformers for individual operators, it needs guidance from human experts. In practice, for an abstract domain, there could be hundreds of operators that need an abstract transformer, and providing tailored hints for each of them is infeasible. Hence, we believe that Amurth cannot be used to automate large-scale synthesis of transformers in a compiler.

B Ablation Study

B.1 Impact of DSL Choice

We evaluate how the choice of operations in DSL affects the performance of NiceToMeetYou. As mentioned in Section 6.4, we run synthesis over the **Full** language and 2 subsets (**Basic** and **BitExt**). Detailed results are summarized in Table 6.

Basic is the DSL with the most limited expressivity and causes substantial precision loss in many benchmarks, though it produces optimal transformers for add and sub. Figure 5b shows a reference implementation of the add transformer in KnownBits, which uses only operations from **Basic** but remains fairly complex. Nevertheless, NiceToMeetYou successfully synthesizes a transformer equivalent to this implementation when the DSL is limited to **Basic** .

BitExt is the DSL variant that excludes multiplicative operations. Since the transformers for shifting operators (e.g., shl) rarely rely on multiplication and division, synthesis with **BitExt** achieves slightly higher precision for these operators compared to using the full DSL.

In conclusion, if certain operations are known to be irrelevant for a specific transformer, removing them from the DSL can improve the precision.

B.2 Impact of Condition Abduction

In this section, we evaluate the impact of condition abduction (Section 4.1.2). We run synthesis with and without condition abduction on both the KnownBits and ConstantRange domains. For KnownBits, condition abduction improves precision by 6.44% on average (geometric mean) across

all benchmarks, with 18/39 benchmarks showing gains. For `ConstantRange`, condition abduction improves precision by 2.3% on average (geometric mean) across all benchmarks, with 16/39 benchmarks showing gains. Results are shown in Tables 7 and 8.

A representative case highlighting the necessity of abduction is the `add` benchmark in the $\mathtt{CR}_u$ domain. Its best transformers, shown in Figure 5a, produce intervals better than $\top$ only when whether overflow happens is known. When inspecting the synthesis process, we observe that `NiceToMeetYou` synthesized the best transformer in 2 rounds: in the first round, it discovered a transformer that only works for non-overflow cases and stored it as one of the unsound but highly precise candidates; in the second round, it successfully identified the overflow condition the complete the full transformers. However, when abduction is disabled, `NiceToMeetYou` failed to synthesize the best transformer within 5 rounds.

To summarize, condition abduction generally improves precision, but may sometimes reduce it since condition abduction reuses part of the parallel search budget. However, one can also run `NiceToMeetYou` with and without abduction and selecting the better result.

Table 7. `KnownBits` results with and without abduction. Each value denotes the percentage of tests where the synthesized transformer matches the best transformer $\widehat{f^{\#}}$ at 8-bit precision.

	Abds	Abdu	Add	AddNsw	AddNswNuw	AddNuw	And	Ashr	AshrExact	AvgCeilS	AvgCeilU	AvgFloorS	AvgFloorU	Lshr	LshrExact	Mods	Modu	Mul	Or	Sdiv
Abd	60.1	59.4	58.7	42.0	45.5	**53.9**	**100.0**	**65.5**	**40.1**	**38.8**	**38.6**	39.3	37.7	**59.3**	**31.4**	**64.7**	**59.0**	60.6	**100.0**	64.1
No Abd	**60.7**	**61.7**	**63.4**	**48.8**	**50.5**	40.3	100.0	49.3	25.0	38.8	38.4	**40.3**	**40.2**	59.3	24.4	64.7	44.6	**64.3**	100.0	**67.8**

	SdivExact	Shl	ShlNsw	ShlNswNuw	ShlNuw	Smax	Smin	SshlSat	Sub	SubNsw	SubNswNuw	SubNuw	UaddSat	Udiv	UdivExact	Umax	Umin	UshlSat	UsubSat	Xor
Abd	**19.3**	**56.9**	26.2	9.8	**40.5**	63.8	**72.8**	**72.4**	**60.6**	47.8	**31.8**	47.1	61.8	**80.8**	15.3	**90.6**	**92.9**	**96.6**	**52.1**	**100.0**
No Abd	19.3	55.5	**27.6**	**41.8**	25.4	**67.0**	54.3	65.7	53.5	**57.8**	14.4	**50.0**	**63.4**	79.8	**17.7**	73.2	62.1	92.9	19.1	100.0

Table 8. `ConstantRange` results with and without abduction. The cells have the same meaning as in Table 7.

	Abds	Abdu	Add	AddNsw	AddNswNuw	AddNuw	And	Ashr	AshrExact	AvgCeilS	AvgCeilU	AvgFloorS	AvgFloorU	Lshr	LshrExact	Mods	Modu	Mul	Or
Abd	**59.8**	**75.0**	**100.0**	**100.0**	**70.7**	**94.0**	**82.3**	**98.0**	**81.7**	0.3	1.8	1.8	30.0	**100.0**	**85.2**	43.5	**89.0**	**90.8**	82.4
No Abd	59.8	67.2	36.5	68.6	61.5	81.7	76.3	95.3	81.0	**0.7**	**2.9**	**2.0**	**67.8**	100.0	72.5	**43.9**	89.0	90.8	**82.5**

	Sdiv	Shl	ShlNswNuw	ShlNuw	Smax	Smin	SshlSat	Sub	SubNsw	SubNswNuw	SubNuw	UaddSat	Udiv	Umax	Umin	UshlSat	UsubSat	Xor
Abd	**12.3**	**83.0**	**88.3**	76.4	**100.0**	**100.0**	**75.5**	**100.0**	**100.0**	**74.5**	94.5	**99.2**	31.7	**100.0**	**100.0**	**100.0**	**100.0**	56.1
No Abd	3.7	83.0	88.3	**87.7**	100.0	100.0	75.2	35.4	70.1	59.7	**100.0**	85.6	**40.5**	100.0	100.0	98.6	98.8	**60.6**

Received 2025-07-10; accepted 2025-11-06