Synthesizing Abstract Transformers

PANKAJ KUMAR KALITA, Indian Institute of Technology Kanpur, India
SUJIT KUMAR MUDULI, Indian Institute of Technology Kanpur, India
LORIS D'ANTONI, University of Wisconsin–Madison, USA
THOMAS REPS, University of Wisconsin–Madison, USA
SUBHAJIT ROY, Indian Institute of Technology Kanpur, India

This paper addresses the problem of creating abstract transformers automatically. The method we present automates the construction of static analyzers in a fashion similar to the way yacc automates the construction of parsers. Our method treats the problem as a program-synthesis problem. The user provides specifications of (i) the concrete semantics of a given operation *op*, (ii) the abstract domain A to be used by the analyzer, and (iii) the semantics of a domain-specific language L in which the abstract transformer is to be expressed. As output, our method creates an abstract transformer for *op* in abstract domain A, expressed in L (an "L-transformer for *op* over A"). Moreover, the abstract transformer obtained is a *most-precise* L-transformer for *op* over A; that is, there is no other L-transformer for *op* over A that is strictly more precise.

We implemented our method in a tool called Amurth. We used Amurth to create sets of replacement abstract transformers for those used in two existing analyzers, and obtained essentially identical performance. However, when we compared the existing transformers with the transformers obtained using Amurth, we discovered that four of the existing transformers were unsound, which demonstrates the risk of using manually created transformers.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Automated reasoning**; **Abstraction**.

Additional Key Words and Phrases: abstract transformer, program synthesis, DSL

ACM Reference Format:
Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 171 (October 2022), 29 pages. https://doi.org/10.1145/3563334

1 INTRODUCTION

Abstract interpretation is a methodology for establishing whether a software system satisfies desired properties. It obtains information about the states that a program (possibly) reaches during execution, without actually running the program on specific inputs. Instead, the program's behavior is explored for all possible inputs, and all possible states that the program can reach, by running the program over *abstract values*—descriptors that represent sets of states. Each operation of the program is interpreted over abstract values in a manner that overapproximates the operation's standard ("concrete") interpretation over the corresponding sets of concrete states. Such an interpretation of

Authors' addresses: Pankaj Kumar Kalita, Indian Institute of Technology Kanpur, Uttar Pradesh, India, pkalita@cse.iitk.ac.in; Sujit Kumar Muduli, Indian Institute of Technology Kanpur, Uttar Pradesh, India, smuduli@cse.iitk.ac.in; Loris D'Antoni, University of Wisconsin–Madison, USA, ldantoni@wisc.edu; Thomas Reps, University of Wisconsin–Madison, USA, reps@cs.wisc.edu; Subhajit Roy, Indian Institute of Technology Kanpur, Uttar Pradesh, India, subhajit@cse.iitk.ac.in.

2475-1421/2022/10-ART171
https://doi.org/10.1145/3563334

operation *op* is called an *abstract transformer* for *op* (denoted by $op^\sharp$). In particular, the result of applying an abstract transformer for a statement must result in an abstract value that represents a superset of the concrete states that can actually arise.

Cousot and Cousot [1977] showed that, under reasonable conditions, for each operation *op*, there exists a *most-precise abstract transformer* $op^\sharp$ (or "best transformer for *op*"). When the power set of concrete states $\mathcal{P}(C)$ is related to the set of abstract values (*abstract domain* A) by a Galois connection $\mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, the best transformer for *op* is the function

$$\widehat{op}^\sharp = \alpha \circ \widetilde{op} \circ \gamma, \tag{1}$$

where $\widetilde{op}$ is the lifting of *op* to sets of concrete states. Informally, $\widehat{op}$ runs *op* on all the states represented by the input abstract value. Eqn. (1) defines the limit of precision obtainable using abstraction function α and concretization function γ. However, Eqn. (1) is just a *specification* of the best transformer: it fails to provide an *algorithm* for either

(a) applying $\widehat{op}^\sharp$ to a given abstract value, or

(b) finding a representation of $\widehat{op}^\sharp$.

A "representation" means either (i) a data structure whose interpretation is $\widehat{op}^\sharp$, or (ii) a program to perform $\widehat{op}^\sharp$.

Prior work on algorithms for (a) and (b) can be categorized as follows. For (a), Graf and Saidi [1997] showed that for *predicate-abstraction domains* (i.e, domains based on a fixed, finite set of state predicates), SMT solvers can be used to apply the best transformer to an abstract value. (Improved techniques were given by Lahiri et al. [2005, 2006].) Reps et al. [2004] showed that for *abstract domains with no infinite ascending chains*, SMT solvers could be used to apply the best transformer to an abstract value. The drawback of these methods is that they generally require making a large number of SMT calls—at least one SMT call for each used abstract value.

For (b), Scherpelz et al. [2007] gave a method for creating a representation of an abstract transformer for predicate-abstraction domains. Because their method was based on term-rewriting heuristics, they had no guarantee of obtaining a representation of the best abstract transformer. Elder et al. [2014] gave a method for creating a representation of a best abstract transformer for the abstract domain of *conjunctions of bit-vector equalities*. That method uses calls on an SMT solver at the time a transformer is created for a statement, basic-block, or large-block encoding [Beyer et al. 2009]; thereafter, all operations are carried out within the abstract domain.

The advantage of methods of type (b) is that they compile the abstract transformer to a form that can be used—i.e., applied or composed—without further incurring any expensive operations, such as SMT calls; all expensive operations are performed once and for all at *compilation time*.

Thanks to our synthesis-based approach—and in contrast with previous work—the work described in this paper is not limited to predicate abstraction or bit-vector equalities. However, our work addresses a slightly different problem from prior work: our method is parameterized by a domain-specific language (DSL) L in which the abstract transformer for operation *op* is to be expressed. Given *op* and abstract domain A, our method creates an abstract transformer for *op* over A, *expressed in DSL* L—what we call an "L-transformer (for *op* over A)." Our algorithm is guaranteed to return a *best* L-transformer. That is, among all L-transformers for *op* over A, there is no other L-transformer that is strictly more precise than the one obtained by our algorithm. (There may be other L-transformers that are incomparable to the one obtained by our algorithm, which is why we say that the algorithm creates *a* best L-transformer.)

Thanks to our automated synthesis technique, we discovered four soundness bugs in the manually written transformers of two real-world abstract-interpretation frameworks.

Contributions. We present a framework to create abstract transformers in a form in which the application of an abstract transformer involves the execution of relatively simple code. In particular, we advance the type (b) approach in two ways:

- Our framework treats the problem as a *program-synthesis problem*, which opens up a new possibility: rather than the representation of $\widehat{op}^\sharp$ being limited to a fixed interface of operations provided by an abstract domain, the user can supply a DSL—by specifying both its syntax and semantics—in which $\widehat{op}^\sharp$ is to be expressed.
- The assumptions of the framework are fairly minimal. The inputs are quite natural (§2). The synthesis algorithm that serves as the engine of the framework (§3 and §4) is formalized using a primitive Synthesize for synthesizing candidate L-transformers; a primitive MaxSatSynthesize, which is biased toward satisfying so-called "positive" examples—see §3; and two primitives that check soundness and precision of candidate L-transformers, generating counterexamples when the respective property fails to hold.
- We implemented a tool, called Amurth (§5), to support our framework, and obtained good results: we used Amurth to create sets of replacement abstract transformers for those used in two existing analyzers, and obtained essentially identical performance (§6). However, when we compared the existing transformers with the replacements synthesized by Amurth, we discovered that four of the existing transformers were *unsound*. These results demonstrate the risk of using manually created transformers, and hence the value of a tool for creating them automatically.

An extended version of this work is also available [Kalita et al. 2021], which contains more detailed experimental results.

2 PROBLEM STATEMENT

In this section, we define the problem addressed by our framework. Throughout the paper, we use a running example in which the goal is to synthesize a most-precise L-transformer for the absolute-value function `abs(x)` = |x| over the domain of intervals, where L is the DSL defined by

$$\begin{aligned} \mathit{Transformer} &::= \lambda \mathtt{a}.[E, E] \\ E &::= \mathtt{a.l} \mid \mathtt{a.r} \mid 0 \mid -E \mid +\infty \mid -\infty \mid E + E \mid E - E \mid E * E \mid \mathtt{min}(E, E) \mid \mathtt{max}(E, E) \end{aligned} \tag{2}$$

and the operations in E have their standard meaning. We describe what a user of the framework has to provide to solve this problem, and what they obtain as output.

The user of our framework needs to provide the following inputs:

Concrete domain: A definition of the concrete domain C—typically some set of values or program states. In our example, C is the set of integers.

Concrete transformer: A definition of the concrete semantics of the function f for which we are trying to synthesize a most-precise L-transformer. In our example, f is the function `abs` : `Int` → `Int`, which takes as input an integer and returns its absolute value. The semantics of `abs`(x) is provided by a logical specification $\Phi_{\mathsf{abs}}(x, x') =_{df} (x \geq 0 \wedge x' = x) \vee (x < 0 \wedge x' = -x)$, where x and x' represent the input and output, respectively.

Abstract domain: A definition of the abstract lattice $(A, \sqsubseteq, \bot)$, where A is the abstract domain, $\sqsubseteq$ is the partial order on elements of A, and $\bot$ is the least element of A. In our example, the domain of *intervals* $\mathcal{A}_{intv}$ abstracts a set of integers by maintaining only the maximum and minimum elements in the set. Each element a is a pair $[a.l, a.r]$ such that $a.l$ (which can be $-\infty$) denotes the minimum element and $a.r$ (which can be $+\infty$) denotes the maximum element.

Relation between the abstract and concrete domains: A definition of the concretization function $\gamma : A \to \mathcal{P}(C)$. In our example, the concretization function is $\gamma(a) = \{a.l, a.l + 1, \ldots, a.r\}$.

Language of Possible Transformers: The syntax and semantics of a DSL in which the synthesizer is to express abstract transformers. In our example, the DSL defined in Eqn. (2).

We assume that all semantic specifications are given in—or can be translated to—formulas in a fragment of first-order logic. For instance, the concretization function for intervals, $\gamma_{Interval}(\mathtt{a})$, can be specified via the predicate $x \in \gamma_{Interval}(\mathtt{a}) =_{df} \mathtt{a.l} \leq x \wedge x \leq \mathtt{a.r}$,

For concrete domain C and abstract domain A, Eqn. (1) specifies the behavior of the best abstract transformer for f, denoted by $\widehat{f^\sharp}$. As mentioned previously, Eqn. (1) does not provide the basis for an implementation of $\widehat{f^\sharp}$ because $\gamma(a)$ is potentially a large set. (It can even be an infinite set for some abstract domains.) Moreover, the introduction of language L into the problem introduces a new wrinkle: there is no guarantee that $\widehat{f^\sharp}$ is even expressible in language L.

Any transformer that is expressible in L is referred to as an *L-transformer*, denoted by $f_L^\sharp$. We use $\widehat{f_L^\sharp}$ to denote a *best* L-transformer for f. A best L-transformer must satisfy the dual objectives of soundness and precision.

Soundness: A *sound* L-transformer for a concrete function f must overapproximate the best transformer $\widehat{f^\sharp}$; i.e., $f_L^\sharp$ is sound iff for all $a \in A$, $\widehat{f^\sharp}(a) \sqsubseteq f_L^\sharp(a)$.

Precision: We define a (pre-)partial order on L-transformers with respect to precision ($\sqsubseteq_{pr}$) as follows: for all $f_1{}^\sharp, f_2{}^\sharp \in L$, $f_1{}^\sharp \sqsubseteq_{pr} f_2{}^\sharp \equiv \forall a \in A.\ \gamma(f_1^\sharp(a)) \subseteq \gamma(f_2^\sharp(a))$. A sound L-transformer $f^\sharp \in L$ is *most-precise* if it is minimal with respect to $\sqsubseteq_{pr}$.

Definition 2.1. An abstract transformer $f_L^\sharp \in L$ is ***a best L-transformer*** for a function f if $f_L^\sharp$ is both sound and most-precise (in which case, we denote it by $\widehat{f_L^\sharp}$). We use $\widehat{\mathcal{S}_L}(f)$ to denote the set of all best L-transformers for a function f.

Note that there may not exist a unique best L-transformer under $\sqsubseteq_{pr}$. For example, if f is the constant-zero function $\lambda x.0$, and language L can only express the functions $\{\lambda a.[0,k], \lambda a.[-k,0] \mid k \in \mathbb{N} \wedge k \geq 1\}$, the transformers $\lambda a.[0,1]$ and $\lambda a.[-1,0]$, which are incomparable under $\sqsubseteq_{pr}$, are both best L-transformers for f.

This paper targets the following problem:

> Given the concrete semantics Φ_f of a concrete transformer f, a description of an abstract domain $(A, \sqsubseteq, \sqcup)$, its relation to the concrete domain (γ), and a domain-specific language L, synthesize a best L-transformer for f.

As illustrated in §4, our method synthesizes the following transformer $\mathtt{abs}^\sharp : \mathcal{A}_{\mathtt{intv}} \rightarrow \mathcal{A}_{\mathtt{intv}}$:

$$\mathtt{abs}^\sharp(\mathtt{a}) = [\mathtt{max}(\mathtt{max}(0, \mathtt{a.l}), -\mathtt{a.r}),\ \mathtt{max}(-\mathtt{a.l}, \mathtt{a.r})]. \tag{3}$$

Even though the concrete function abs and the interval abstract domain are both quite simple, the transformer in Eqn. (3) is non-trivial, providing motivation for this work.

We now provide an informal argument that the transformer $\mathtt{abs}^\sharp$ is a best L-transformer $\widehat{abs_L^\sharp}$ over the interval domain. Given an input interval $\mathtt{a} \in \mathcal{A}_{intv}$, $\widehat{abs_L^\sharp}$ must behave as follows: (1) If $\gamma(\mathtt{a})$ only contains non-negative values (i.e., $\mathtt{a.l} \geq 0$), $\widehat{abs_L^\sharp}$ should return the input interval a itself. (2) If $\gamma(\mathtt{a})$ only contains non-positive values (i.e., $\mathtt{a.r} \leq 0$), $\widehat{abs_L^\sharp}$ should return the interval $[-\mathtt{a.r}, -\mathtt{a.l}]$. (3) If $\gamma(\mathtt{a})$ contains both positive and negative values (i.e., $\mathtt{a.l} < 0 \wedge \mathtt{a.r} > 0$), $\widehat{abs_L^\sharp}$ should return the interval $[0, \mathtt{max}(-\mathtt{a.l}, \mathtt{a.r})]$. A transformer meeting all these conditions is a best L-transformer (cf. Eqn. (1)). With a little bit of case analysis, one can see that the transformer $\mathtt{abs}^\sharp$ handles all of the cases as described above.

Note that, for a given abstract domain, all of the components provided as inputs to the framework are reusable. To synthesize a best L-transformer for a different concrete transformer g, one only needs to supply the specification of g.

For a given concrete transformer f, to synthesize a variety of best L_i-transformers over different abstract domains $(A_i, \sqsubseteq_i, \bot_i)$, one needs to supply the definitions of the different abstract domains, and—typically—define DSLs L_i with suitable operations to manipulate the various components in the representations of A_i values.

3 POSITIVE EXAMPLES, NEGATIVE EXAMPLES, SOUNDNESS, AND PRECISION

The engine that underlies our framework is an example-based synthesis algorithm to synthesize the target L-transformer. The key insight behind the algorithm is as follows:

> Use both positive and negative examples. Treat positive examples as hard constraints and negative examples as soft constraints.

The synthesis algorithm is formalized using a primitive Synthesize for synthesizing candidate L-transformers; a primitive MaxSatSynthesize, which is biased toward satisfying positive examples; and two primitives, CheckSoundness and CheckSoundness, which check the soundness and precision of candidate L-transformers, respectively—generating counterexamples when the respective property fails to hold. The algorithm proceeds in iterations, and maintains a set of examples $E = \langle E^+, E^- \rangle$, divided into positive (E^+) and negative (E^-) examples. It also issues *queries* to check whether the current candidate L-transformer is sound and precise. If it fails either the soundness or precision criterion, a new candidate L-transformer is created. In this section, we discuss the soundness and precision queries (§3.1 and §3.2, respectively). The algorithm itself is presented in §4, and the role of MaxSatSynthesize is explained in §4.2.

Definition 3.1 (Positive and Negative Examples). A *positive example* is a pair $\langle a, c' \rangle$ such that $a \in A$ and $c' \in \gamma(\widehat{f}^\sharp(a))$. A *negative example* is a pair $\langle a, c' \rangle$ such that $a \in A$, and there exists *some* best L-transformer $\widehat{f}_L^\sharp \in \widehat{\mathcal{S}}_L$ such that $c' \notin \gamma(\widehat{f}_L^\sharp(a))$.

Example 3.2. For the interval domain and the function abs, $\langle [5, 9], 6 \rangle$ is a positive example, but $\langle [5, 9], 12 \rangle$ is not. Along the same lines, assuming the DSL L from Eqn. (2), $\langle [5, 12], 2 \rangle$ is a negative example, while $\langle [5, 12], 7 \rangle$ is not a negative example.

Fig. 1 illustrates a case with two best L-transformers, $\widehat{\mathcal{S}}_L = \{\widehat{f}_{1L}^\sharp, \widehat{f}_{2L}^\sharp\}$, shown by the blue outlines; the black dashed outline shows the best abstract transformer $\widehat{f}^\sharp$. Points on the plot depict examples $\langle a, c' \rangle$; a point $\langle a, c' \rangle \in f^\sharp$ denotes that $c' \in \gamma(f^\sharp(a))$ and vice-versa. Point p_1 is a positive example because it is inside $\widehat{f}^\sharp$. Point n_1 (resp. n_2) is a negative example because it is outside best L-transformer $\widehat{f}_{1L}^\sharp$ (resp. $\widehat{f}_{2L}^\sharp$); and n_3 is a negative example because it is outside of both $\widehat{f}_{1L}^\sharp$ and $\widehat{f}_{2L}^\sharp$. Point x_0 is neither a positive example nor a negative example: it is outside $\widehat{f}^\sharp$, but inside both best L-transformers.

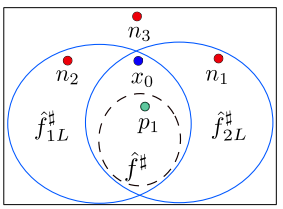
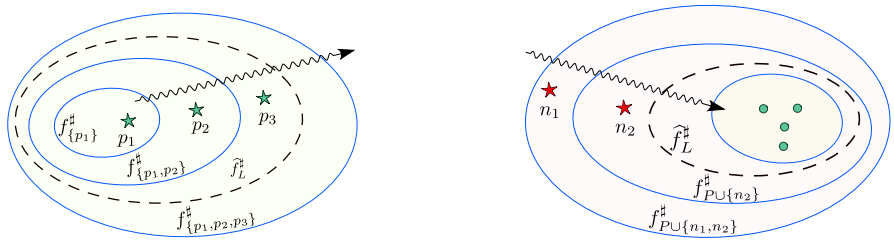


Fig. 1. Positive and negative examples, along with two best L-transformers and $\widehat{f}^\sharp$. The blue example x_0 is neither positive nor negative.

We assume that we have available a Synthesize procedure that accepts a set of examples $\langle E^+, E^- \rangle$, and returns *an* L-transformer $f_E^\sharp$ that includes all $e^+ \in E^+$ and excludes all $e^- \in E^-$.



(a) Adding positive counterexamples

(b) Adding negative counterexamples. P is a set of positive examples (•) .

Fig. 2. The blue ovals represent the successions of synthesized L-transformers; the black dashed oval represents a best L-transformer.

That is, $f^\sharp_E$ satisfies

$$sat^+(f^\sharp_E, E^+) \wedge sat^-(f^\sharp_E, E^-), \text{ where } sat^+(f^\sharp_E, E^+) =_{df} \forall \langle a, c\rangle \in E^+ \,.\, c \in \gamma(f^\sharp_E(a)), \\ \text{and } sat^-(f^\sharp_E, E^-) =_{df} \forall \langle a, c\rangle \in E^- \,.\, c \notin \gamma(f^\sharp_E(a)). \tag{4}$$

3.1 Soundness Queries (CheckSoundness)

Definition 3.3. A *soundness query* takes as input an L-transformer $f^\sharp_E$ (that is consistent with the set of examples E) and returns

(1) *True* if $f^\sharp_E$ is sound for all possible inputs; i.e., $\forall a.\ \widehat{f}^\sharp(a) \sqsubseteq f^\sharp_E(a)$,
(2) *False* and a pair of abstract and concrete values $\langle a, c'\rangle$, such that $c' \in \gamma(\widehat{f}^\sharp(a)) \setminus \gamma(f^\sharp_E(a))$. a is a witness to the unsoundness of $f^\sharp_E$; $\langle a, c'\rangle$ is called a ***positive counterexample***.

Because a logical specification Φ_f for the semantics of the concrete function f is provided, where Φ_f is expressed in a decidable logic, the soundness check can be carried out by checking the following formula for satisfiability:

$$\exists \langle a, c'\rangle, \text{ where } a \in A, \text{ and } c' \in C, \text{ such that } \exists c \in C, c \in \gamma(a) \wedge \Phi_f(c, c') \wedge c' \notin \gamma(f^\sharp_E(a)) \tag{5}$$

Let us now define the interface:

$$\textsc{CheckSoundness}(f^\sharp_E, f) = \begin{cases} \mathit{False}, \langle a, c'\rangle & \text{if Eqn. (5) is SAT} \\ \mathit{True}, \_ & \text{otherwise} \end{cases} \tag{6}$$

One might wonder if is is possible to solve the problem of synthesizing a best L-transformer using CheckSoundness alone. For example, one could use a counterexample-guided inductive synthesis (CEGIS) algorithm that uses CheckSoundness iteratively, to synthesize a succession of candidate L-transformers that cover larger and larger sets of examples. This (hypothetical) algorithm would maintain a set of positive examples E^+, use Synthesize to generate a sound L-transformer $f^\sharp_{E^+}$ for E^+, and issue a query to CheckSoundness to determine whether the current candidate L-transformer is sound in general. If not, the algorithm would add the positive counterexample to E^+ and repeat.

For example, suppose that on some iteration $\mathtt{abs}^\sharp_{E^+_0} = \lambda \mathtt{a}\text{:}\mathtt{[a.l,a.r]}$, and CheckSoundness is called to generate the positive counterexample $\langle [10, 15], 12\rangle$; a synthesizer, given $E^+_1 = E^+_0 \cup \{\langle [10, 15], 12\rangle\}$, may generate the L-transformer $\mathtt{abs}^\sharp_{E^+_1}(\mathtt{a}) = \mathtt{[0, a.r]}$, which is sound on all examples. The positive-counterexample-guided-synthesis algorithm is illustrated in Fig. 2a. The blue ovals represent the succession of synthesized L-transformers; the black dashed oval represents the best

transformer. The green stars represent positive counterexamples. If this algorithm terminates, it is guaranteed to generate an L-transformer that is sound on all of the possible inputs; however, the result is not guaranteed to be precise. For instance, our example may converge to the maximally imprecise result $\mathsf{abs}^\sharp_{E^+}(\mathsf{a}) = [-\infty, +\infty]$.

3.2 Precision Queries (CheckPrecision)

To help the reader's understanding, we start by discussing a slightly idealized version of the CheckPrecision query, denoted by CheckPrecision*. (Note the $*$ symbol.)

Definition 3.4. A *precision query* takes as input an L-transformer $f^\sharp_E$ (that is consistent with the set of examples E) and returns

(1) *False* if there exists an abstract example $\langle a, c'\rangle$, where $a \in A$ is an abstract input, and c' is a concrete value such that there exists a best L-transformer $\widehat{f}^\sharp_L \in \widehat{\mathcal{S}}_L$ for which $c' \in \gamma(f^\sharp_E(a)) \setminus \gamma(\widehat{f}^\sharp_L(a))$, and $\widehat{f}^\sharp_L$ is also *consistent with the set of examples* E. $\langle a, c'\rangle$ is a witness that $f^\sharp_E$ is not a best L-transformer; $\langle a, c'\rangle$ is called a ***negative counterexample***.
(2) *True* otherwise.

Let us start by considering the case when $\widehat{\mathcal{S}}_L = \{\widehat{f}^\sharp_L\}$ is a singleton set. Given a precision query, we could try to solve our problem using a CEGIS algorithm that uses the precision query iteratively, to synthesize successively more precise L-transformers. This (hypothetical) algorithm would maintain a set of examples E^- that we want our L-transformer to avoid, and would issue a precision query to determine whether the current candidate L-transformer is a most-precise L-transformer. If not, the algorithm would add the example returned by the query to E^- and repeat. For example, starting with $\mathsf{abs}^\sharp_\emptyset(\mathsf{a}) = [-\infty, +\infty]$, CheckPrecision* may generate a negative counterexample $\langle [1, 6], 10\rangle$ that improves the precision of the L-transformer, after which a new L-transformer is synthesized (and $E^- = \{\langle [1, 6], 10\rangle\}$ starts to build up): $\mathsf{abs}^\sharp_{E^-}(\mathsf{a}) = [\mathtt{0},\ \mathtt{a.l} + \mathtt{a.r}]$. Fig. 2b illustrates how this algorithm synthesizes a more-precise L-transformer on each iteration. (Green circles represent positive examples, and red stars represent negative counterexamples.)

The precision query only returns a negative counterexample when there exists an L-transformer that satisfies the current set of examples E. We can illustrate the definition by returning to the situation depicted in Fig. 1, where there are two best L-transformers, $\widehat{\mathcal{S}}_L = \{\widehat{f}^\sharp_{1L}, \widehat{f}^\sharp_{2L}\}$. Suppose that $E^+ = \{p_1\}, E^- = \{n1\}$, and that Synthesize has found best L-transformer $\widehat{f}^\sharp_{1L}$ (so we would like the algorithm to terminate). At this point, the only other best L-transformer that the precision query could find is $\widehat{f}^\sharp_{2L}$. It is true that $n_2 \in \gamma(\widehat{f}^\sharp_{1L}) \setminus \gamma(\widehat{f}^\sharp_{2L})$, but if n_2 were added to E^-, Synthesize would then be blocked from finding *any* L-transformer consistent with E, and thus a CEGIS procedure would fail (without returning any L-transformer). However, a best L-transformer returned by the precision query must satisfy both E^+ and E^-, and $\widehat{f}^\sharp_{2L}$ does not satisfy E^- (which contains n_1). The requirement to satisfy E prevents the precision query from returning $\widehat{f}^\sharp_{2L}$; instead, the precision query would return `True`, and CEGIS would terminate with $\widehat{f}^\sharp_{1L}$.

The following lemma describes how the CheckPrecision* query relates to the problem of synthesizing a best L-transformer.

Lemma 3.5. *Suppose that $f^\sharp$ is a sound L-transformer for f. $f^\sharp$ is a best L-transformer for f if and only if for every set of examples E with which $f^\sharp$ is consistent, the answer to the query CheckPrecision* with respect to $f^\sharp$ and examples E is True.*

The lemma shows how a *False* answer from CheckPrecision* guarantees that a more precise L-transformer exists in L, but a positive answer on a *single* set of examples E does not guarantee that the L-transformer $\widehat{f^\sharp}$ is a best L-transformer.

From CheckPrecision to CheckPrecision.* Given an L-transformer $f_E^\sharp$, the precision query CheckPrecision* returns true if $f_E^\sharp$ is a maximally precise overapproximation of $\widehat{f_L^\sharp}$ (with respect to $\sqsubseteq_{pr}$); otherwise it returns a counterexample $\langle a, c' \rangle$. Even though a logical specification Φ_f for the semantics of the concrete function f is provided, where Φ_f is expressed in a decidable logic, instantiating CheckPrecision* would be challenging because $\widehat{f_L^\sharp}$ is not known.

Instead, CheckPrecision—note $*$ symbol—uses the current set of examples E to approximate $\widehat{f_L^\sharp}$: a most-precise L-transformer that satisfies E (denoted by $\widehat{f_E^\sharp}$) is optimistically *assumed* to be $\widehat{f_L^\sharp}$. Of course, this approximation improves as more positive examples are discovered. Furthermore, we do not need to compute $\widehat{f_E^\sharp}$; any L-transformer $h_L^\sharp \sqsupseteq_{pr} \widehat{f_E^\sharp}$ (where $h_L^\sharp$ satisfies certain other conditions) suffices, as we explain next.

CheckPrecision attempts to discover a negative counterexample (e^-), while ensuring that there exists a sound L-transformer $h_L^\sharp$ that continues to satisfy both E^+ and E^- of example set E, in addition to satisfying the negative counterexample e^-. Clearly, $h_L^\sharp \sqsupseteq_{pr} \widehat{f_E^\sharp}$ for some $\widehat{f_E^\sharp}$. Given a candidate transformer $f_E^\sharp$, CheckPrecision asserts the following conditions:

- All examples in E^+ and E^- are satisfied (see Eqn. (4)).
- There exists a feasible L-transformer $h_L^\sharp$ that satisfies E, as well as a new negative counterexample $\langle a, c' \rangle \notin E^-$

$$sat^+(h_L^\sharp, E^+) \wedge sat^-(h_L^\sharp, E^-) \wedge sat^-(h_L^\sharp, \{\langle a, c' \rangle\}).$$

- The transformer $f_E^\sharp$ does not satisfy the new (negative) example $\langle a, c' \rangle$

$$sat^-(f_E^\sharp, \{\langle a, c' \rangle\}) = \mathit{false}.$$

The precision check can be expressed as follows:

$$\exists h_L^\sharp, \langle a, c' \rangle.\ sat^+(h_L^\sharp, E^+) \wedge sat^-(h_L^\sharp, E^- \cup \{\langle a, c' \rangle\}) \wedge \neg sat^-(f_E^\sharp, \{\langle a, c' \rangle\}) \tag{7}$$

We can now define the CheckPrecision interface:

$$\textsc{CheckPrecision}(f_E^\sharp, E^+, E^-) = \begin{cases} \mathit{False}, \langle a, c' \rangle & \text{if Eqn. (7) is SAT} \\ \mathit{True}, \_ & \text{otherwise} \end{cases} \tag{8}$$

4 AN ALGORITHM TO SYNTHESIZE A BEST L-TRANSFORMER

4.1 Accommodating Competing Objectives

The competing objectives of soundness and precision might seem to stand in the way of designing an algorithm that can achieve both. To address this issue, our algorithm essentially runs *two* CEGIS loops—one for soundness and one for precision. At each step, it non-deterministically chooses to query CheckSoundness or CheckPrecision, improving soundness or precision, respectively. When neither query generates any further counterexamples, the algorithm has provably synthesized a best L-transformer.

Because the two CEGIS loops operate independently, interacting only through examples, improving soundness can temporarily compromise precision and vice-versa. For example, the first positive counterexample could lead the synthesizer to emit $\lambda a.\top$, allowing progress only via a negative counterexample. Similarly, a negative counterexample can temporarily compromise soundness. (See §4 and §5.)

Example 4.1. We now illustrate the principles used in the algorithm on the abs function, using the interval abstract domain and the DSL from Eqn. (2). To start, $\mathsf{abs}^\sharp_{\langle\emptyset,\emptyset\rangle}(\mathsf{a}) = \bot$. Now suppose that CheckSoundness is used to generate a positive counterexample, say, $\langle[5,5],5\rangle$. The algorithm records this example, and attempts to synthesize an L-transformer that satisfies it, say,

$$\mathsf{abs}^\sharp_{\langle\{\langle[5,5],5\rangle\},\emptyset\rangle}(\mathsf{a}) = [\mathsf{a.l}, \infty].$$

Because the above L-transformer is still not sound, there could be additional calls to CheckSoundness to generate new counterexamples to soundness. However, the algorithm can choose nondeterministically to perform a precision step, calling CheckPrecision to generate a negative counterexample. Suppose that CheckPrecision generates $\langle[1,6],10\rangle$, after which the following L-transformer is synthesized:

$$\mathsf{abs}^\sharp_{\langle\{\langle[5,5],5\rangle\},\{\langle[1,6],10\rangle\}\rangle}(\mathsf{a}) = [\mathtt{0},\ \mathsf{a.l} + \mathsf{a.r}].$$

Eventually, the algorithms will not be able find any additional positive or negative examples, and will return the following sound and precise L-transformer:

$$\mathsf{abs}^\sharp_E(\mathsf{a}) = [\mathtt{max}(\mathtt{max}(\mathtt{0}, \mathsf{a.l}), -\mathsf{a.r}),\ \mathtt{max}(-\mathsf{a.l}, \mathsf{a.r})].$$

4.2 Consistency of Positive and Negative Examples

We define the following interface functions:

$$\textsc{CheckConsistency}(E^+, E^-) = \exists f^\sharp_E.\ sat^+(f^\sharp_E, E^+) \wedge sat^-(f^\sharp_E, E^-)$$

$$\textsc{Synthesize}(E^+, E^-) = \begin{cases} f^\sharp_E & \text{if } \exists f^\sharp_E.\ sat^+(f^\sharp_E, E^+) \wedge sat^-(f^\sharp_E, E^-) \\ \bot & \text{otherwise} \end{cases} \tag{9}$$

In our algorithm, Synthesize is only called if CheckConsistency returns true and is, therefore, guaranteed to return a transformer.

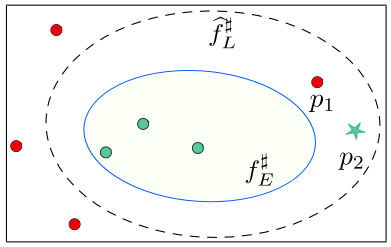
Recall that whenever a new negative counterexample e^- is generated for a candidate transformer $f^\sharp_E$, CheckPrecision uses $h^\sharp_L$ as an over-approximation ($\sqsupseteq_{pr}$) of (some) $\widehat{f^\sharp_E}$. Because e^- is excluded from $h^\sharp_L$, e^- is also excluded from some $\widehat{f^\sharp_L}$. However, there are two possible cases:

(1) The negative counterexample $\langle a, c'\rangle$ is such that $c' \notin \gamma(\widehat{f^\sharp_L}(a))$ (and $c' \in \gamma(f^\sharp_E(a))$). This case is illustrated by the red star labeled n_1 in Fig. 3, where $\widehat{f^\sharp_L}$ and $f^\sharp_E$ are shown as the dashed black and solid blue ovals, respectively. In this case, n_1 is inside $f^\sharp_E$, but outside $\widehat{f^\sharp_L}$.

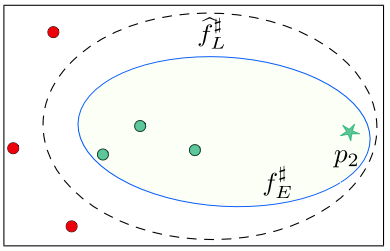
(2) The negative counterexample $\langle a, c'\rangle$ is such that $c' \in \gamma(\widehat{f^\sharp_L}(a))$ (and $c' \in \gamma(f^\sharp_E(a))$). Hence, by excluding this negative counterexample, the synthesized transformer $f^\sharp_E$ remains sound w.r.t. the examples E, but can become unsound w.r.t. $\widehat{f^\sharp_L}$. This case is illustrated by the red star labeled n_2 in Fig. 3: n_2 is inside both $f^\sharp_E$ and $\widehat{f^\sharp_L}$.

If $E^+ = \{\langle[1,5],2\rangle\}$ and $f^\sharp_E$ is $\lambda a.[a.l, +\infty]$, the negative example $\langle[-15,-11],2\rangle$ illustrates an example of case 2 above, and the transformer $\lambda a.[a.l, a.r]$ would be a possible $h^\sharp_L$.

Because $\widehat{f^\sharp_L}$ is unknown, our algorithm has no means for identifying which of the above cases a negative counterexample falls into. Hence, the algorithm tentatively marks the example as a negative example. However, such an assumption has the risk of making the set of positive and negatives examples inconsistent: the examples $E = \langle E^+, E^-\rangle$ are inconsistent if there does not exist any L-transformer $g^\sharp_E$ such that $sat^+(g^\sharp_E, E^+) \wedge sat^-(g^\sharp_E, E^-)$.



(a) Positive example p_2 creates an inconsistency w.r.t. p_1.



(b) The negative example p_1 is dropped.

Fig. 4. (a) Inconsistent positive and negative examples. (b) Illustration of how our algorithm resolves the inconsistency. (★: most recent positive example)

In our example, suppose that in the next iteration the algorithm selects the soundness check, calling CheckSoundness to generate another positive example, say $\langle[-1, 0], 1\rangle$ (p_2 in Fig. 4a). Now the examples are inconsistent because there does not exist any L-transformer that includes $\langle[-1, 0], 1\rangle$ and excludes $\langle[-15, -11], 2\rangle$. At this point, CheckConsistency finds that the positive and negative examples are inconsistent, and hence, some negative example added by CheckPrecision must have been a positive example.

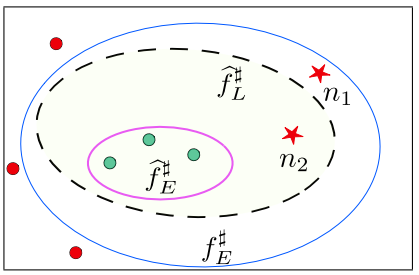


Fig. 3. Negative examples: n_1 is outside of $\widehat{f}_L^\sharp$ and n_2 is inside $\widehat{f}_L^\sharp$.

The algorithm uses Occam's razor to solve this dilemma: the goal of the synthesis step becomes "synthesize a transformer that ignores the *smallest number of negative examples*." In our case, by ignoring the negative example $\langle[-15, -11], 2\rangle$, it is possible to synthesize the abstract transformer,

$$f_E^\sharp(\mathsf{a} : \mathcal{A}_{\mathtt{intv}}) : \mathcal{A}_{\mathtt{intv}} = [\mathtt{max}(\mathtt{max}(0, \mathsf{a.l}), -\mathsf{a.r}),\ \mathtt{max}(-\mathsf{a.l}, \mathsf{a.r})]$$

To reestablish the consistency of the sets of positive and negative examples, the algorithm now drops the negative examples that it could not satisfy from E^-. This scenario is illustrated in Fig. 4b, where the negative example p_1 is dropped, so that positive and negative examples are now consistent. Hence, in addition to a procedure, Synthesize, the algorithm needs access to a stronger synthesis procedure, MaxSatSynthesize. This procedure is similar to the standard partial MaxSat procedure [Li and Manya 2009] (allowing for hard and soft constraints), but applied in the context of program synthesis. When the positive and negative example sets are conflicting, MaxSatSynthesize attempts to synthesize a transformer by dropping the smallest number of negative examples that make the query satisfiable.

Definition 4.2. Given example set $\langle E^+, E^-\rangle$ for which there is no L-transformer $g_E^\sharp$ such that CheckConsistency$(g_E^\sharp, E^+, E^-) = \mathit{true}$, MaxSatSynthesize returns an L-transformer that can be synthesized by dropping the smallest set of negative examples:

$$\textsc{MaxSatSynthesize}(E^+, E^-) = \begin{cases} f_E^\sharp, D & \text{if } \exists f_E^\sharp, D.\ sat^+(f_E^\sharp, E^+) \wedge sat^-(f_E^\sharp, E^- \setminus D), \\ & \textit{where } D \textit{ is minimal}, \\ \bot & \text{otherwise} \end{cases} \tag{10}$$

Algorithm 1: SynthesizeTransformer(f) // $C, (A, \sqsubseteq, \bot), \gamma, L$

```
1  f_E^♯ ← λa : ⊥
2  isSound, isPrecise ← False
3  E⁺, E⁻ ← InitializeExamples()
4  while ¬isSound ∨ ¬isPrecise do
5      if ¬isSound ∧ ¬isPrecise then
6          if CheckConsistency(E⁺, E⁻) then
7              f_E^♯ ← Synthesize(E⁺, E⁻)
8          else
9              f_E^♯, δ ← MaxSatSynthesize(E⁺, E⁻)
10             if f_E^♯ ≠ ⊥ then
11                 E⁻ ← E⁻ \ δ
12             else
13                 return ⊥
14     if ∗ then
15         isSound, e ← CheckSoundness(f_E^♯, f)
16         if ¬isSound then
17             isPrecise ← False
18             E⁺ ← E⁺ ∪ {e}
19     else
20         isPrecise, e ← CheckPrecision(f_E^♯, E⁺, E⁻)
21         if ¬isPrecise then
22             isSound ← False
23             E⁻ ← E⁻ ∪ {e}
24 return f_E^♯
```

4.3 Putting It All Together

The algorithm to synthesize a best L-transformer is given as Alg. 1. In essence, it runs two CEGIS loops, attempting to meet the dual goals of soundness and precision. The two CEGIS loops interact with each other via the set of positive and negative counterexamples that they generate, and the algorithm can terminate only when both loops have attained their objectives. The algorithm starts off with a trivial transformer that always returns the abstract bottom, and (potentially empty) example sets E^+ and E^- (line 3). While the synthesized transformer is either unsound or imprecise (line 4), the algorithm synthesizes a transformer (line 7) that is consistent with the current set of positive (E^+) and negative (E^-) examples. Then, it non-deterministically chooses to perform either a soundness check (line 14) or a precision check (line 19), expanding its set of examples (positive or negative, respectively) in each case if the check fails.

As discussed in §4.2, it is possible that the positive and negative examples from CheckSoundness and CheckPrecision become inconsistent. If the examples are inconsistent (i.e., CheckConsistency fails in line 6), the algorithm invokes MaxSatSynthesize to synthesize a transformer that satisfies all positive examples and a maximal set of negative examples, returning the unsatisfied negative examples in δ. The examples are adjusted to reinstate consistency (dropping the unsatisfied negative examples), and the loop continues. If MaxSatSynthesize fails to synthesize a transformer,

then it must be the case that language L is inadequate to express a valid abstraction of f, and the algorithm terminates with $\bot$.

The invariant of Alg. 1 is that, on each iteration, at line 14 a transformer $f_E^\sharp$ has been computed that is (i) expressible in L, and (ii) consistent with respect to $\langle E^+, E^- \rangle$. Then, in lines 14–23, soundness or precision is checked. When both hold, $f_E^\sharp$ is a suitable L-transformer, and Alg. 1 terminates.

Alg. 1 is stated as a non-deterministic procedure, as a way to separate mechanism from policy. However, any fair scheduler can be used to resolve the non-determinism—e.g., alternating between CheckSoundness and CheckPrecision in successive iterations. (We describe the scheduler that we use in our implementation in §5.)

Theorem 4.3. *If Alg. 1 terminates with a valid $f_E^\sharp$ (i.e., does not terminate with $\bot$), then $f_E^\sharp$ is a best L-transformer for the concrete function f.*

Proof. For the algorithm to return a valid L-transformer, both CheckSoundness (Eqn. (6)) and CheckPrecision (Eqn. (8)) must have returned true (lines 4, 15, and 20).

Passing the test of CheckSoundness guarantees that the L-transformer is sound. CheckPrecision attempts to find a more precise L-transformer than $f_E^\sharp$ that still satisfies the current set of positive examples; the inability of CheckPrecision to find such an L-transformer proves that $f_E^\sharp$ must also be a best L-transformer.

□

Theorem 4.4. *Let L be the DSL whose syntax and semantics has been supplied by the user. If L is a finite language (i.e., the grammar for L generates only a finite number of terms), and the non-deterministic choice at line 14 is resolved by a fair scheduler, then Alg. 1 always terminates.*

Proof. We assume that the executions of Alg. 1 are fair: i.e., the non-deterministic scheduler chooses each of lines 15 and 20 infinitely often. Consider a "normalized" execution trace, in which all calls to CheckSoundness on a sound L-transformer and CheckPrecision on a precise L-transformer are filtered out. That is, each remaining call on CheckSoundness adds to E^+, and each remaining call on CheckPrecision adds to E^-.

Consider the normalized sub-trace t between any two consecutive calls to CheckSoundness. Note that by the assumption of fairness, only a finite number of calls on CheckSoundness and CheckPrecision could have been filtered out of t. Now consider the number of calls to CheckPrecision and MaxSatSynthesize in t.

- There can be at most one call on MaxSatSynthesize. The reason is that a call on MaxSatSynthesize establishes consistency among E^+ and E^-, and subsequent calls to CheckPrecision can only generate negative counterexamples that are consistent with the current set E^+. (That is, in the suffix of t after the call on MaxSatSynthesize, E^+ and E^- always pass CheckConsistency.)
- In effect, each call to CheckPrecision removes at least one L-transformer from consideration (by generating a new negative counterexample that is added to E^-). Thus, when DSL L is finite, there can only be a finite number of calls on CheckPrecision in t.

Now consider the full normalized trace. Each call on CheckSoundness adds a new positive example to E^+, and thus removes at least one L-transformer from consideration during a subsequent call on Synthesize. Consequently, there can only be a finite number of calls on CheckSoundness in the normalized trace, and hence every execution of Alg. 1 with a finite DSL L must terminate. □

Corollary 4.5. *Assuming the same premises as Thm. 4.4, if Alg. 1 terminates with a valid $f_E^\sharp$ (i.e., does not terminate with $\bot$), then $f_E^\sharp$ is a best L-transformer for the concrete function f.*

```
1  generator interval genT(interval inv) {
2     int t = ??;
3     if(t == 0) return inv;
4     if(t == 1) return [0, 0];
5     ...
6     interval x = genT(inv);
7     interval y = genT(inv);
8     ...
9     if(t == 2) return addInterval(x, y);
10    if(t == 3) return subtractInterval(x, y);
11    ...
12 }
```

(a) A snippet of a Sketch generator for the DSL shown in Eqn. (2). In essence, lines 3, 4, 9, and 10] are "productions" in a grammar with a supplied semantics that gives an interpretation over interval arithmetic.

```
1 interval fun♯(interval inv){
2    if(isBot(inv)){
3       return ⊥;
4    }
5    else{
6       return genT(inv);
7    }
8 }
```

(b) Representative function for the interval-domain transformer, which returns $\bot$ if the input is also $\bot$; otherwise it returns the output of generator genT.

Fig. 5. Template for the abstract transformers using a Sketch generator.

5 IMPLEMENTATION

We implemented our framework in a tool, called Amurth. Amurth is written in Python and uses the Sketch synthesizer (v. 1.7.5) [Solar-Lezama 2013] as a subroutine. Amurth's inputs are

(1) The type of the elements in the concrete domain, e.g., integers.
(2) A logical specification Φ_f of the semantics of concrete function f.
(3) An implementation, written in Sketch, that specifies the partial order ($\sqsubseteq$) in the abstract domain, along with an implementation of `gammaCheck`(c, a), which, given $c \in C$ and $a \in A$, checks whether $c \in \gamma(a)$.
(4) The definition of the DSL, written in the Sketch language.
(5) Optionally, a set of initial positive and negative examples.

Sketch is built on top of the C programming language, and allows one to write programs with holes, assertions, and a minimization objective over an integer expression. The goal of the Sketch solver is to find integer values for the holes that cause all assertions to hold, while minimizing the value of the given objective. As discussed below, Amurth uses these features to implement the primitives used in Alg. 1.

One specifies a DSL L in Sketch using generators. Generators are special functions—possibly recursive—that contain holes, and allow one to build complex programs via recursion. Sketch allows one to set a bound on how deep the recursion can be—thereby bounding the size of a generated program. Generators contain holes that allow the Sketch solver to pick productions to build a program that satisfies a given specification. Fig. 5a shows how the DSL from Eqn. (2) can be specified using a generator. In such an encoding, a generator does two things simultaneously: (1) it uses the hole value to select some production, and (2) it "executes" the code corresponding to the production. Fig. 5a is an example of a Sketch idiom: it sidesteps generating an intermediate object (e.g., an abstract syntax tree) that then needs to be executed by an interpreter to obtain the semantics. Fig. 5b shows the template for the abstract transformers for the interval domain. It takes as input an interval, and returns $\bot$ in case the input is $\bot$; otherwise, it returns the output of the call on the generator.

A generator for an interval-to-interval transformer has a signature of the following form:

```
generator interval genT(interval in){...}
```

Sketch allows imposing constraints on the programs produced by a generator. For instance, one can provide a set of input/output pairs, and assert that the program exhibits those behaviors, e.g.,

$$\texttt{assert fun}^\sharp([1,2]) == [2,3] \;\&\&\; \ldots \;\&\&\; \texttt{fun}^\sharp([1,3]) == [2,4]$$

The solver will then compute a realization of `genT` (i.e., an L-transformer) that correctly matches the given examples.

Because our algorithm may require synthesizing transformers that are correct on "most" of a set of examples, we can use Sketch to count how many examples are satisfied, and then maximize that value. Although there is no maximize primitive, it can be simulated using the `minimize` construct:

```
c = NumExamples;
if(fun#([1, 2]) == [2, 3]) c = c − 1; ...
minimize(c);
```

In §6, we show some of the grammars for the DSLs used in our experiments, e.g., Eqns. (11), (12), (13), and (16), each of which is encoded using the idiom described above. Because the grammars are recursive, their languages are of infinite cardinality; however, Sketch imposes a (user-controllable) unrolling bound on generators that are recursive. Cor. 4.5 holds for all of our experiments with Amurth—i.e., for a concrete function f, if Alg. 1 terminates with something other than $\bot$, then the function $f^\sharp_E$ obtained is a best L-transformer for f—however, L is the finite language of programs that fall within the unrolling bound, not the full infinite-cardinality language of the grammar. The current implementation indicates the existence (yes/no) of a bug in a manually written transformer. To locate a bug, a user must currently inspect the faulty transformer and the one synthesized by Amurth manually.

CheckSoundness. Given a logical specification Φ_f of concrete function f, we use Sketch as a satisfiability solver. For a given candidate L-transformer $f^\sharp$, the `gammaCheck` primitive is used to specify that Sketch should try to find an input $c \in \gamma(a)$, such that $\Phi_f(c) \notin \gamma(f^\sharp(a))$. If successful, $\langle a, \Phi_f(c)\rangle$ is returned as a positive counterexample.

CheckPrecision, CheckConsistency, Synthesize, and MaxSatSynthesize. Although in Alg. 1 CheckPrecision, CheckConsistency, and Synthesize are shown as separate procedure calls, Amurth implements all three calls via a single invocation of Sketch (which finds a new L-transformer that is more precise than the given one, as well as a witness example). MaxSatSynthesize uses the `minimize` construct from Sketch, as described above.

5.1 Designing a DSL

As in all synthesis tasks, the design of the DSL is important. DSLs for expressing abstract transformers essentially involve a combination of

(1) Primitives that operate on concrete values (e.g., addition, subtraction, etc. for numeric values; isSubset, size, containsSpace, etc. for strings). The reason is that concrete values may appear as components of abstract values, such as endpoints of an interval or a member of a string set.
(2) Operations to deconstruct an abstract value (e.g., to access one of the two endpoints of an interval).
(3) Operations to construct an abstract value (e.g., to pair the two endpoints of an interval).
(4) Boolean connectives.
(5) Control-flow constructs.

Consider the DSL shown in Eqn. (2). The production *Transformer* ::= λa.$[E, E]$ is an example of item (3): an interval is created from the values of two expressions. Nonterminal E derives expressions for use in a transformer. The productions E ::= `a.l` | `a.r` are examples of item (2):

here a refers to the λ-bound variable in λa.[E, E], and a.l (a.r) selects the left (right) end of interval a. The remaining productions of nonterminal E are examples of item (1): they allow synthesizing arithmetic operations, e.g., sums, differences, products, etc. This DSL is a natural fit for the problem of creating an interval transformer for the absolute-value function.

The key point in our work is that we have a DSL L, and (provably) obtain one of the best results possible in L. For a given DSL grammar G, one has different languages L_1, L_2, etc. according to the grammar depth that one chooses to use. Thus, if you increase the grammar depth from k to $k+1$, you would then obtain a best L_{k+1}-transformer, which can be more precise than a best L_k-transformer. In the experiments described in §6, which involved synthesizing 57 transformers (for 15 operations and 8 abstract domains: $57 = 6 \times 5 + 9 \times 3$; see Tab. 1), we found that most transformers could be synthesized with reasonably simple grammars and small grammar depths. In our experiments, we used grammar depth 3 for almost all of the transformer-synthesis tasks. We chose 3 because we found that we were able to synthesize almost all transformers in a reasonable time (<600 seconds), and the transformers obtained were equal to or better than (i.e., were sound) the transformers used in SAFE_{str} and the tool of Navas et al..

In §6.3, we provide additional discussion about the design of DSLs, using example DSLs from §6.1 and §6.2. §6.3 also discusses how changing the DSL can lead to a different (more-precise or less-precise) transformer and/or impact the synthesis time.

5.2 Refinements to Alg. 1

In §4.3, we stated Alg. 1 as a non-deterministic procedure, as a way to separate mechanism from policy. However, any fair scheduler can be used to resolve the non-determinism (e.g., alternating between CheckSoundness and CheckPrecision in successive iterations). We found that a simple deterministic scheduler that prioritizes CheckPrecision over CheckSoundness works well in practice. In Alg. 1, if both the *isSound* and *isPrecise* flags are *False*, our algorithm chooses the branch that calls CheckPrecision. Fairness is ensured by forcing a call on CheckSoundness after k consecutive calls on CheckPrecision have been performed. Our implementation uses this strategy (with $k = 50$).

6 EVALUATION

We performed two studies with Amurth.

Case Study 1 (§6.1): We used Amurth to synthesize abstract transformers for string operations using the multiple string abstract domains employed in SAFE_{str} [Amadini et al. 2017]. We compared the synthesized transformers with the hand-crafted transformers implemented in SAFE_{str}.

Case Study 2 (§6.2): We used Amurth to synthesize abstract transformers for simple mathematical operations using three kinds of interval abstract domains defined by Navas et al. [2012]. We compared the synthesized transformers with those implemented by Navas et al.

Table 1. List of abstract operations synthesized by Amurth for the String and Fixed-Bitwidth Interval domains.

Domain Type	Abstract Domains	Operations
String	Constant String ($\mathcal{CS}$) String Set (size k) ($\mathcal{SS}_k$) Char Inclusion ($\mathcal{CI}$) Prefix-Suffix ($\mathcal{PS}$) String Hash ($\mathcal{SH}$)	charAt$^\sharp$, concat$^\sharp$, contains$^\sharp$, toLower$^\sharp$, toUpper$^\sharp$, trim$^\sharp$
Fixed Bitwidth Interval	Unsigned-Int ($\mathcal{A}_{\texttt{uintv}}$) Signed-Int ($\mathcal{A}_{\texttt{uintv}}$) Wrapped ($\mathcal{W}$)	add$^\sharp$, sub$^\sharp$, mul$^\sharp$, and$^\sharp$, or$^\sharp$, xor$^\sharp$, shl$^\sharp$, ashr$^\sharp$, lshr$^\sharp$

See Tab. 1 for the abstract domains and operations used in these studies. The two studies were designed to shed light on the following research questions:

[RQ1]: How long does it take Amurth to synthesize best L-transformers?
[RQ2]: How do the abstract transformers synthesized by Amurth compare to manually written ones?

We ran all experiments on an Intel(R) Xeon(R) 2.00GHz E5-2620 CPU with 32GB RAM, running Ubuntu 16.04. Each reported time is the median of three runs of Amurth. We used an unrolling depth of 3 for the DSLs used in our experiments. We used a timeout value of 600s per call on Sketch.

6.1 Case Study 1: Transformers for the String Domains in SAFE$_{\text{str}}$

SAFE$_{\text{str}}$ is a state-of-art static analyzer for programs involving complex string operations. In this study, we used Amurth to create abstract transformers over the abstract domains used in SAFE$_{\text{str}}$.

6.1.1 String Abstract Domains. Our study considered the five string abstract domains summarized below, which are all used in SAFE$_{\text{str}}$.

String Set ($\mathcal{SS}_k$) [Madsen and Andreasen 2014]. This domain can abstract a finite set of strings precisely, as long as the size of the set does not exceed k. An element of this domain (with the exception of $\top_{\mathcal{SS}_k}$) is a set of constant strings of size up to k—i.e., $\mathcal{SS}_k = \{\top_{\mathcal{SS}_k}\} \cup \{S \mid S \subseteq \Sigma^* \wedge |S| \leq k\}$, where Σ is the set of all characters, and $\bot_{\mathcal{SS}_k}$ is the empty set. Let S be a set of strings, then the abstraction function is defined as $\alpha_{\mathcal{SS}_k}(S) = S$ if $|S| \leq k$ and $\top_{\mathcal{SS}_k}$ otherwise. The lattice operations, $\sqcup_{\mathcal{SS}_k}$ (join) and $\sqsubseteq_{\mathcal{SS}_k}$ (partial order) are defined in terms of set $\cup$ and $\subseteq$, respectively. If a set exceeds size k, its abstraction is $\top_{\mathcal{SS}_k}$.

Constant String ($\mathcal{CS}$) [Madsen and Andreasen 2014]. This domain can abstract precisely up to one concrete string; it is a special case of the previous domain—i.e., $\mathcal{CS} = \mathcal{SS}_1$.

Character Inclusion ($\mathcal{CI}$) [Amadini et al. 2017]. An element of this domain is a pair of two sets of characters, $[L, U]$. The set L (resp. U) denotes what characters a string must (resp. may) contain to be in the concretization of this abstract element. We will sometimes refer to L and U as *must* and *may* sets, respectively, in our discussion. Formally, $\mathcal{CI} = \{\bot_{\mathcal{CI}}\} \cup \{[L, U] \mid L, U \subseteq \Sigma, L \subseteq U\}$. Given a string $s \in \Sigma^*$, let $char(s)$ denote the set of characters in s. The abstraction function is defined as $\alpha_{\mathcal{CI}}(\{s_1, \ldots, s_n\}) = [\bigcap_i char(s_i), \bigcup_i char(s_i)]$, and the concretization function is then defined as $\gamma_{\mathcal{CI}}([L, R]) = \{s \mid L \subseteq char(s) \subseteq R\}$. The partial-order relation is defined as $[L_1, U_1] \sqsubseteq_{\mathcal{CI}} [L_2, U_2] \Leftrightarrow (L_1 \subseteq L_2 \wedge U_1 \subseteq U_2)$, and the $\sqcup_{\mathcal{CI}}$ (join) operation is defined as $[L_1, U_1] \sqcup_{\mathcal{CI}} [L_2, U_2] = [L_1 \cap L_2, U_1 \cup U_2]$.

Prefix-Suffix ($\mathcal{PS}$) [Amadini et al. 2017]. An element of this domain is a pair consisting of two strings $\langle pre, suf \rangle$ (we use a different pair notation to distinguish from the previous domain), where $pre \in \Sigma^*$ is the longest common prefix (*lcp*) and $suf \in \Sigma^*$ is the longest common suffix (*lcs*) for the corresponding set of strings. The abstraction function is defined as $\alpha_{\mathcal{PS}}(S) = \langle lcp(S), lcs(S) \rangle$, and the concretization function is defined as $\gamma_{\mathcal{PS}}(\langle pre, suf \rangle) = \{s \mid \exists s_1 \in \Sigma^*, \exists s_2 \in \Sigma^*.\ s{=}pre.s_1 \wedge s{=}s_2.suf\}$. The partial order is defined as $\langle pre_1, suf_1 \rangle \sqsubseteq_{\mathcal{PS}} \langle pre_2, suf_2 \rangle \Leftrightarrow lcp(\{pre_1, pre_2\}) = pre_2 \wedge lcs(\{suf_1, suf_2\}) = suf_2$, and the join operation is defined as $\langle pre_1, suf_1 \rangle \sqcup_{\mathcal{PS}} \langle pre_2, suf_2 \rangle = \langle lcp(\{pre_1, pre_2\}), lcs(\{suf_1, suf_2\}) \rangle$.

String Hash ($\mathcal{SH}$) [Madsen and Andreasen 2014]. This domain uses a hash function $h : \Sigma^* \to U$, which takes the sum of the character codes in a string, and maps it to an element in a fixed-size universe $U = \{0, \ldots, b-1\}$. The concrete implementation of SAFE$_{\text{str}}$ uses the function $h(s) = (\Sigma_{c \in char(s)} I(c))\ mod\ b$, where $I : \Sigma \to \mathbb{N}$, is a mapping from characters to an integer value. An element of the abstract domain $\mathcal{SH}$ is a set $H \subseteq U$ denoting the hash values of the strings being tracked. Let S be a set of strings, then the abstraction function is defined as $\alpha_{\mathcal{SH}}(S) = \{h(s) \mid s \in S\}$.

The concretization function is defined as $\gamma_{\mathcal{SH}}(H) = \{s \in \Sigma^* \mid h(s) \in H\}$. The partial order ($\sqsubseteq_{\mathcal{SH}}$) and join ($\sqcup_{\mathcal{SH}}$) are defined as $\subseteq$ and $\cup$ on sets, respectively.

6.1.2 Abstract Transformers for String Operations. Our study involved six string-manipulation operations: `concat`, `contains`, `charAt`, `toLower`, `toUpper`, and `trim`. For each domain from §6.1.1, we used Amurth to synthesize abstract transformers for the six functions. For each concrete function and abstract domain, we specified a particular DSL L, and then ran Amurth to synthesize a best L-transformer. The time taken by Amurth to synthesize an L-transformer across all experiments varies between 3.73s and 1,983.83s; see Tab. 2. A table showing detailed results for the string-domain experiments is available in the extended version [Kalita et al. 2021].

As it happens, the $\mathcal{SH}$ domain only supports a non-trivial abstract transformer for the concrete function `concat`; for each of the other five functions, the best abstract transformer is the trivial abstract transformer $\lambda a.(\text{if } a = \bot \text{ then } \bot \text{ else } \top)$ [Amadini et al. 2017; Madsen and Andreasen 2014]. Amurth synthesized these transformers, too; each took a nontrivial amount of time because Amurth had to establish that the DSL could not express a more precise abstract transformer.

Table 2. Time, in seconds, to synthesize abstract transformers for string functions. The shaded cell indicates that we used a sketch of the transformer in this case, and Amurth only synthesizes the *holes* in the sketch.

f	$\mathcal{CS}$	$\mathcal{SS}_k$	$\mathcal{CI}$	$\mathcal{PS}$	$\mathcal{SH}$
`charAt`	18.29	3.94	24.91	5.94	3.76
`concat`	99.05	9.57	1,983.83	8.92	609.30
`contains`	132.06	78.42	1,804.69	9.13	10.39
`toLower`	11.26	11.74	381.65	6.91	8.44
`toUpper`	9.77	12.18	735.13	5.85	3.73
`trim`	4.31	16.35	641.53	8.52	8.29

In this section, we focus the discussion on the transformers that produced interesting behaviors and challenges, and discuss the DSLs that we used.

Transformer for `contains` in the $\mathcal{CI}$ Domain. The concrete function `contains(arg1, arg2)` returns `True` if `arg2` is a contiguous substring of `arg1`, and `False` otherwise. Amurth takes 1,804.69s to synthesize the abstract transformer $\texttt{contains}^\sharp_{\texttt{CI}}$ shown in Fig. 6a. The transformer takes two abstract inputs in $\mathcal{CI}$, and returns an abstract Boolean value in `AbsBool`. `AbsBool` contains four elements, `BoolBot`, `BoolTrue`, `BoolFalse`, and `BoolTop`, and these elements satisfy the partial order `BoolBot` $\sqsubseteq$ *value* $\sqsubseteq$ `BoolTop`, where *value* can be either `BoolTrue` or `BoolFalse`. `BoolTrue` and `BoolFalse` are incomparable. The DSL we used to synthesize this transformer is

$$
\begin{aligned}
\textit{Transformer} &::= \lambda \mathtt{a}_1, \mathtt{a}_2.AB \\
AB &::= \mathtt{ite}(B, AB, AB) \mid \mathtt{BoolTop} \mid \mathtt{BoolBot} \mid \mathtt{BoolTrue} \mid \mathtt{BoolFalse} \\
B &::= \mathtt{isSubset}(LU, LU) \mid \mathtt{size}(LU) \leq 1 \mid \mathtt{isBot}(CI) \mid \mathtt{isTop}(CI) \mid \mathtt{isEmpty}(CI) \\
&\quad \mid \neg B \mid B \wedge B \mid B \vee B \\
CI &::= \mathtt{a}_1 \mid \mathtt{a}_2 \\
LU &::= CI.\mathtt{l} \mid CI.\mathtt{u}
\end{aligned}
\tag{11}
$$

A program in this DSL computes an `AbsBool`, hence the initial nonterminal AB. Other nonterminals denote the types Boolean (B), $\mathcal{CI}$ (CI), and L, U values (LU). This DSL contains a number of auxiliary functions, e.g., `isBot`, `isTop`, `isSubset`, which can be used to inspect abstract values. The auxiliary function `isBot` (resp. `isTop`) checks whether an abstract value in $\mathcal{CI}$ is $\bot_{\mathcal{CI}}$ (resp. $\top_{\mathcal{CI}}$). The function, `isSubset(a1,a2)` returns `True` iff `a1` is a subset of `a2`. `isEmpty(a)` returns `True` iff `a` represents the empty set.

When comparing the transformer $\texttt{contains}^\sharp_{\texttt{CI}}$ synthesized using Amurth (with line 9 in Fig. 6a) to the one implemented in SAFE_{str} (with line 8 in Fig. 6a), we discovered that the latter was not sound.

```
1  contains♯CI(a1 : CI)(a2 : CI) : AbsBool =
2   ite(isBot(a1.l, a1.u)∨isBot(a2.l, a2.u),
3     boolBot,
4   [−] ite(isTop(a1.l, a1.u) ∨ isTop(a2.l, a2.u),  // Bug
5   [−]   boolTop,                                  // Bug
6         ite(¬isSubset(a2.l, a1.u),
7           boolFalse,
8     [-] ite(size(a2.u) ≤ 1 ∧ isSubset(a2.u, a1.l), // Bug
9     [+] ite(isEmpty(a2),                          // Fix
10            boolTrue,
11      [−]   boolTop))))                           // Bug
12      [+]   boolTop)))                            // Fix
```

(a) Abstract transformers for `contains`.

```
1  trim♯CI(a : CI) : CI =
2   ite(isBot(a.l,a.u),
3     Bot,
4     ite(isTop(a.l,a.u),
5       Top,
6       ite(size(a.u)≤1∧containsSpace(a.u),
7         [∅, ∅],
8     [-] a                          // Bug
9     [+] [removeSpace(a.l), a.u]    // Fix
10        )))
```

(b) Abstract transformers for `trim`.

Fig. 6. Bugs found and fixed in the $C\mathcal{I}$ domain for `contains` and `trim`. The lines in blue show how the synthesized transformers differ from the incorrect ones in SAFE_{str} (denoted by the lines in red).

The following example illustrates the problem. Consider two abstract values `a1` = [{'a'}, {'a','b'}] and `a2` = [{}, {'a'}]. When the $C\mathcal{I}$ abstract transformer implemented in SAFE_{str} is applied to `a1` and `a2`, it returns `BoolTrue`. For `BoolTrue` to be the correct answer, every string in γ(`a2`) must be a contiguous substring of every string in γ(`a1`). However, "aaa" $\in \gamma$(`a2`), and "ababa" $\in \gamma$(`a1`), but "aaa" is not a contiguous substring of γ(`a2`). Therefore, SAFE_{str}'s $C\mathcal{I}$ transformer has a bug: it is unsound. The transformer synthesized by Amurth is sound (and a best L-transformer with respect to the DSL given above).

Our inspection also revealed that $\texttt{contains}^{\sharp}_{\text{CI}}$ in SAFE_{str} contained a precision bug: $\texttt{contains}^{\sharp}_{CI}$ should return `boolTrue` when $a_1 = \top$ and a_2 is the empty string. In SAFE_{str}, it returns `boolTop`, which is sound but imprecise. In contrast, the transformer synthesized by Amurth (without lines 4–5, and line 12 in place of line 11) returns `boolTrue`: `isSubset`(a_2.l, a_1.u) is `true`, and a_2 is empty.

Transformer for `trim` in the $C\mathcal{I}$ Domain. The function `trim` takes a string `s` and removes all the whitespace at the beginning and the end of `s`. Amurth synthesizes the transformer $\texttt{trim}^{\sharp}_{\text{CI}}$ in Fig. 6b in 641.53s. The DSL used when synthesizing this transformer is

$$
\begin{aligned}
\textit{Transformer} &::= \lambda \texttt{a}.CI \\
CI &::= \texttt{a} \mid [\emptyset, \emptyset] \mid [LU, LU] \mid \texttt{ite}(B, CI, CI) \\
B &::= \texttt{size}(LU) \leq 1 \mid \texttt{isBot}(CI) \mid \texttt{isTop}(CI) \mid \texttt{containsSpace}(LU) \\
&\quad \mid \neg B \mid B \wedge B \mid B \vee B \\
LU &::= CI.\texttt{l} \mid CI.\texttt{u} \mid \texttt{removeSpace}(LU)
\end{aligned}
\tag{12}
$$

A program in the DSL returns an abstract string in the $C\mathcal{I}$ domain, hence the initial non-terminal is CI. The DSL contains the following operators: `size` returns the size of the argument set, `containsSpace` returns `True` iff the argument set contains a whitespace character, and `removeSpace` removes any whitespace character from the argument set.

When comparing the transformer synthesized using Amurth to the one implemented in SAFE_{str} (blue and red lines in Fig. 6b), we discovered that the latter was not sound. Consider the abstract input value `absArg` = `[{'␣','a'}, {'␣','a','b','c'}]` for which the concretization contains—among other values—the concrete string `s` = `"␣␣abc␣␣"`. On this input, $\texttt{trim}^{\sharp}_{\text{CI}}$ returns as output the abstract value `absArg'`, which is same as `absArg`. However, `trim(s)="abc"`, whereas `"abc"` $\notin$

```
1 trim♯PS(a : PS) : PS =
2  ite(isBot(a.p,a.s),
3    BOT,
4    ite(isTop(a.p,a.s),
5     TOP,
6 [-] [trimStart(a.p), trimEnd(a.s)] // Bug
7 [+] [trim(a.p), trim(a.s)]         // Fix
8 ))
```

Fig. 7. Abstract transformers for `trim` in the $\mathcal{PS}$ domain.

```
1 concat♯(a : Long)(b : Long) : Long =
2   r ← reverse(b); c ← 0; i ← 0
3   WHILE i < b
4     r ←rotateLeft(r, 1)
5     IF (a & r) ≠ 0 THEN
6 [−]   c ← c | (1 << i)   //SAFEstr
7 [+]   c ← c ^ (1 << i)   //Amurth
8     i ← i + 1
9   RETURN c
```

Fig. 8. Abstract transformers for `concat` in the $\mathcal{SH}$ domain.

γ(`absArg'`) (because `'␣'` is an element of the must-set of `absArg'`). Consequently, the abstract-transformer implementation in SAFE_{str} is unsound. The transformer synthesized by Amurth ($\texttt{trim}^\sharp_{\text{synCI}}$) does not have this issue.

Transformer for `trim` in the $\mathcal{PS}$ Domain.

An element of the $\mathcal{PS}$ domain is a pair $[p, s]$ describing the longest common prefix p and suffix s of a set of strings. Consider the abstract value a = ["␣b␣", "␣b␣"], and the string "␣b␣" $\in \gamma(a)$. A most precise transformer for `trim` on input a should output a' = ["b", "b"].

The transformer $\texttt{trim}^\sharp_{\text{PS}}$ implemented in SAFE_{str} is unsound. It incorrectly produces the output a' = ["b␣", "␣b"], whose concretization fails to contain the concrete value `trim`("␣b␣") = b. This bug is due to the statement at line 6 of Fig. 7. Using the DSL defined in Eqn. (13), Amurth is able to synthesize (in 8.52s) a correct version of $\texttt{trim}^\sharp_{\text{PS}}$: in Fig. 7, line 6 is replaced by line 7.

$$\begin{aligned} \textit{Transformer} &::= \lambda \texttt{a}.PS \\ PS &::= \texttt{a} \mid [\texttt{''}, \texttt{''}] \mid [LU, LU] \mid \texttt{ite(B, PS, PS)} \mid BOT \mid TOP \\ B &::= \texttt{isBot(PS)} \mid \texttt{isTop(PS)} \\ LU &::= PS\texttt{.p} \mid PS\texttt{.s} \mid \texttt{trim(LU)} \mid \texttt{trimStart(LU)} \mid \texttt{trimEnd(LU)} \end{aligned} \tag{13}$$

Transformer for `concat` in the $\mathcal{SH}$ Domain.

The concrete function `concat` takes two concrete strings and returns their concatenation. Pseudocode for the $\texttt{concat}^\sharp$ transformer used in SAFE_{str} is shown in Fig. 8 with line 6 (and without line 7). SAFE_{str} uses a universe U of values of size 64 (the range of the hash function). The abstract transformer uses a 64-bit `long` value a as a bit-vector encoding of a subset $A \subseteq U$—i.e., the i-th bit of a is 1 iff $i \in A$. Amurth synthesizes the abstract transformer $\texttt{concat}^\sharp_{\text{syn}}$ whose pseudocode is shown in Fig. 8 with line 7 (and without line 6). The DSL used to synthesize this transformer uses a different structure than the ones described above. In particular, it is a sketch of the loop that we expect the target function to contain, as shown in Fig. 9. From the options specified within {|...|...|...|}, the Sketch synthesizer selects an option that makes the resulting program consistent with the specification. For example, at line 10, Sketch selects from among the provided options `rotateLeft(r,1)` and `rotateRight(r,1)`. The sketch can then be completed using bitwise operators—i.e., not (¬), or (|), and (&), xor (^), left shift (<<), right shift (>>), left-rotate, right-rotate, and `reverse`. Amurth takes 609.30s to synthesize the abstract transformer for `concat` in the $\mathcal{SH}$ domain (see Tab. 2).

This transformer is particularly interesting because it contains complex logic, and an implementation trick that is hard to reason about. While we had to provide the overall structure of the program, Amurth could fill in the tricky implementation details automatically. This type of approach has been used before in program synthesis. For example, the original motivation for

Sketch itself was to synthesize tricky implementation details of user-provided implementation sketches for bit-vector operations. This particular example shows that Amurth can synthesize highly non-trivial abstract transformers.

6.1.3 Performance and Precision of the Synthesized Transformers in a Program Analyzer. We compared the performance and precision of the hand-written transformers in SAFE_{str} with the transformers synthesized by Amurth. The three bugs found in SAFE_{str} were fixed for this experiment. We ran all the benchmark verification problems provided in SAFE_{str}, and collected the same precision metric ("*imprecision index*" [Amadini et al. 2017]) used to evaluate SAFE_{str}. For each benchmark, we compared the performance with respect to the time, the number of fixpoint iterations of the analysis, the number of reachable program states, and the precision metric from SAFE_{str}.

```
1  #define W 64
2
3  bit[W] absConcat(bit[W] a, bit[W] b){
4    bit[W] r = {| a | b | reverse(a) | reverse(b)|};
5    bit[W] one = {1};
6    bit[W] c = {0};
7    bit[W] cond;
8    for(int i = 0; i < W; i++){
9      one = {1};
10     r = {| rotateLeft(r, 1) | rotateRight(r, 1) |};
11     cond = {| (a & r) | (a | r) | (a ^ r) |};
12     if( (cond) != (bit[W]){0}){
13       c = {| (c & rotateLeft(one, i))
14            | (c | rotateLeft(one, i))
15            | (c ^ rotateLeft(one, i))
16            | (c & rotateRight(one, i))
17            | (c | rotateRight(one, i))
18            | (c ^ rotateRight(one, i)) |};
19     }
20   }
21   return c;
22 }
```

Fig. 9. Sketch to synthesize abstract transformer for $\texttt{concat}^\sharp$ in $\mathcal{SH}$

The scatter plots in Figs. 10 (plots for other domains are available in extended version [Kalita et al. 2021]) show the data from runs using the SAFE_{str} transformers on the x-axis, and the data from runs using the transformers synthesized by Amurth on the y-axis. For both the hand-written transformers in SAFE_{str} and the transformers synthesized by Amurth, each run of the analyzer is given a timeout threshold of 600 seconds. An analysis run can also terminate with a SAFE_{str} imprecision-trigger exception ("Imprec") if the imprecision becomes too great. The following symbols are used in the plots to show the status of an analysis: a magenta square (■) shows that the analysis timed out; a blue triangle (▲) indicates normal termination; and a red circle (●) indicates termination due to an imprecision-trigger exception. The plots show that the transformers synthesized by Amurth have the same performance and precision as the ones implemented in SAFE_{str} (i.e., all points lie essentially on the diagonal line). There were no examples of a run using the hand-written SAFE_{str} transformers that timed out, but the corresponding run with the transformers synthesized by Amurth completing within the time limit, or vice versa.

For the $\mathcal{SH}$ domain, the abstract transformer synthesized by Amurth for `concat` (line 7 of Fig. 8) is equivalent to the hand-written transformer in SAFE_{str} (line 6 of Fig. 8). Hence, we do not show plots for the $\mathcal{SH}$ domain.

Because a singleton set of strings cannot be represented precisely in the $\mathcal{CI}$ and $\mathcal{PS}$ domains, to provide a better basis for comparing precision, we used instead their direct products with $\mathcal{CS}$ (i.e., $\mathcal{CS} \times \mathcal{CI}$ and $\mathcal{CS} \times \mathcal{PS}$). See Fig. 10 for $\mathcal{CS} \times \mathcal{CI}$; the results for $\mathcal{CS} \times \mathcal{PS}$ are available in the extended version [Kalita et al. 2021].

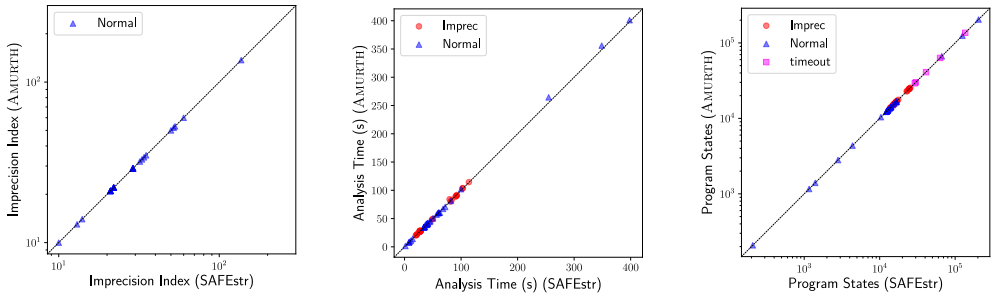


Fig. 10. Performance and precision of the synthesized transformers with the product domain $\mathcal{CS} \times \mathcal{CI}$. In the right-hand plot, all examples labeled "timeout" exceeded the timeout threshold with both the hand-written SAFE_{str} transformers and the transformers synthesized by Amurth. (There were no examples in which one set of transformers exceeded the timeout threshold and the other set did not.)

Finding [RQ1]: The time taken by Amurth to synthesize an L-transformer across all of the string-transformer experiments varies between 3.73s and 1,983.83s.
Finding [RQ2]: A manual comparison of the automatically-generated and manually-written transformers revealed that three of the manually written transformers in SAFE_{str} were unsound: $\texttt{contains}^\sharp$ for the $\mathcal{CI}$ domain, and $\texttt{trim}^\sharp$ for $\mathcal{CI}$ and $\mathcal{PS}$.

Fig. 10 (and the plots for other corresponding domains in the extended version [Kalita et al. 2021]) show that the abstract transformers synthesized by Amurth for the six string operations are empirically indistinguishable—in terms of analysis time and precision—from the manually written ones used in SAFE_{str} (after the three buggy SAFE_{str} transformers were fixed).

6.2 Case Study 2: Transformers for Three Fixed-Bitwidth Interval Domains

In this study, we used Amurth to synthesize abstract transformers in the three fixed-bitwidth interval domains described in §6.2.1, for nine different mathematical and logical operations. Concrete arithmetic operations are performed in modular arithmetic (sometimes known as "machine-integer arithmetic"). Each domain represents a set of fixed-bitwidth integers, and is parameterized on w, which denotes the number of bits in a represented integer.

6.2.1 Fixed-Bitwidth Interval Domains. Our study considered the three fixed-bitwidth interval domains summarized below [Navas et al. 2012].

Unsigned-Integer Intervals ($\mathcal{A}_{\texttt{uintv}}$). An element in the w-bit unsigned-integer interval domain is either $\bot$, which denotes the empty set, or is from the set $\{[a, b] \mid 0 \leq a \leq b < m\}$, where $m = 2^w$. The concretization function (γ_{u}) is defined as, $\gamma_{\text{u}}([a, b]) = \{a, a + 1, \ldots, b - 1, b\}$.

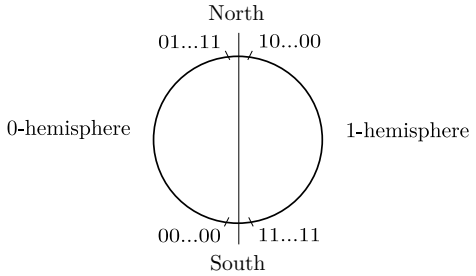


Fig. 11. Wrapped Interval ($\mathcal{W}$) number circle.

Signed-Integer Intervals ($\mathcal{A}_{\texttt{sintv}}$). An element in the w-bit signed-integer interval domain is either $\bot$, which denotes the empty set, or is from the set $\{[a, b] \mid -m \leq a \leq b < m\}$, where $m = 2^{w-1}$. Negative numbers are interpreted in their two's-complement representation. The concretization function γ_s is defined as, $\gamma_s([a, b]) = \{a, a + 1, \ldots, b - 1, b\}$.

Table 3. Time, in seconds, to synthesize abstract transformers for the fixed-bitwidth interval domains. Templates were used in synthesizing the abstract transformers of the shaded cells in the table.

Domain	Arith. Ops.			Bitwise Ops.					
	add	sub	mul	and	or	xor	shift left	arithmetic shift right	logical shift right
Unsigned	133.70	99.07	1,449.85	14.54	17.36	1,095.94	12.52	6.15	12.88
Signed	607.63	213.17	1,287.83	284.42	958.16	1,234.94	11.78	7.61	4.23
Wrapped	858.60	739.47	880.12	1,360.98	1,311.37	962.13	198.90	598.62	548.72

Wrapped Intervals ($\mathcal{W}$). An element in the wrapped interval is either $\bot$, which denotes the empty set, $\top$, which represents the set $\{-2^{w-1} \le a < 2^{w-1}\}$, or it is represented by $[a, b]$, where a, b are w-bit bit-vectors such that $a \not\equiv (b+1) \bmod 2^w$. This domain is a sign-agnostic domain. Wrapped intervals are permitted to cross either or both of the "North pole" and "South pole" shown in Fig. 11.[1] The concretization function (γ_w) for the wrapped interval domain is defined as follows, where $\le_l$ is lexicographic ordering on bit-vectors.

$$\gamma_w([a,b]) = \begin{cases} \{a, \ldots, b\} & \text{if } a \le_l b \\ \{0^w, \ldots, b\} \cup \{a, \ldots, 1^w\} & \text{otherwise.} \end{cases} \tag{14}$$

For example, for 3-bit intervals, $\gamma([111, 101]) = \{000, 001, 010, 011, 100, 101, 111\}$ illustrates the second case of the Eqn. (14).

6.2.2 Abstract Transformers for Fixed-Bitwidth Interval Domains. For each of the three domains, we used Amurth to synthesize abstract transformers for nine operations (`add`, `sub`, `mul`, `and`, `or`, `xor`, `shl`, `ashr`, and `lshr`). Amurth takes 4.23s to 1,449.85s to synthesize a best L-transformer; see Tab. 3. Table showing detailed results for the fixed-bitwidth interval-domain experiments are available in the extended version [Kalita et al. 2021]. While in most cases, merely providing a DSL grammar was enough (indicated by the unshaded cells in Tab. 3), for some of the more involved transformers, we had to provide a *template*—a sketch of the high-level implementation—and Amurth was able to fill in the details (see the shaded cells). The transformers for which we needed an implementation sketch are discussed in §6.2.4.

When we tried to synthesize the abstract transformers for `xor` for the $\mathcal{A}_{\texttt{sintv}}$ and $\mathcal{W}$ domains using the auxiliary functions used by Navas et al., Amurth failed. A closer examination revealed that there was a bug in the Navas et al. implementation of the `minAnd` auxiliary function: in two places, they used a bitwise negation($\sim$) instead of an arithmetic negation ($-$) as shown in Fig. 12. After we fixed this bug, all of the L-transformers synthesized by Amurth are semantically equivalent to those provided in the implementation that accompanies the Navas et al. paper [2012].

6.2.3 Abstract transformer for multiplication for $\mathcal{A}_{\texttt{uintv}}$ and $\mathcal{A}_{\texttt{sintv}}$. For this operation, whenever there is a possibility of an overflow, the transformer returns $\top$. An overflow is detected as follows:

$$\begin{aligned} \texttt{overflow\_mul}^\sharp(a_1 : \mathcal{A}_{\texttt{uintv}}, a_2 : \mathcal{A}_{\texttt{uintv}}) = \quad & \texttt{overflows}(a_1.l * a_2.l) \vee \texttt{overflows}(a_1.l * a_2.r) \\ \vee\ & \texttt{overflows}(a_1.r * a_2.l) \vee \texttt{overflows}(a_1.r * a_2.r) \end{aligned}$$

where $\texttt{overflows}(x * y)$ returns `true` whenever the unsigned multiplication of x and y overflows. A similar case holds for the case of signed multiplication, but in this case, the overflow is checked with an overloaded version of `overflows()` that checks overflows of signed multiplications.

[1] In contrast, unsigned-integer and signed-integer intervals are allowed to cross only the North and South poles, respectively.

```
1 llvm::APInt minAnd(llvm::APInt a, const llvm::APInt& b, llvm::APInt c, const llvm::APInt& d) {
2     llvm::APInt m = llvm::APInt::getOneBitSet(a.getBitWidth(), a.getBitWidth() - 1);
3     while (m != 0) {
4         if ((~a & ~c & m)) {
5         [-] llvm::APInt temp = (a | m) & ~m; // Bug
6         [+] llvm::APInt temp = (a | m) &  - m; // Fix
7             if (temp <= b) {
8                 a = temp;
9                 break;}
10        [-] temp = (c | m) & ~m; // Bug
11        [+] temp = (c | m) &  - m; // Fix
12            if (temp <= d) {
13                c = temp;
14                break;}
15        }
16        m = m.lshr(1);
17    }
18    return a & c;
19 }
```

Fig. 12. Buggy implementation of `minAnd` from the Navas et al. implementation.

To complete the transformer, we used Amurth to synthesize the case where there was no overflow. In essence, we used the following transformer template:

$$\texttt{mul}^\sharp(a_1, a_2) = \begin{cases} \top & \texttt{overflow\_mul}^\sharp(a_1, a_2) = true \\ \texttt{mul}^\sharp_{no} & \texttt{overflow\_mul}^\sharp(a_1, a_2) = false \end{cases} \tag{15}$$

We used the following DSL for both the signed and unsigned domains:

$$\begin{aligned} \mathit{Transformer} &::= \lambda \texttt{a}.[E, E] \\ E &::= \texttt{a.l} \mid \texttt{a.r} \mid 0 \mid -E \mid \texttt{INTMAX} \mid \texttt{INTMIN} \mid \texttt{min}(E, E) \mid \texttt{max}(E, E) \mid \texttt{mul}(E, E) \end{aligned} \tag{16}$$

Eqn. (17) shows the abstract transformer synthesized by Amurth for the unsigned-integer interval domain ($\mathcal{A}_{\texttt{uintv}}$). In this case, the abstract transformer is quite simple, multiplying the corresponding left and right limits of the multiplicands.

$$\texttt{mul}^\sharp_{no}(\texttt{a}_1 : \mathcal{A}_{\texttt{uintv}}, \texttt{a}_2 : \mathcal{A}_{\texttt{uintv}}) : \mathcal{A}_{\texttt{uintv}} = [\texttt{mul}(\texttt{a}_2\texttt{.l},\ \texttt{a}_1\texttt{.l}),\ \texttt{mul}(\texttt{a}_2\texttt{.r},\ \texttt{a}_1\texttt{.r})] \tag{17}$$

In contrast, the case of signed multiplication is more involved. Eqn. (18) shows the abstract transformer synthesized by Amurth for the signed-integer interval domain ($\mathcal{A}_{\texttt{sintv}}$), again using the DSL from Eqn. (16).

$$\texttt{mul}^\sharp_{no}(\texttt{a}_1 : \mathcal{A}_{\texttt{sintv}}, \texttt{a}_2 : \mathcal{A}_{\texttt{sintv}}) : \mathcal{A}_{\texttt{sintv}} = \left[\texttt{min}\begin{pmatrix}\texttt{min}(\texttt{mul}(\texttt{a}_2\texttt{.r},\ \texttt{a}_1\texttt{.r}),\ \texttt{mul}(\texttt{a}_1\texttt{.r},\ \texttt{a}_2\texttt{.l})),\\ \texttt{min}(\texttt{mul}(\texttt{a}_1\texttt{.l},\ \texttt{a}_2\texttt{.r}),\ \texttt{mul}(\texttt{a}_2\texttt{.l},\ \texttt{a}_1\texttt{.l}))\end{pmatrix}, \texttt{max}\begin{pmatrix}\texttt{max}(\texttt{mul}(\texttt{a}_1\texttt{.l},\ \texttt{a}_2\texttt{.l}),\ \texttt{mul}(\texttt{a}_2\texttt{.r},\ \texttt{a}_1\texttt{.r})),\\ \texttt{max}(\texttt{mul}(\texttt{a}_1\texttt{.r},\ \texttt{a}_2\texttt{.l}),\ \texttt{mul}(\texttt{a}_1\texttt{.l},\ \texttt{a}_2\texttt{.r}))\end{pmatrix}\right] \tag{18}$$

This abstract transformer takes the product of every pair of bounds from the two intervals, and chooses the minimum element as the left bound of the resultant interval. Similarly, it picks the maximum element as the right bound of the resultant interval.

Note that, for all the above domains, although Amurth was provided with the same DSL syntax shown in Eqn. (16), the supplied semantics of the constructs differed, according to the domain.

That is, depending on the intended domain, `a.l` and `a.r` were interpreted as signed or unsigned integers; `INTMAX` and `INTMIN` were interpreted according to the bitwidth and signedness under consideration; etc.

6.2.4 Abstract transformers for `xor`. The abstract transformer for `xor` in the $\mathcal{W}$ domain is quite complex because it involves nested loops. However, the "high-level" algorithm is quite intuitive and proceeds by splitting intervals. If 0^k falls in the interval $[a, b]$, where a, b are k-bit bit-vectors, a split at zero (South pole) in $\mathcal{W}$ will split the interval $[a, b]$ into two intervals $[a, 1^k]$ and $[0^k, b]$.

We show the sketch for this high-level structure in Fig. 13, where the the i^{th} *hole* to be filled by Amurth is denoted by "`?i`". Both the multiplicands are split at 0 (lines 2–3), and the sketch loops through each possible pair of interval segments generated (lines 5–6). For each pair, it specifies *holes* for functions to be synthesized (lines 7–8) that constitute the lower and upper bounds of the interval. Finally, the intervals corresponding to all such segments are joined to construct the final interval (line 9).

```
1 W xorGen (W a1, W a2) {
2   W s1[] = intervalSplitAtZero(a1.l,a1.r);
3   W s2[] = intervalSplitAtZero(a2.l,a2.r);
4   W result = ⊥;
5   for(int i = 0; i < s1.size(); i++){
6     for(int j = 0; j < s2.size(); j++){
7       int res0 = ?1(s1[i], s2[j]);
8       int res1 = ?2(s1[i], s2[j]);
9       result = Join_W([res0, res1], result);
10 }}}
```

Fig. 13. Sketch for $\texttt{xor}^\sharp$ in $\mathcal{W}$ and $\mathcal{A}_{\texttt{sintv}}$. Function `intervalSplitAtZero` splits the interval at 0, if interval contains 0.

Amurth synthesizes the functions shown in Eqns. (20) and (21) for the two holes in Fig. 13, using the following grammar:

$$\begin{aligned}
&\textit{Transformer} ::= \lambda \mathsf{a}_1, \mathsf{a}_2.\ EX\\
&EX \ ::= \ \mathsf{minOr}(B,B,B,B) \mid \mathsf{maxOr}(B,B,B,B) \mid \mathsf{minAnd}(B,B,B,B) \mid \mathsf{maxAnd}(B,B,B,B)\\
&\quad \mid \ \mathsf{minOr}(0,EX,0,EX) \mid \mathsf{maxOr}(0,EX,0,EX) \mid \mathsf{minAnd}(0,EX,0,EX) \mid \mathsf{maxAnd}(0,EX,0,EX)\\
&\quad \mid \ \mathsf{or}(EX,EX) \mid \mathsf{and}(EX,EX)\\
&B \ ::= \ a_1.l \mid a1_1.r \mid a_2.l \mid a_2.r \mid \sim B
\end{aligned} \tag{19}$$

$$?1(\mathsf{a}_1 : \mathcal{W}, \mathsf{a}_2 : \mathcal{W}) : \mathcal{W} = \mathsf{or}\begin{pmatrix} \mathsf{minAnd}(a_1.l, a_1.r, \sim a_2.r, \sim a_2.l), \\ \mathsf{minAnd}(\sim a_1.r, \sim a_1.l, a_2.l, a_2.r) \end{pmatrix} \tag{20}$$

$$?2(\mathsf{a}_1 : \mathcal{W}, \mathsf{a}_2 : \mathcal{W}) : \mathcal{W} = \mathsf{maxOr}\begin{pmatrix} 0,\ \mathsf{maxAnd}(a_1.l, a_1.r, \sim a_2.r, \sim a_2.l), \\ 0, \mathsf{maxAnd}(\sim a_1.r, \sim a_1.l, a_2.l, a_2.r) \end{pmatrix} \tag{21}$$

Using the sketch (Fig. 13) and the same DSL (Eqn. (19)), with numbers and operations interpreted as signed integers Amurth ends up synthesizing the same transformers for the signed domain ($\mathcal{A}_{\texttt{sintv}}$). For the unsigned domain ($\mathcal{A}_{\texttt{uintv}}$), the transformer is simpler: Amurth could synthesize the $\mathcal{A}_{\texttt{uintv}}$ transformer just from the DSL (Eqn. (19))—with numbers and operations interpreted as unsigned—without any need for a sketch. Eqn. (22) (`minXor`) and Eqn. (23) (`maxXor`) show the minimum and maximum limits for the `xor` abstract transformer in the $\mathcal{A}_{\texttt{uintv}}$ domain, respectively.

$$\mathsf{minXor}(\mathsf{a}_1 : \mathcal{A}_{\texttt{uintv}}, \mathsf{a}_2 : \mathcal{A}_{\texttt{uintv}}) : \mathcal{A}_{\texttt{uintv}} = \mathsf{or}\begin{pmatrix} \mathsf{minAnd}(a_1.l, a_1.r, \sim a_2.r, \sim a_2.l), \\ \mathsf{minAnd}(\sim a_1.r, \sim a_1.l, a_2.l, a_2.r) \end{pmatrix} \tag{22}$$

$$\texttt{maxXor}(\texttt{a}_1 : \mathcal{A}_{\texttt{uintv}}, \texttt{a}_2 : \mathcal{A}_{\texttt{uintv}}) : \mathcal{A}_{\texttt{uintv}} = \texttt{maxOr}\left(\begin{array}{l} 0,\ \texttt{maxAnd}(a_1.l, a_1.r, \sim a_2.r, \sim a_2.l), \\ 0, \texttt{maxAnd}(\sim a_1.r, \sim a_1.l, a_2.l, a_2.r) \end{array}\right) \tag{23}$$

Finding [RQ1]: The time taken by Amurth to synthesize an L-transformer across all of the fixed-bitwidth interval-transformer experiments varies between 4.23s and 1,449.85s.
Finding [RQ2]: Our experiments using Amurth uncovered bugs in the abstract transformers implemented by Navas et al. for `xor` for the $\mathcal{A}_{\texttt{sintv}}$ and $\mathcal{W}$ domains. The two bugs had a single root cause, which was that an auxiliary function used in their interval-analysis tool was unsound due to a mistranscription of code from *Hacker's Delight* [Warren 2012], which came to light when we used the Navas et al. auxiliary function, and Amurth failed to synthesize a correct abstract transformer.

After the bug in the auxiliary function was fixed, all of the synthesized abstract transformers for the three kinds of interval domains are sound, precise, and semantically equivalent to those provided in the implementation that accompanies the paper by Navas et al..

6.3 Experience with Designing DSLs

In this section, we discuss more about the design of DSLs for transformer synthesis, using the DSLs shown in Eqns. (11), (12), and (13) as examples.

Some aspects of DSLs are common across the DSLs for different operations and different abstract domains. For instance, all the DSLs in Eqns. (11), (12), and (13) provide a way to check whether an abstract value is $\bot$ or $\top$, and to perform different actions depending on the outcome.

```
1 contains♯syn(a1 : CI)(a2 : CI) : AbsBool =
2  ite(isBot(a1.l, a1.u) ∨ isBot(a2.l, a2.u),
3    boolBot,
4    ite(isTop(a1.l, a1.u) ∧ isSubset(a2.u, a1.l),
5      boolTrue,
6      ite(¬isSubset(a2.l, a1.u),
7        boolFalse,
8        boolTop )))
```

Fig. 14. Another abstract transformer for `contains` in the $\mathcal{CI}$ domain.

Other parts of a DSL typically reflect the properties that are observable in the abstract domain for which the DSL will be used for synthesizing transformers. For instance, in the case of the DSL for the `contains` operator in the character inclusion ($\mathcal{CI}$) domain (Eqn. (11)), it is natural to add constructs such as `isSubset()` and `isEmpty()` to compare sets of characters. In the case of the DSL for the `trim` operation in $\mathcal{CI}$ domain (Eqn. (12)), we use constructs such as `containsSpace()` and `removeSpace()` to handle the space (␣) character. Similarly, Eqn. (13) is also a DSL for the `trim` operation, but for use with the prefix-suffix ($\mathcal{PS}$) domain. Eqn. (13) is similar to that of Eqn. (12), but instead of operations on sets, here operations are on strings, such as `trimStart()` and `trimEnd()`, which return strings in which spaces have been removed from the beginning and the end, respectively, of the argument string.

We now turn to what we observed when Amurth is supplied with a DSL that is a misfit for the problem at hand. Such misfitting can take two forms: one can have "too many" constructs (§6.3.1), or "too few" constructs (§6.3.2).

6.3.1 DSLs with "too many" constructs. Consider the DSL shown in Eqn. (24).

$$\begin{array}{rcl} \textit{Transformer} & ::= & \lambda \texttt{a}.[E, E] \\ E & ::= & \texttt{a.l} \mid \texttt{a.r} \mid 0 \mid -E \mid +\infty \mid -\infty \mid E + E \mid E - E \mid E * E \\ & & \mid \texttt{min}(E, E) \mid \texttt{max}(E, E) \mid \texttt{E * E * E} \mid \texttt{pow(E, E)} \end{array} \tag{24}$$

Eqn. (24) is similar to the DSL shown in Eqn. (2),except that it has some extra constructs, which are highlighted in yellow. We tried to synthesize an abstract transformer for `abs` with this DSL variant. We ran Amurth three times with a timeout of 6,000 seconds per call on Sketch—ten times the usual timeout. The median time of the three runs was 6010.7 seconds. In all three runs, Amurth could only synthesize a sound but imprecise interval transformer for absolute value in which the right-hand limit of the return value is always positive infinity ($+\infty$). An investigation of the reason for this result revealed that Amurth exceeded the timeout threshold in calls to CheckPrecision, causing it to synthesize an imprecise solution.

6.3.2 DSLs with "not enough" constructs. Now consider the DSL defined by Eqn. (11) but with the production $B ::= \texttt{isEmpty}(CI)$ removed, and suppose that we ask Amurth to synthesize an abstract transformer for the `contains` operation in the $\mathcal{CI}$ domain. In this case, Amurth synthesizes the abstract transformer shown in Fig. 14. This abstract transformer is a best L-transformer, where L is the language defined by Eqn. (11) without the production $B ::= \texttt{isEmpty}(CI)$.

The absence of `isEmpty()` in the DSL causes Fig. 14 to be less precise than the (corrected) abstract transformer shown in Fig. 6a (without line 8 and with line 9). Concretely, the empty string is contained in every string. The corrected abstract transformer in Fig. 6a returns `boolTrue` whenever a_2 is the empty string—see Fig. 6a, line 10. In contrast, the transformer in Fig. 14 returns `boolTrue` only in the case that $\texttt{a}_2$ is the empty string and $\texttt{a}_1$ is $\top$: in line 4 it checks whether a_1 is $\top$ and $a_2.u$ is subset of $a_1.l$; when a_1 is $\top$ and a_2 is the empty string, $a_1.l$ and $a_2.u$ both hold the empty set, and `boolTrue` is returned (line 5). However, when a_1 is not $\top$, due to the absence of `isEmpty()` in the DSL, the transformer cannot check whether argument a_2 is the empty string, and thus returns the sound answer `boolTop`.

7 RELATED WORK

The related work closest to ours was discussed in §1; here we discuss some other related work.

Program synthesis has recently gained a lot of attention, and has found applications in diverse areas. CEGIS [Solar-Lezama 2013] is a popular synthesis strategy. In our work, we interleave two CEGIS loops to handle competing objectives, considering a "negative-example" classification as a soft constraint, allowing `MaxSatSynthesize` to find a better classification. Work on synthesizing data-structure invariants [Miltner et al. 2020] also deals with two competing objectives—weakening/strengthening candidate invariants—for which they employ three CEGIS loops.

Several papers by Reps, Sagiv, Yorsh, Thakur, and others address (explicitly or implicitly) the problem of creating best abstract transformers for a variety of abstract-interpretation frameworks [Reps et al. 2004; Reps and Thakur 2016; Thakur et al. 2012, 2015; Thakur and Reps 2012] (with slightly different requirements among the different papers). Elder et al. [2014] gave a method for creating best abstract transformers for the abstract domain of conjunctions of bit-vector equalities. The main differences with our work is that (i) those papers use positive examples only, and (ii) they do not allow the user to specify a DSL. Our algorithm uses both positive and negative examples, and the user can supply a DSL of their own design.

Wang et al. [2018] presented an approach for learning abstract transformers for a given abstract domain. There are major differences between their approach and ours. For them, (i) abstract transformers are expressed in a specific language: conjunctions of a (learned) set of fixed predicates over affine expressions, and (ii) the operations of the DSL are the concrete operations for which the system tries to find suitable abstract transformers. In contrast, with Amurth the user (i) supplies their own DSL in which the abstract transformer is to be expressed, and (ii) provides a logical specification of the concrete operation for which an abstract transformer is sought.

Bielik et al. [2017] present a method to learn program analyzers from data. Their method attempts to automatically learn program-analysis inference rules for program primitives (such as assignment statements, pointer dereferences, etc.), with an emphasis on learning corner-cases for such rules. The user must supply training data of the form ⟨*program*, *analysis output*⟩. The algorithm finds patterns to apply to program abstract-syntax trees to produce analysis results that match the dataset as closely as possible. It also uses program-mutation operations to test the learned rules, and to augment the dataset in a CEGIS loop. Transfer functions are expressed as decision trees, learned using a modification of the ID3 algorithm. Because the transfer functions cannot use arithmetic/bitwise operations, their method is suited to *selecting* from a set of facts, rather than *constructing* arbitrary abstract values. Bielik et al. handle precision by attempting to minimize a cost function on the dataset. They do not attempt to verify that their solution is indeed precise. While their approach works well for pointer analysis, and for some forms of type analysis and constant propagation, due to (i) the inability to use arithmetic, and (ii) the way precision is handled, their technique cannot generate abstract transformers for the domains considered in our experiments.

A recent paper [Wang et al. 2021] is a synthesis-based technique for creating sound abstract transformers using learned predicates; however, their method of using Datalog query containment corresponds to CheckSoundness only. They have no analogue of CheckPrecision, and hence no mechanism for controlling the precision of the transformers that they obtain.

Prabhu et al. [2021] synthesize code specifications using CHC solvers. The problem they tackle shares some commonalities with ours: a specification has to be not only sound but precise (e.g., True is not a very useful specification). To synthesize precise specifications, their algorithm uses CHC solvers to strengthen the synthesized specification in a CEGIS loop. While this aspect shares some structure with our precision queries, the task solved by our algorithm is much harder as it requires synthesizing programs over a DSL instead of logical specifications in a given theory. Moreover, the problem that they address is dual to ours: their work goes from code to logic, whereas our work goes from logic to code.

Astorga et al. [2021] use a test generator to generate positive examples to synthesize a contract that is sound with respect to the examples. There are two main differences between their work and ours. (i) They only use positive examples, whereas we use both positive and negative ones; negative examples are the key to synthesizing best L-transformers. (ii) Their notion of "tight" is with respect to a syntactic restriction on the logic in which the contract is to be specified. That is, their system works with a specific logic fragment—for example, the contract is to be specified by a formula that uses at most k disjuncts. In contrast, in our work the user-specified DSL provides another "knob," which can be used to explore the trade-off between precise solutions and pragmatic solutions. There has also been the use of test generators (like fuzzers) to analyze programs for *closed-box* functions (program components whose logical specifications are not available) [Lahiri and Roy 2022; Muduli and Roy 2022; Pandey et al. 2019]. Amurth can also be targeted for such applications to infer abstract transformers for such *closed-box* functions by driving the soundness check by a test generator; we intend to pursue such directions in the future.

Data-Availability Statement. We provide a complete Docker image containing the source code of Amurth, and relevant dependencies for the experiments, on Zenodo [Kalita et al. 2022].

Acknowledgments. Supported, in part, by a gift from Rajiv and Ritu Batra; by multiple Facebook Research Awards; by a Microsoft Faculty Fellowship; by NSF under grants 1420866, 1763871, 1750965, 1918211, and 2023222; and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. 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.

REFERENCES

Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. 2017. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–57.

Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing Contracts Correct Modulo a Test Generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485481

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, Simon Fraser University, and Roberto Sebastiani. 2009. Software Model Checking via Large-Block Encoding. In *2009 Formal Methods in Computer-Aided Design*. 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 233–253.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. 2014. Abstract Domains of Affine Relations. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 11 (Oct. 2014), 73 pages. https://doi.org/10.1145/2651361

Susanne Graf and Hassen Saidi. 1997. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification*, Orna Grumberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–83.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2021. Synthesizing Abstract Transformers. https://doi.org/10.48550/ARXIV.2105.00493 Extended version of the current work.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. *Synthesizing Abstract Transformers*. https://doi.org/10.5281/zenodo.7068650 Software artifact of the current work.

Sumit Lahiri and Subhajit Roy. 2022. Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 352–364. https://doi.org/10.1145/3533767.3534381

Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. 2005. Predicate Abstraction via Symbolic Decision Procedures. In *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 24–38.

Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. 2006. SMT Techniques for Fast Predicate Abstraction. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–437.

Chu Min Li and Felip Manya. 2009. MaxSAT, Hard and Soft Constraints. *Handbook of satisfiability* 185 (2009), 613–631.

Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–217.

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15.

Sujit Kumar Muduli and Subhajit Roy. 2022. Satisfiability Modulo Fuzzing: A Synergistic Combination of SMT Solving and Fuzzing. In *Proceedings of the 2022 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2022)*. Association for Computing Machinery. https://doi.org/10.1145/3563332

Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2012. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code, Implementation available at: https://github.com/sav-tools/wrapped-intervals. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–130.

Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 228–238. https://doi.org/10.1145/3293882.3330554

Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification Synthesis with Constrained Horn Clauses. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. https://doi.org/10.1145/3453483.3454104

Thomas Reps, Mooly Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer Berlin Heidelberg, Berlin,

Heidelberg, 252–266.

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. https://doi.org/10.1007/978-3-662-49122-5_1

Erika Rice Scherpelz, Sorin Lerner, and Craig Chambers. 2007. Automatic Inference of Optimizer Flow Functions from Semantic Meanings. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 135–145. https://doi.org/10.1145/1250734.1250750

Armando Solar-Lezama. 2013. Program Sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (01 Oct 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Aditya V. Thakur, Matt Elder, and Thomas W. Reps. 2012. Bilateral Algorithms for Symbolic Abstraction. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 111–128. https://doi.org/10.1007/978-3-642-33125-1_10

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. https://doi.org/10.1016/j.entcs.2015.02.003 Fourth Workshop on Tools for Automatic Program Analysis (TAPAS 2013).

Aditya V. Thakur and Thomas W. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 174–192. https://doi.org/10.1007/978-3-642-31424-7_17

Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 810–822. https://doi.org/10.1109/ICSE43902.2021.00079

Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426.

Henry S. Warren. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.