Absynthe: Abstract Interpretation-Guided Synthesis

SANKHA NARAYAN GURIA, University of Maryland, USA
JEFFREY S. FOSTER, Tufts University, USA
DAVID VAN HORN, University of Maryland, USA

Synthesis tools have seen significant success in recent times. However, past approaches often require a complete and accurate embedding of the source language in the logic of the underlying solver, an approach difficult for industrial-grade languages. Other approaches couple the semantics of the source language with purpose-built synthesizers, necessarily tying the synthesis engine to a particular language model. In this paper, we propose Absynthe, an alternative approach based on user-defined abstract semantics that aims to be both lightweight and language agnostic, yet effective in guiding the search for programs. A synthesis goal in Absynthe is specified as an abstract specification in a lightweight user-defined abstract domain and concrete test cases. The synthesis engine is parameterized by the abstract semantics and independent of the source language. Absynthe validates candidate programs against test cases using the actual concrete language implementation to ensure correctness. We formalize the synthesis rules for Absynthe and describe how the key ideas are scaled-up in our implementation in Ruby. We evaluated Absynthe on SyGuS strings benchmark and found it competitive with other enumerative search solvers. Moreover, Absynthe's ability to combine abstract domains allows the user to move along a cost spectrum, *i.e.,* expressive domains prune more programs but require more time. Finally, to verify Absynthe can act as a general purpose synthesis tool, we use Absynthe to synthesize Pandas data frame manipulating programs in Python using simple abstractions like types and column labels of a data frame. Absynthe reaches parity with AutoPandas, a deep learning based tool for the same benchmark suite. In summary, our results demonstrate Absynthe is a promising step forward towards a general-purpose approach to synthesis that may broaden the applicability of synthesis to more full-featured languages.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: program synthesis, abstract interpretation

ACM Reference Format:
Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 171 (June 2023), 24 pages. https://doi.org/10.1145/3591285

1 INTRODUCTION

In recent years, there has been a significant interest in automatically synthesizing programs from high-level specifications, which often take the form of logical formulas [Feng et al. 2018], type signatures [Polikarpova et al. 2016], or even input-output examples [Frankle et al. 2016]. Program synthesis has seen significant success in domains such as spreadsheets [Gulwani 2011], compilers [Phothilimthana et al. 2019] or even database access programs [Guria et al. 2021]. Much of the prior work, however, requires a complete and accurate embedding of the source language in the logic of the underlying solver the synthesis tool uses. These often range from symbolic

Authors' addresses: Sankha Narayan Guria, University of Maryland, College Park, Maryland, 20742, USA, sankha@cs.umd.edu; Jeffrey S. Foster, Tufts University, Medford, Massachusetts, 20742, USA, jfoster@cs.tufts.edu; David Van Horn, University of Maryland, College Park, Maryland, 20742, USA, dvanhorn@cs.umd.edu.

2475-1421/2023/6-ART171
https://doi.org/10.1145/3591285

execution [Torlak and Bodík 2014], counter-example guided synthesis [Solar-Lezama 2013], or over-approximate semantics as predicates [Feng et al. 2017; Kim et al. 2021; Polikarpova et al. 2016] (often requiring termination measures and additional predicates for verifcation). This is infeasible for many industrial-grade languages such as Ruby or Python. Other approaches are strongly coupled with the semantics of the source language with purpose-built solvers [Reynolds et al. 2015], but this necessarily ties the synthesis engine to the particular language model used.

In this paper, we propose Absynthe, an alternative approach based on user-defined abstract semantics that aims to be both lightweight and language agnostic. The abstract semantics are lightweight to design, simplifying away inconsequential language details, yet effective in guiding the search for programs. The synthesis engine is parameterized by the abstract semantics [Cousot and Cousot 1977] and independent of the source language. In Absynthe, users define a synthesis problem via concrete test cases and an abstract specification in some user-defined abstract domain. These abstract domains, and the semantics of the target language in terms of the abstract domains, are written by the user in a DSL. Moreover, the user can define multiple simple domains, each defining a partial semantics of the language, which they can combine together as a product domain automatically. Absynthe uses these abstract specifications to automatically guide the search for the program using the abstract semantics. The key novelty of Absynthe is that it separates the search procedure from the definition of abstract domains, allowing the search to be guided by any user-defined domain that fits the synthesis task. More specifically, the program search in Absynthe begins with a hole tagged with an abstract value representing the method's expected return value. At each step, Absynthe substitutes this hole with expressions, potentially containing more holes, until it builds a concrete expression without any holes. Each concrete expression generated is finally tested in the reference interpreter to check if it passes all test cases. A program that passes all tests is considered the solution. (§ 2 gives a complete example of Absynthe's synthesis strategies).

We formalize Absynthe for a core language $\mathcal{L}_f$ and define an abstract interpreter for $\mathcal{L}_f$ in terms of abstract transformer functions. Next, we describe a DSL $\mathcal{L}_{meta}$ used to define these abstract transformers. Notably, as Absynthe synthesizes terms at each step, it creates holes tagged as abstract variables $\tilde{x}$, *i.e.,* holes which will be assigned a fixed abstract values later. We give evaluation rules for these transformers written in $\mathcal{L}_{meta}$, that additionally narrows these abstract variables to sound range of abstract values. For example, given a specification that requests Pandas programs that should evaluate to a data frame, a term $(\square : \tilde{x}_1).\mathsf{query}(\square : \tilde{x}_2)$ is a viable candidate that queries a data frame. However, semantics of $\mathcal{L}_{meta}$ help with constraining the bounds on $\tilde{x}_1$ and $\tilde{x}_2$ such that these holes are substituted by values of a DataFrame and String respectively. Finally, we present the synthesis rules used by Absynthe to generate such terms. Specifically, we discuss how Absynthe specializes term generation based on the properties of the domain, such as a finite domain enables enumeration through domain, or a domain that can be lifted to solvers can use solver-backed operations, or domains expressed as computations not supported by dedicated SMT solvers. (§ 3 discusses our formalism).

We implemented Absynthe as a core library in Ruby, that provides the necessary supporting classes to implement user-specific abstract domains and abstract interpretation semantics. It further integrates automatic support for $\top$ and $\bot$ values and abstract variables , as well as ProductDomain to combine the individual domains point-wise. The Absynthe implementation has interfaces to call a concrete interpreter with a generated program to check if a program satisifies the input/output examples. Finally, we also discuss some optimizations to scale Absynthe for practical problems, such as caching small terms and guessing partial programs based on testing predicates on the input/output examples, and some limitations of the tool. (§ 4 discusses our implementation).

We evaluate Absynthe as a general-purpose tool on a diverse set of synthesis problems while being at par on performance with state-of-the-art tools. We first use Absynthe to solve the

	id	valueA
0	255	1141
1	91	1130
2	347	830
⋮	⋮	⋮
8	225	638
9	257	616

(a) First data frame (arg0)

	id	valueB
0	255	1231
1	91	1170
2	5247	954
⋮	⋮	⋮
12	211	575
13	25	530

(b) Second data frame (arg1)

	id	valueA	valueB
0	255	1141	1231
1	91	1130	1170
2	347	830	870
5	159	715	734
8	225	638	644

(d) Output data frame

```
1  def goal(arg0, arg1, arg2):
2    return arg0.merge(arg1, on=['id']).query(arg2)
```

(c) Synthesized program

```
1  'valueA ≠ valueB'
```

(e) Query string (arg2)

Fig. 1. Data Frame Manipulation Example [Bavishi et al. 2019]. The synthesis goal is to produce (d) given inputs (a), (b), and a query string (e). ABSYNTHE synthesizes the solution (c).

SyGuS strings benchmarks [Alur et al. 2017a] using simple domains such as string prefix, string suffix, and string length to guide the search. Though ABSYNTHE operates with minimal semantic information about SyGuS programs, it still performs similar to enumerative search solvers such as EUPHONY [Alur et al. 2017b], solving most benchmarks in less than 7 seconds. SMT solvers such as CVC4, or BLAZE that rely on precise abstractions perform much faster than ABSYNTHE, but require large specification effort. We further evaluate the impact of our performance optimizations and verify that ABSYNTHE's synthesis cost adjusts with the expressiveness of the domain. More specifically, the string prefix and suffix domains written in Ruby generate a concrete candidate 0.41ms average, whereas string length domain being a solver-aided domain takes around 16.93ms per concrete candidate on average due to calls to Z3. Next, we use ABSYNTHE synthesize an unrelated benchmark suite, for which it is harder to write precise formal semantics—Python programs that use Pandas, a data frame manipulation library. We evaluate ABSYNTHE on the AUTOPANDAS benchmarks [Bavishi et al. 2019], a suite of Pandas data frame manipulating programs in Python. The AUTOPANDAS tool trains deep neural network models to synthesize Pandas programs. ABSYNTHE is at par with AUTOPANDAS, including a significant overlap in the benchmarks both tools can solve, despite using simple semantics such types and column labels of a data frame while running on a consumer Macbook Pro without specialized hardware requirements. (§ 5 discusses our evaluation).

In summary, we think ABSYNTHE represents an important step forward in the design of practical synthesis tools that provide lightweight formal guarantees while ensuring correctness from tests.

2 OVERVIEW

In this section, we demonstrate ABSYNTHE by using it to synthesize data frame manipulation programs in Python using the Pandas library [Reback et al. 2022]. In this example, we abstract data frames as sets of column names, and use a lightweight type system for Pandas API methods to effectively guide synthesis.

A *data frame* is a collection of data organized into rows and columns, similarly to a database table. Data frame manipulation is a key task in data wrangling, a preliminary step for data science or scientific computing tasks. For example, Figure 1 shows a data frame manipulation synthesis task taken from the AUTOPANDAS benchmark suite [Bavishi et al. 2019]. The goal is to use the Python Pandas library [Reback et al. 2022] to produce the data frame in Figure 1d, given the two input data

```
class ColNames < AbstractDomain
  attr_reader :cols
  def initialize(cols)
    @cols = cols.to_set
  end

  def ⊆(rhs)
    rhs.cols.subset?(@cols)
  end

  def ∪(rhs)
    ColNames.new(@cols ∪ rhs.cols)
end end
```

(a) ColNames Domain

```
class ColNameInterp < AbsInterp
  def self.interpret(env, prog)
    # details omitted for brevity
  end

  def self.pd_merge(left,right,opt)
    left ∪ right
  end

  def self.pd_query(df, pred)
    df
  end
end
```

(b) ColNames Abstract Semantics

```
class PyType < AbstractDomain
  attr_reader :ty

  def initialize(ty)
    @ty = ty
  end

  def ⊆(rhs)
    @ty ≤ rhs.ty
  end
end
```

(c) PyType Domain

```
class PyTypeInterp < AbsInterp
  def self.pd_merge(left,right,opt)
    DataFrame if left ⊆ DataFrame ∧
         right ⊆ DataFrame ∧
         opt ⊆ {on: Array<String>}
  end

  def self.pd_query(df, pred)
    DataFrame if df ⊆ DataFrame ∧
         pred ⊆ String
end end
```

(d) PyType Abstract Semantics

Fig. 2. Abstract domain definition for column names domain (a) and types domain definition (c). Abstract semantics for the required methods are defined in (b) using ColNames domain and (d) using PyType domain.

frames in Figure 1a and 1b and a query string (Figure 1e). In this case, the output joins the input rows with the same id but with different values in valueA and valueB columns. The Pandas library provides a wide range of methods that perform complex data frame manipulation. For example, calling `left.merge(right, on: ['col'])` joins the data frames `left` and `right` on column `col`. As another example, calling `df.query(str)` returns a new data frame with the rows of `df` that satisfy query string predicate `str` (as in Figure 1e).

To keep the synthesis task tractable, Absynthe restricts its search to Python code consisting of input variables arg0 through arg2; constants such as column names `'id'`, `'valueA'`, and `'valueB'` or row labels 0, 1, . . ., 13 from the data frames; array literals and indexing; and dictionaries (for keyword arguments). Additionally, for this discussion we will limit Absynthe to the merge and query methods just mentioned, even though our evaluation (§ 5.2) supports many more methods. Nonetheless, even with this restricted search space, naïve enumeration of possible solutions times out after 20 minutes. In contrast, using Absynthe, we can guide the search using abstract interpretation to find a solution in 0.47 seconds.

Abstract Domains and Semantics. The first step in using Absynthe is to identify appropriate abstract domains for the abstract interpretation and implement the abstract semantics. Typically, we develop the domain by looking at the input/output examples and thinking about the problem domain. In our running example, we observe that the data frames use columns id, valueA, and valueB, but each frame has a slightly different set of columns. This gives us the idea of introducing a domain ColNames that abstracts data frames to a set of column labels.

Abstract values, drawn from an abstract domain, represent a set of concrete values in the program. The abstract semantics define the evaluation rules of the program under values from this abstract domain. This approach has seen considerable success in practical static analysis tools such as ASTREÉ [Cousot et al. 2005] and Sparrow [Oh et al. 2012]. Figure 2a shows, similar to these tools, the definition of the ColNames domain, which is a class whose instances are domain values. Absynthe is implemented in Ruby, and Absynthe domains subclass AbstractDomain, which provides foundational definitions such as $\top$ and $\bot$ (see § 4). A value in the ColNames domain stores the set of columns it represents in the instance variable @cols, which by line 2 can be read with an accessor method cols. All abstract domains *require* a partial ordering relation $\subseteq$ on the domain that returns true if and only if the first columns label set (rhs.cols) is a subset of the second set (@cols). Finally, the $\cup$ method returns a new abstract value containing the union of the column names of the two arguments. The $\cup$ method is optional, however we define this as it will be used in the abstract semantics.

After defining the abstract domain, next we need to define the abstract interpreter to give semantics to the target language in our abstract domain. Figure 2b defines the abstract interpreter for ColNames domain as ColNameInterp class. All abstract interpreters are defined as a subclass of AbsInterp class, provided by Absynthe. It needs a definition of the interpret class method (the preceding `self.` denotes it is a class method), that given an environment env, and a term prog reduces it to a value of type ColNames. The interpret is a standard recursive interpreter, so we omit the definition of interpret for brevity. Then we define the pd_merge and pd_query class methods that define the operations for the Pandas merge and query methods on values from ColNames domain. A call to left.merge(right, opt) in the source term under abstract interpretation is computed via a call pd_merge(abs(left), abs(right), abs(opt)), where abs() indicates the abstract values of the arguments. In the column name abstraction, we only need to compute the column names of the resulting data frame, which is just the union of the column names of the input data frame (line 7). Notice the opt argument can be ignored, as it impacts how the data frames are merged in the concrete domain, but the set column names of the final data frame is unaffected. Similarly, a call to df.query(pred) is abstractly evaluated via a call to pd_query(abs(df), abs(pred)). Since the data frame returned by query has the same columns as its input data frame, pd_query simple returns the abstract data frame df (line 11).

Absynthe can also combine multiple domains together pointwise. We observe that the Pandas API methods expect values of a specific type. Hence, we also introduce a PyType abstract domain as a lightweight type system for Python. Figure 2c defines the abstract domain, which stores a type in the @ty field as a type from RDL [Foster et al. 2020], a Ruby type system. We build on RDL for representing Python types because it comes with built-in representations for nominal types, generic types, etc. and a subtyping relation between them. The $\subseteq$ method for PyType simply calls the subtyping method $\leq$ of RDL types. The subtyping method $\leq$ is a special-case of the partial ordering relation $\subseteq$.

Figure 2d defines gives the abstract semantics for merge and query in the PyType domain. The method pd_merge checks that the types of left and right are subtypes of DataFrame, *i.e.,* the type that represents Pandas data frames as shown in Figure 1, and that opt is a dictionary with a key on that admits an array of strings. If this check is satisfied, the return type is DataFrame. Otherwise,

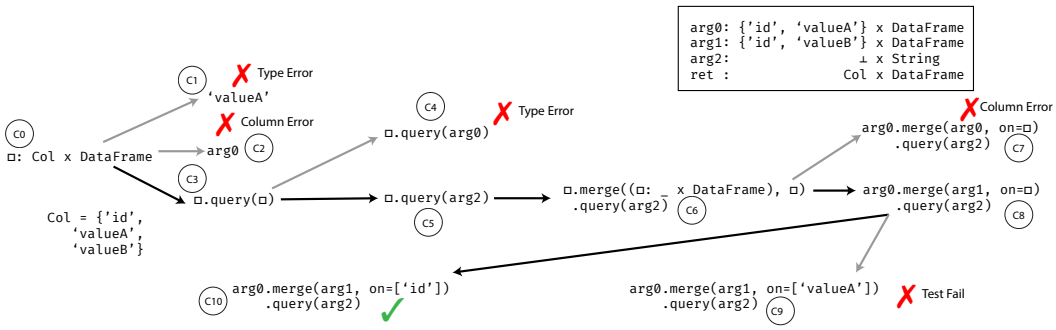

Fig. 3. Steps in the synthesis of solution to the problem in Figure 1. Some choices available to the synthesis algorithm has been omitted for simplicity.

pd_merge returns `nil`, which Absynthe interprets as $\top$, *i.e.,* any value is possible. Note, in a type checker, if the arguments do not match the expected types a type error occurs. Here, in contrast, we are computing what would be a valid abstraction, and since we don't have a specific type we can assume $\top$, *i.e.,* anything can happen. Later, during synthesis the search procedure will appropriately do the pruning by *type-checking* when it is provided a user specification. pd_query also checks if the receiver is a subtype of DataFrame and the query string is a String. If so, it returns DataFrame, otherwise it returns `nil`.

These domains are combined together using a ProductDomain class, provided by Absynthe. Here we write $\times$ to pair elements from the ColNames domain and the PyType domain. For example, {'id', 'valueA'} $\times$ DataFrame denotes all data frames that have the columns 'id' and 'valueA'. The ProductDomain also comes with a ProductInterp that evaluates product domain values with respective individual semantics and combines these into a final product abstract value.

Synthesizing Solutions. An Absynthe synthesis problem is specified by giving input/output examples for the synthesized function. Synthesis begins by abstractly interpreting the input/output examples to compute an *abstract signature* for the function. We have automated this for the AutoPandas benchmark suite. The upper-right corner of Figure 3 gives the abstract signature for out example. In particular, the first argument is a DataFrame with columns 'id' and 'valueA'; the second argument is a DataFrame with columns 'id' and 'valueB'; and the third argument is a String and has no columns. The synthesized function should return a DataFrame that has columns 'id', 'valueA', and 'valueB'. Additionally, Absynthe also uses a set of constants that can be used during the synthesis process. It constructs this from the rows and columns of the dataframes in the input/output example: {'id', 'valueA', 'valueB', 0, 1, . . . , 13}.

Absynthe iteratively produces candidate function bodies that may contain *holes* $\square : a$, where each hole is labeled with the abstract value a its solution must abstractly evaluate to. Synthesis begins (left side of figure) with candidate C0, which is a hole labeled with the abstract return value of the function. At each step, Absynthe replaces a hole with an expression that satisfies its labels. For example, candidate C1 is not actually generated because its concrete value 'valueA' is not of type DataFrame. The process continues until the program has been full concretized, at which point is it tested in the Python interpreter against the input/output examples. Synthesis terminates when it finds a candidate that matches the input/output examples. For our running example, Figure 1c shows the solution synthesized by Absynthe.

The rest of the figure illustrates the search process. The candidate C2 does not satify the abstract specification on columns, so it is also never generated. The candidate C3 instead expands the hole to

a call to query, which itself has holes for the receiver and argument. Note we omit the abstraction labels here because Absynthe has not fixed the abstract value for that hole yet. Absynthe treats these as abstract variables that can be used during abstract interpretation, but will be eventually substituted with a fixed abstract value as the search proceeds (discussed in § 3).

After a single set of expansion of holes, Absynthe runs the abstract interpreter on all candidates (including the partial programs). Running C3 through the abstract interpreter calls the pd_query function from PyTypeInterp (Figure 2d). From the evaluation of the pd_query Absynthe can infer the first hole has to be a subtype of DataFrame and the second hole should be a subtype of String. Thus candidates like C4 will not be generated as it is ill-typed (arg0 is a DataFrame).

We use filling the remaining hole in C5 to illustrate another feature of Absynthe, enumerating finite abstract domains. Absynthe has the upper bound of this hole at DataFrame, it will substitute all possible values from PyType that are subtypes of DataFrame. Since there is only one type *i.e.,* DataFrame, it synthesizes expressions of that type at the hole. For the next candidate C6, again by running abstract interpreter bounds for □ are determined. The _ in the □ signifies that ColNames domain still is an abstract variable, while the types have been concretized. Absynthe can determine bounds for variables only if the abstract transformers have conditionals (discussed in § 3.1), not present in pd_merge of ColNameInterp. Running the abstract interpreter eliminates candidate C7 as the partial program will not satisfy the synthesis goal. Eliminating partial programs removes a family of concrete programs, narrowing the search space further. Absynthe next generates candidates C8 and C9. C8, however, is eliminated because the ColNames domain interpreter computes the final data frame will have columns {'id', 'valueA'}. Eventually, the keyword argument to the merge method is filled with an array. Some ways of filling that argument fail the test cases (C10), but C11 passes all tests and is accepted as the solution (after being wrapped in a Python method defintion), also shown in Figure 1c.

3 FORMALISM

In this section we formalize Absynthe in a core language $\mathcal{L}_f$. Figure 4a shows the $\mathcal{L}_f$ syntax. Expressions in $\mathcal{L}_f$ have values v, drawn from a set of *concrete values* $\mathbb{V}$; variables x; holes $\square : a$ tagged with an abstraction a; and function application $f(e, \ldots, e)$. Note that these are external functions f, e.g., to call out to libraries. Programs in $\mathcal{L}_f$ consist of a single function definition $\textbf{def}\ m(x) = e$ of a function m that takes an argument x and returns the result of evaluating e.

Abstractions a include abstract values $\tilde{v}$ drawn from an abstract domain $\mathbb{A}$. We assume this domain forms a complete lattice with greatest element $\top$, least element $\bot$, and partial ordering $a_1 \sqsubseteq a_2$. Abstractions also include *abstract variables* $\tilde{x}$, which Absynthe uses to label holes whose abstractions cannot immediately be determined. For example, if Absynthe synthesizes an application of a function f, it labels f's arguments with abstract variables. During synthesis, Absynthe maintains bounds on such variables to narrow down the search space (see below). We refer to abstract values from § 2 as *abstractions* in this section to avoid the ambiguity between abstract variables and values. Concrete values are lifted to abstract values using the abstraction function α, mapping concrete values to abstract values, *i.e., α* maps $\mathbb{V}$ to $\mathbb{A}$. Likewise, abstract values map to a set of concrete values using the concretization function γ, *i.e., γ* maps $\mathbb{A}$ to the $\wp(\mathbb{V})$. We write $v \in \tilde{v}$ as a shorthand for checking that v is in the concretization of $\tilde{v}$. We assume that for each function f, we have a corresponding abstract transfer function $f^{\#}$ that soundly captures its semantics. Finally, during synthesis, Absynthe maintains two variable environments: Γ, binding variables x to their abstractions, and Δ, binding abstract variables $\tilde{x}$ to their bounds. Abstract variable bounds are written as a tuple of the lower and upper bound respectively (details in § 3.1).

$$
\begin{array}{lrcl}
\textit{Expressions} & e & ::= & v \mid x \mid \square : a \mid f(e, \ldots, e) \\
\textit{Programs} & P & ::= & \textbf{def}\ m(x) = e \\
\textit{Concrete Values} & v & \in & \mathbb{V} \\
\textit{Abstractions} & a & ::= & \tilde{v} \mid \tilde{x} \\
\textit{Abstract Values} & \tilde{v} & \in & \mathbb{A} \\
\\
\textit{Abstraction Function} & \alpha & : & \mathbb{V} \to \mathbb{A} \\
\textit{Concretization Function} & \gamma & : & \mathbb{A} \to \wp(\mathbb{V}) \\
\textit{Inclusion} & v \in \tilde{v} & \text{if} & v \in \gamma(\tilde{v}) \\
\textit{Abstract Transfer Function} & f^{\#} & : & (\mathbb{A}, \ldots, \mathbb{A}) \to \mathbb{A} \\
\\
\textit{Abstract Environment} & \Gamma & ::= & \emptyset \mid x : a,\ \Gamma \\
\textit{Bounds Environment} & \Delta & ::= & \emptyset \mid \tilde{x} : (a, a),\ \Delta
\end{array}
$$

(a) Syntax and relations of $\mathcal{L}_f$.

$$\boxed{\Gamma \vdash e \Downarrow a}$$

$$\frac{\eta(v) = a}{\Gamma \vdash v \Downarrow a}\ \textsc{E-Val} \qquad \frac{}{\Gamma \vdash x \Downarrow \Gamma[x]}\ \textsc{E-Var} \qquad \frac{}{\Gamma \vdash \square : a \Downarrow a}\ \textsc{E-Hole}$$

$$\frac{\Gamma \vdash e_1 \Downarrow a_1 \quad \ldots \quad \Gamma \vdash e_n \Downarrow a_n}{\Gamma \vdash f(e_1, \ldots, e_n) \Downarrow f^{\#}(a_1, \ldots, a_n)}\ \textsc{E-Fun}$$

(b) Abstract semantics for $\mathcal{L}_f$.

Fig. 4. Syntax, relations, and abstract semantics of $\mathcal{L}_f$.

Abstract Semantics. Next, we define semantics to abstractly interpret candidate programs in our domain. Figure 4b presents the relation $\Gamma \vdash e \Downarrow a$ that, given an abstract environment Γ, evaluates an expression e to an abstract value a. E-Val lifts a concrete value to the abstract domain by applying the abstraction function. E-Var lifts a variable to an abstract value by substituting the value from the environment Γ. E-Hole abstractly evaluates a hole to its label. Finally, E-Fun recursively evaluations a function application's arguments and then applies the abstract transfer function $f^{\#}$.

Synthesis Problem. We can now formally specify the synthesis problem: Given an abstract domain $\mathbb{A}$, a set of abstract transformers $f^{\#}$, and an abstract specification of the function's input and output $a_1 \to a_2$, synthesize a set of programs P such that NoHole(P), i.e., P has no holes in it, and $x : a_1, \emptyset \vdash P \Downarrow a_2$, i.e., P abstractly evaluates to a_2 given that x has abstract value a_1. Then, the final solution is chosen as a synthesized candidate P that passes all input/output examples.

3.1 Abstract Transformer Function DSL

Figure 5a shows $\mathcal{L}_{meta}$, the DSL to define abstract transformer functions $f^{\#}$ for Absynthe. The primary purpose of the DSL is to let users define $f^{\#}$ that can handle both abstract values a and variables $\tilde{x}$ correctly. It is expressive enough to write the abstract transformer function for domains in § 2. Expressions $\hat{e}$ in $\mathcal{L}_{meta}$ can be either such abstractions a, variables y, function application $g(\hat{e})$ and if-then-else statements. We consider g as uninterpreted abstract functions. The conditionals b for if statements include **top?** that tests if an expression is $\top$, **bot?** that tests if an expression is $\bot$,

$$
\begin{array}{llll}
\textit{Expressions} & \hat{e} & ::= & a \mid y \mid g(\hat{e}) \mid \textbf{if } b \textbf{ then } \hat{e} \textbf{ else } \hat{e} \\
 & & \mid & \textbf{if } \hat{e} \subseteq \hat{e} \textbf{ then } \hat{e} \textbf{ else } \top \mid \textbf{if } g(\hat{e}) \textbf{ then } \hat{e} \textbf{ else } \top \\
\textit{Conditionals} & b & ::= & \textbf{top?}\ \hat{e} \mid \textbf{bot?}\ \hat{e} \mid \textbf{var?}\ \hat{e} \mid \textbf{val?}\ \hat{e} \\
\textit{Transfer Functions} & \hat{P} & ::= & \textbf{def } f^{\#}(y) = \hat{e}
\end{array}
$$

(a) Syntax of $\mathcal{L}_{meta}$.

$$\boxed{\Gamma \vdash \langle \Delta, \hat{e} \rangle \Downarrow \langle \Delta, a \rangle}$$

$$\frac{\Gamma[y] = a}{\Gamma \vdash \langle \Delta, y \rangle \Downarrow \langle \Delta, a \rangle}\ \text{A-Var} \qquad \frac{\Gamma \vdash \langle \Delta_1, \hat{e} \rangle \Downarrow \langle \Delta_2, a \rangle}{\Gamma \vdash \langle \Delta_1, g(\hat{e}) \rangle \Downarrow \langle \Delta_2, g(a) \rangle}\ \text{A-Func}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, b \rangle \Downarrow \langle \Delta_2, \texttt{true} \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_1 \rangle \Downarrow \langle \Delta_3, v \rangle\end{array}}{\Gamma \vdash \langle \Delta_1, \textbf{if } b \textbf{ then } \hat{e}_1 \textbf{ else } \hat{e}_2 \rangle \Downarrow \langle \Delta_3, v \rangle}\ \text{A-IfT} \qquad \frac{\Gamma \vdash \langle \Delta, \hat{e} \rangle \Downarrow \langle \Delta, \top \rangle}{\Gamma \vdash \langle \Delta, \textbf{top?} \rangle \Downarrow \langle \Delta, \texttt{true} \rangle}\ \text{A-TopT}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, \hat{e}_1 \rangle \Downarrow \langle \Delta_2, \tilde{x} \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_2 \rangle \Downarrow \langle \Delta_3, \tilde{v} \rangle \\ \Delta_3[\tilde{x}] = (a_2, a_3) \quad a_2 \subseteq \tilde{v} \subseteq a_3 \\ \Delta_4 = \Delta_3[\tilde{x} \mapsto (a_2, \tilde{v})]\end{array}}{\Gamma \vdash \langle \Delta_1, \hat{e}_1 \subseteq \hat{e}_2 \rangle \Downarrow \langle \Delta_4, \texttt{true} \rangle}\ \text{A-VC} \qquad \frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, \hat{e}_1 \rangle \Downarrow \langle \Delta_2, \tilde{x}_1 \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_2 \rangle \Downarrow \langle \Delta_3, \tilde{x}_2 \rangle\end{array}}{\Gamma \vdash \langle \Delta_1, \hat{e}_1 \subseteq \hat{e}_2 \rangle \Downarrow \langle T(\Delta_3, \tilde{x}_1, \tilde{x}_2), \texttt{true} \rangle}\ \text{A-VS}$$

$$T(\Delta, \tilde{x}_1, \tilde{x}_2) = \begin{cases} \Delta[\tilde{x}_1 \mapsto (a_3, a_4)] & \textit{if } a_1 \subseteq a_3, a_4 \subseteq a_2 \\ \Delta[\tilde{x}_1 \mapsto (a_3, a_2), \tilde{x}_2 \mapsto (a_3, a_2)] & \textit{if } a_3 \subseteq a_2 \subseteq a_4 \\ \Delta & \textit{if } a_1 \not\subseteq a_4 \\ \quad \textit{where } \Delta[\tilde{x}_1] = (a_1, a_2), \Delta[\tilde{x}_2] = (a_3, a_4) & \end{cases}$$

(b) Selected $\mathcal{L}_{meta}$ evaluation rules.

Fig. 5. Syntax and evaluation rules of $\mathcal{L}_{meta}$.

var? that tests if an expression is an abstract variable $\tilde{x}$, and **val?** that tests if an expression is a abstract value a. Additionally, expressions $\hat{e}$ can test for ordering using $\subseteq$ or can call an abstract function $g(\hat{e})$. The else branch of these conditionals evaluate to $\top$, *i.e.,* it evaluates to the largest possible abstraction $\top$ if a test of ordering fails. This is done to soundly over-approximate program behavior, while sacrificing precision. The abstract transformer is defined as a function $f^{\#}$ that takes the input abstract value as argument y and computes the output abstraction by evaluating the expression $\hat{e}$.

Figure 5b shows selected big-step evaluation rules for the abstract transformer functions written in $\mathcal{L}_{meta}$. Under an abstract environment Γ and a bounds environment Δ, expression $\hat{e}$ evaluates to a new bounds environment and a value v. In general these rules reflect standard big step semantics, except for the $\subseteq$ operation, where the bounds get constrained because of the comparison. The rule A-IfT evaluates the branch condition b and evaluates $\hat{e}_1$ if it is `true`. A similar rule (omitted here) can be written if the conditional evaluates to `false`. A-TopT checks if the expression $\hat{e}$ evaluates to $\top$. We omit evaluation rules for the `false` case and other branching predicates such as **bot?**, **var?**, and **val?** which are similar to **top?**.

The rules for evaluating $e \subseteq e$ are most interesting, as these test for the $\subseteq$ relation while constraining abstract variables $\tilde{x}$ to the range under which the relation $e \subseteq e$ holds. In general, the

$$\boxed{\Delta, \Gamma \vdash e \rightsquigarrow e : a}$$

$$\frac{v \in \gamma(a') \quad a' \subseteq a}{\Delta, \Gamma \vdash \square : a \rightsquigarrow v : \alpha(v)}\ \textsc{S-Val} \qquad \frac{\Gamma[x] = a' \quad a' \subseteq a}{\Delta, \Gamma \vdash \square : a \rightsquigarrow x : a'}\ \textsc{S-Var}$$

$$\frac{\begin{array}{c} f^{\#}(\tilde{x}_1, \ldots, \tilde{x}_n) = a' \quad \Delta[\tilde{x}_i] = (a_{i,1}, a_{i,2}) \\ a_{i,1} \subseteq a_i \quad a_i \subseteq a_{i,2} \quad a' \subseteq a \end{array}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(\square : a_1, \ldots, \square : a_n) : a'}\ \textsc{S-Finite}$$

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash \square : \tilde{x}_1 \rightsquigarrow e_1 : \tilde{x}_1 \quad \ldots \quad \Delta, \Gamma \vdash \square : \tilde{x}_{n-1} \rightsquigarrow e_{n-1} : \tilde{x}_{n-1} \\ \Gamma \vdash e_1 \Downarrow a_1 \quad \ldots \quad \Gamma \vdash e_{n-1} \Downarrow a_{n-1} \\ f^{\#}(a_1, \ldots, a_n) = a' \quad a' \subseteq a \end{array}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(e_1 : a_1, \ldots, \square_n : a_n) : a'}\ \textsc{S-Solve}$$

$$\frac{\tilde{x}_i \text{ and } \tilde{x}' \text{ is fresh}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(\square : \tilde{x}_1, \ldots, \square : \tilde{x}_n) : \tilde{x}'}\ \textsc{S-Enumer}$$

Fig. 6. Hole replacement rules for $\mathcal{L}_f$.

abstract variable narrowing reduces the range of $\tilde{x}$ to a sound range for that evaluation through $f^{\#}$. In effect it is finding satisfiable range for $\tilde{x}$ for that branch. A-VarConst tests for the $\subseteq$ relation when $\hat{e}_1$ evaluates to a variable $\tilde{x}$ and $\hat{e}_2$ evaluates to a values $\tilde{v}$. In such a case, if $\tilde{v}$ is within the range of the variable $\tilde{x}$ the term evaluates to `true`, while updating the upper bound of $\tilde{x}$ to $\tilde{v}$. This narrows the abstract variables, while still being sound under which the partial order relation $\subseteq$ holds true. A similar symmetrical rule exists (omitted here) where the left hand evaluates to $\tilde{v}$ and right hand evaluates to $\tilde{x}$. Finally, A-VarSub gives the rules for comparing two abstract variables $\tilde{x}_1$ and $\tilde{x}_2$. It uses a metafunction T to describe the cases where $\tilde{x}_1$ is contained in $\tilde{x}_2$, or has some overlap, or $\tilde{x}_1$ is less than $\tilde{x}_2$.

3.2 Abstraction-Guided Synthesis

To perform abstraction-guided synthesis, Absynthe recursively replaces holes by suitable expressions and then tests fully concretized candidates. Figure 6 shows the rules for hole replacement. These rules prove judgments of the form $\Delta, \Gamma \vdash e_1 \rightsquigarrow e_2 : a$, meaning in bounds environment Δ and abstract environment Γ, expression e_1 takes a step by replacing a hole in e_1 to yield a new expression e_2. In particular, S-Val replaces $\square : a$ with a value v from the concrete set that a abstracts. Similarly, S-Var replaces a hole with a variable that is compatible with the hole's label.

The next few rules are used to generate function applications, or more generally, any term that may have more holes. First, S-Finite generates function application when the domain from which a is drawn is finite, *e.g.,* a simple type system that without polymorphic types or first class lambdas, or an effect system as used in Guria et al. [2021]. This rule can produce multiple candidates with each hole tagged with distinct abstract values from the domain. Second, for abstract domains with infinite values that can be represented in a background theory solver, Absynthe applies the S-Solve. If the function application requires n arguments, only $n-1$ arguments are concretized to a term. This gives the constraint $f^{\#}(a_1, \ldots, a_n) = a'$ with only one unknown, a_n, that can solved for and assigned to the hole. For $f^{\#}$ to be lifted to a SMT solver $f^{\#}$ should also have an interpretation a background theory supported by the solver. This is useful for representing predicate abstractions or numeric domains such as intervals or string lengths (used in SyGuS

Algorithm 1 Synthesis of programs that passes a spec s

```
1:  procedure Generate(a1 → a2, maxSize)
2:      Γ ← [x ↦ a1]
3:      e0 ← □ : a2
4:      workList ← [e0]
5:      while workList is not empty do
6:          ecurr ← pop(workList)
7:          ωenumer ← {et | Γ ⊢ ecurr ⇝ et : a}
8:          ωvalid ← {et ∈ ωenumer | Γ ⊢ et ⇓ a ∧ a ⊆ a2}
9:          ωeval ← {et ∈ ωvalid | NoHole(et)}
10:         ωrem ← ωvalid − ωeval
11:         for all et ∈ ωeval do
12:             return et if TestProgram(et)
13:         end for
14:         ωrem ← {et ∈ ωrem | size(et) ≤ maxSize}
15:         workList ← reorder(workList + ωr)
16:     end while
17:     return Error: No solution found
18: end procedure
```

evaluation in § 5.1). Finally, S-Enumer replaces a hole with a function application with fresh abstract variables $\tilde{x}_i$ for the arguments and return. Notice there is no guarantee f will produce a value of the appropriate abstraction. This is because, while we assume we have an abstract transfer function $f^{\#}$, we do not know what abstraction it will compute without concretizing the arguments. However, unsound partial programs will be eliminated by the abstract interpreter as discussed below. Given only forward evaluation semantics and no other information about the domains, this is best way to construct partial program candidates. Absynthe can switch between bottom-up synthesis (S-Enumer) and top-down goal-directed synthesis (rest of the S- rules) depending on which rule is applied. While these rules are non-deterministic, the Absynthe implementation (§ 4) chooses and applies these rules for the correct domain in a fixed order to yield solutions.

Synthesis Algorithm. Algorithm 1 performs abstraction-guided synthesis. The algorithm uses a work list and combines synthesis rules for candidate generation with search space pruning based on abstract interpretation, in addition to testing in a concrete interpreter. The ordering of programs in the worklist determines the order in which program candidates are explored (discussed in § 4). The synthesis algorithm starts off with an empty candidate e_0 as a base expression in the work list. At every iteration it pops one item from the work list and applies synthesis rules (Figure 6) in a non-deterministic order to produce multiple candidates ω_{enumer}. Each candidate is abstractly interpreted, and then checked to see if the computed abstraction satisfies the goal abstraction. If it is satisfied it is added to the set of valid candidates ω_{valid} (line 8). As partial programs with holes represent a class of programs, abstractly interpreting these eliminate a class of programs if they are not included in the goal a_2. Thus, the algorithm iterates through partial programs which are *sound* with respect to the abstract specification. Any unsound programs generated by S-Enumer are pruned here.

Finally, all concrete programs ω_{eval} are tested in the interpreter to check if a program satisfies all test cases, in which case it is returned as the solution. The remaining programs ω_{rem} contain holes, so these can be expanded further by the application of synthesis rules. Only programs below

the maximum size of the search space are put back into the work list, and the order of the work list is always based on some domain-specific heuristics (§ 4 discusses our program ordering).

4 IMPLEMENTATION

Absynthe is implemented in approximately 3000 lines of Ruby excluding dependencies. It is architected as a core library whose interfaces are used to build a synthesis tool for a problem domain. Additionally, to support solver-backed domains, we developed a library (~460 lines) to lazily convert symbolic expressions to Z3 constraints and solve those in an external process. Absynthe uses a term enumerator that, at each step, visits holes in a term and substitutes it with values or subterms containing more holes applying the rules shown in § 3.2. Absynthe requires users to define a translation from the ASTs to the source program and a method that tests a candidate to return if the test passed or not. Users may provide a set of constants for the language which are used as values to be used in the concretization function. In practice, this is useful when the language has infinite set of terminals (like Python), and selecting values from the set of constants makes the term generation tractable. For AutoPandas benchmarks, we infer such constants from the data frame row and column labels (§ 2).

Absynthe explores program candidates in order of their size, preferring smaller programs first (line 15 of Algorithm 1). We plan to explore other program exploration order in future work. The synthesis rules presented in § 3.2 are non-deterministic, however, our implementation fixes an order of application such rules. It prefers to synthesize constants and variables followed by function applications, hashes, arrays, etc. Moreover, based on the definition of abstract domains (discussed below), it can automatically choose to apply the S-Finite or S-Solver rules. If none of these specialized rules apply, it uses S-Enumer rule to synthesize subterms.

Abstract Domains. To guide the search, users need to implement an abstract domain. Absynthe provides a base class—AbstractDomain from which a programmer can inherit their own abstract domains implementation, like Figure 2. The base classes come with machinery that gives built-in implementation of $\top$, $\bot$, abstract variables $\tilde{x}$, and supporting code for partial ordering between these abstract values. The user has to define how to construct abstract values for that domain (the `initialize` method in § 2), the partial ordering relation $\subseteq$ between two abstract *values*. The abstract variable narrowing (§ 3.1) is implemented as the $\subseteq$ method in the AbstractDomain base class. Solver-aided domains (such as string length in § 5.1) construct solver terms when initializing an abstract value, or apply functions that compute abstract values (including $\cup$ and $\cap$). These terms are checked for satisfiability of $a_1 \subseteq a_2$ in the solver when the $\subseteq$ method is invoked, and any solved abstract variables are assigned to its holes. If the solver proves the solver term unsatisfiable, the candidate is eliminated. The rule S-Finite is applied for domains with finite abstract values and S-Solve is used for domains whose values can be inferred using an SMT solver yielding top-down goal-directed synthesis. In case these cannot be applied, Absynthe falls back to using the S-Enumer rule that is equivalent to bottom-up term enumeration. We plan to explore a more ergonomic API for the Absynthe framework in future work.

Absynthe also provides a ProductDomain class to automatically derive product domains by combining any user-defined domains as needed. The $\subseteq$ method on ProductDomain returns the conjunction of respective $\subseteq$ on the individual domains it is composed of.

Abstract Interpreters. Each abstract domain needs a definition of abstract semantics, inherited from the `AbstractInterp` class provided by Absynthe (as shown in § 2). All subclasses override the `interpret` method that takes as argument the abstract environment and the AST of the term that is being evaluated. In practice, it is implemented as evaluating subterms recursively, and then applying the abstract transformer function written in a subset of Ruby (similar to $\mathcal{L}_{meta}$ in § 3.1) to

evaluate the program in the abstract domain. A sound interpreter for ProductDomain is derived automatically, by composing the interpreters of its base domains. More specifically, it evaluates the term under individual base domains and then combines the results pointwise into a product.

Concrete Tests. Any synthesized term without holes that satisfies the abstract specification is tested by Absynthe in a reference interpreter against concrete test cases. Absynthe expects the programmer to define a test_prog method that calls the reference interpreter with the synthesized source program (as a string in the source language), and returns a boolean to indicate if the tests passed. The reference interpreter runs the test case, which in many cases boils down to checking the program against the provided input/output examples. If the program passes all test cases, it is considered the correct solution. If the program fails a test, it is discarded.

Optimizations. In practice, Absynthe uses a min-heap to store a work list of candidates ordered by their size. This eliminates the reorder step (Algorithm 1 line 15), saving an average cost of $O(n \log n)$ at each synthesis loop iteration. Additionally, we found certain common subterms occur frequently in the same program, *e.g.*, computing the index of the first space in a string in a SyGuS program. Absynthe caches small terms (containing up to one function application) that do not have any holes to save the cost of synthesizing these small fragments. Whenever, a hole with compatible abstract value is found, these fragments are substituted directly without doing the repetitive work of synthesizing the function application from scratch again (similar to subterm reuse in DryadSynth [Huang et al. 2020]). Finally, Absynthe tests a set of predicates against given input/output examples, to guess a partial program instead of starting from just a □ term. For example, Absynthe has a predicate that checks if the output is contained in the input, then the output is a substring of the input. For the SyGuS language, if the predicate (str.contains output input) tests true, then the partial program is inferred to be (str.substr input □ □). This reduces the problem complexity by cutting down the search space. Another predicate (str.suffixof output input) tests if the input ends with output, then it infers the partial program (str.substr input □ (str.len input)), *i.e.,* the program is possibly a substring of the input from some index to the end. We evaluate the performance impact of the latter two optimizations in § 5 (No Template column in Table 1).

Limitations. While Absynthe is a versatile tool to define custom abstract domains and combine it with testing in a reference interpreter, the approach does have some limitations. First, Absynthe only works with forward evaluation rules over the abstract domain, in contrast to FlashMeta [Polozov and Gulwani 2015] that requires "inverse semantics", *i.e.,* rules that given a target abstraction computes the arguments to the abstract transformer. While specifying only the forward semantics eases the specification burden for users, it require more compute time to synthesize subterms such as arguments to functions. Second, while we found product domain useful to combine separate domains, these domains remain independent through synthesis, unlike predicates where all defined semantics can be considered at the same time. We plan to explore methods to make product domains more expressive in future work. Third, problems where one can define full formal semantics are a better fit for solver-aided synthesis tools such as Rosette [Torlak and Bodík 2014] or SemGuS [Kim et al. 2021]. We share performance benchmarks on SyGuS strings (which have good solver-aided tools) to give some evidence for this in our evaluation (§ 5.1). Notably, solver-aided tools can jointly reason about subterms. In contrast, when using solver-aided domains, Absynthe concretizes some of the subterms which requires enumeration through larger number of terms. Finally, Absynthe falls back on term enumeration when abstract domains do not provide any more guidance, often leading to combinatorial explosion for larger terms.

5 EVALUATION

We evaluate Absynthe by targeting it in variety of domains, to verify it can synthesize different workloads. The primary motivation is to evaluate the general applicability of abstract interpretation-guided synthesis to diverse problems rather than being a state-of-the-art tool at a single synthesis benchmark suite. The questions we aim to answer in our evaluation are:

- How well does Absynthe work for problems traditionally targeted using solver-based strategies using the SyGuS strings benchmark [Alur et al. 2017a] (§ 5.1)? We also discuss the performance impact of optimizations and program exploration behavior in Absynthe.
- Can Absynthe be adapted to an unrelated problem (not handled by any tools that solve SyGuS benchmarks) where it is difficult to write precise formal semantics? We test this by using Absynthe to synthesize Python programs that use the Pandas library from the AutoPandas [Bavishi et al. 2019] benchmark suite (§ 5.2).

5.1 SyGuS Strings

Benchmarks. To test that Absynthe is a viable approach to synthesize programs that has been well explored in prior work, we target it on the SyGuS strings benchmark suite [Alur et al. 2017a]. We believe strings form a good baseline to compare Absynthe with other synthesis approaches that rely on enumerative search [Alur et al. 2017b], SMT solvers [Reynolds and Tinelli 2017], and abstract methods directed by solvers [Wang et al. 2017b] (discussed in details in § 6). In contrast, Absynthe uses only abstract domains with their forward transformers to guide the search. We *do not expect* Absynthe to out-perform the past tools, rather to evaluate if it can solve most of the benchmarks at a lower cost of defining lightweight abstract domains and partial semantics upfront.

SyGuS strings has 22 benchmarks with 4 variants of each—standard (baseline set of input/output examples), small (fewer examples than standard), long (more examples than standard), long-repeat (more examples than long with repeated examples). As our approach is dependent only on the abstract specification and testing, not on the number of examples, we show detailed results for the standard version of these benchmarks. These results generalize to all variants of each benchmark. As we aim to evaluate how abstraction guided search performs, we exclude any programs containing branches. Previous work like RbSyn [Guria et al. 2021] and EuSolver [Alur et al. 2017b] have used test cases that cover different paths through a program to do more efficient synthesis of branching programs. These can be adapted to a system like Absynthe with minor effort.

Absynthe parses the SyGuS specification files directly to prepare the synthesis goal and load the target language. As SyGuS does not come with an official concrete interpreter for programs, we provide one written in Ruby that is compliant with the SyGuS specifications [Raghothaman and Udupa 2014]. Absynthe uses this interpreter as a black-box and does not receive any additional feedback other than the generated SyGuS programs satisfied the input-output examples or not.

Abstract Domains. We defined the following abstract domains and their semantics to run the benchmark suite:

(1) **String Length.** A solver-aided domain to lift strings to their lengths, while lifting integers and booleans without transformation. This means the concretization of the abstract value 5 can be the number 5 and the set of all strings of length 5, whereas the boolean abstract value `true` or `false` represents identical concrete values.
(2) **String Prefix.** A domain to represent the set of strings that begin with a common prefix. For example, an abstract value with string "fo" is wider than an abstract value with string "foo", as the former denotes all strings starting with "fo" and the latter includes a subset of that, *i.e.,*

strings starting with "foo". The $\subseteq$ operation checks if the prefix of one string starts with the prefix of the other.

(3) **String Suffix.** A domain to represent the set of strings that end with a common suffix, similar to string prefix domain. The $\subseteq$ operation checks if the suffix of one string ends with the suffix of the other.

These domains were created by looking at the input/output examples in the synthesis specs, and encoding the simplest partial semantics that guides the reasoning. For example, a few problems have programs that start with or end with a string constant. This is how we designed the string prefix and suffix domains respectively. On the other hand, many problems produce strings of fixed lengths or the length of the output string is a function of the length of the input string. The string string length domain expresses semantics constraints of this kind. As the string length domain is solver-aided, it can handle symbolic constraints from abstract variables like the string length of a substring `str.substr` operation is `j - i` where `i` and `j` are the start and end index respectively. Although the string length domain does not preserve type information, SyGuS being a typed language (type-soundness enforced by the grammar) all programs in the language are type-correct by construction. Consequently, we did not need to write a type system as an abstract domain.

Finally, we give abstract specifications in the selected abstract domains where required. Specifically, we run each benchmark without an abstract annotation, *i.e.,* equivalent to $\top \longrightarrow \top$ specification which results in naive enumeration combined with abstract interpretation. If a benchmark times out, then we add an abstract annotation, such as $\top \longrightarrow$ "Dr. " for the dr-name example (Table 1). This specification means, Absynthe should find a function that given any input string ($\top$), it computes strings starting with "Dr. " only.

Results. Table 1 shows the results of running the SyGuS strings benchmarks through Absynthe with the discussed domains. The numbers are reported as a median of 11 runs on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16 GB RAM. All experiments had a timeout of 600 seconds. In Table 1, *Benchmark* column is the name of the problem, *# Ex* shows the number of input/output examples. *Time* shows the median running time of the benchmark along with the semi-interquartile range over 11 runs. The *Size* and *Ht* columns give the size of the synthesized program as the count of the AST nodes in the SyGuS language and the height of the synthesized program AST respectively. The *# Tested* column lists the number of programs that were tested in the concrete interpreter before a solution was found. An abstraction that works well reduces this number compared to a worse abstraction or naïve enumeration. *Domains* column lists the domains used for synthesizing the program. These domains were provided as a specification in the abstract domain. $\top$ denotes that an abstract specification was provided as a product of $\top$ values in all individual domains for input and output, resulting in just term enumeration. The rows which mention the domain was provided abstract specs only from that domain, resulting in guidance from the provided specification. The *# Elim* lists the number of partial programs (denotes a family of concrete programs) that were eliminated by running the abstract interpreter with the provided specification during the search. For the problems which used the $\top$ domain, the abstract interpreter did not eliminate any partial programs, as specification admits all programs. Any row with – denotes time out of the benchmark under these abstract specifications.

Most benchmarks are solved within ~7 seconds, with exceptions being *name-combine-3*, *phone-6*, and *phone-7* which take longer. In general a larger program takes much longer to synthesize, due to combinatorial increase in the number of terms being searched through as the AST size increases. For example, larger programs with same AST height take longer to synthesize due to higher number of function arguments. The number of examples do not impact the time for synthesis as most time is spent in abstract interpretation and term generation. Testing a candidate on the examples take

Table 1. Results of running Absynthe on SyGuS strings benchmarks. *# Ex* lists the number of I/O examples; *Time* lists the median and semi-interquartile range for 11 runs; *Size* and *Ht* reports the number of AST nodes and the height of the program AST respectively; *# Tested* is the number of programs run in the concrete interpreter before a solution was found; *Domains* lists the domains used to specify the abstract spec; and *# Elim* lists the number of partial programs eliminated by the abstract interpreter during search. *No cache* and *No Temp* measure the performance of Absynthe when small expression cache and template inference (§ 4) are disabled respectively.

Benchmark	# Ex	Time (sec)	Size	Ht	# Tested	Domains	# Elim	No Cache	No Temp
bikes	6	1.70 ± 0.02	7	4	4808	⊤	0	2.55	35.05
dr-name	4	1.54 ± 0.02	11	4	4797	Prefix	46610	139.53	2.92
firstname	4	0.03 ± 0.00	7	3	4	⊤	0	0.63	0.18
initials	-	-	-	-	-	-	-	-	-
lastname	4	0.02 ± 0.00	10	4	15	⊤	0	0.81	18.72
name-combine	6	0.21 ± 0.00	5	3	566	⊤	0	0.24	0.22
name-combine-2	4	6.01 ± 0.06	9	4	9723	Suffix	48516	6.65	8.28
name-combine-3	6	47.86 ± 0.23	9	5	117370	Suffix	124573	68.29	43.63
name-combine-4	-	-	-	-	-	-	-	-	-
phone	6	0.03 ± 0.00	4	2	3	⊤	0	0.03	0.12
phone-1	6	0.16 ± 0.00	6	3	1189	⊤	0	0.20	7.32
phone-2	6	0.05 ± 0.01	7	3	41	⊤	0	0.04	63.82
phone-3	-	-	-	-	-	-	-	-	-
phone-4	6	0.05 ± 0.01	4	2	1577	⊤	0	0.05	0.14
phone-5	7	0.03 ± 0.00	7	3	18	⊤	0	2.16	0.20
phone-6	7	100.54 ± 0.51	14	4	5937	Length	12234	-	27.79
phone-7	7	103.92 ± 0.37	14	4	54051	Length	12639	-	-
phone-8	7	0.72 ± 0.00	10	4	217	Length	31	1.37	-
phone-9	-	-	-	-	-	-	-	-	-
phone-10	-	-	-	-	-	-	-	-	-
reverse-name	6	0.35 ± 0.00	5	3	593	⊤	0	0.41	0.42
univ-1	6	6.69 ± 0.07	7	3	19683	⊤	0	8.08	7.73

minimal time. Absynthe performs reasonably well, solving around the same number of benchmarks as EuSolver [Alur et al. 2017a]. We selected EuSolver as it is based on an enumerative search method like Absynthe. The timeout of 600 seconds only applies to our Absynthe evaluation, whereas EuSolver was evaluated with a timeout of 3600 seconds. Absynthe solves around 77% of the benchmarks despite being a tool written a Ruby, one of the more slower languages. We suspect additional performance gains can be had by writing the tool in performant language that compiles to native code. We plan to explore this in future work. Additionally, Absynthe does not have the problem of overfitting because the search algorithm does not use the input/output examples. It merely uses it as a test case, and since they do not influence term enumeration they do not cause overfitting with respect to the examples.

Domain-specific synthesis costs. Another key advantage of the Absynthe approach is only pay for what you use. The time of synthesis is dependent on the semantics of the abstract domain. String prefix and suffix are implemented in pure Ruby and does not incur much cost for invoking the solver, so these still guide the search without much cost. However, the string length domain being a solver-backed domain, requires a call to Z3 for every $\subseteq$ check. So it give more precise pruning, while taking a longer time for synthesis. Comparing the average time to generate all the concrete programs explored gives evidence for this. For example, consider *phone-6* which explores

5937 candidates in 100.54 seconds (16.93ms average) with the string length domain, whereas *name-combine-3* explores 117370 candidates in 47.86 seconds (0.41ms average) with the string suffix domain. Depending on how expensive a domain is, one can combine the domains to fit in a variety of synthesis time budgets.

Impact of performance optimizations. We explore the impact of performance optimizations discussed in § 4. First, the performance of Absynthe on these benchmarks when the small expressions cache is disabled is reported in the *No cache* column. It is slower than the baseline across all benchmarks. Notably, *phone-6* and *phone-7* reuse function application subterms. So without caching small expressions, these two benchmarks do repetitive work synthesizing the same expressions in different call sites, resulting in a timeout. Second, the *No Temp* column reports the performance numbers of Absynthe when it is run on these benchmarks with the template inference by testing predicates is disabled. It is slower on most benchmarks than the baseline, and even causing timeouts on some (*phone-7* and *phone-8*). The exceptions are *phone-6* and *name-combine-3*, where the no templates version is faster than the baseline. Recall, that the inferred templates have holes, that have are tagged with a fresh abstract variable $\tilde{x}$ resulting in enumeration of more terms. In contrast, the candidate generation rules (S-) applied during the program search that may potentially synthesize holes with more precise abstractions resulting in less terms being enumerated. We plan to explore mechanisms to infer template holes with more precise abstractions in future work.

5.2 AutoPandas

Benchmarks. We want to test if the approach used by Absynthe, of guiding the search with lightweight abstract semantics combined with testing to ensure correctness, is general enough to be useful for another domain. For this purpose we use the AutoPandas [Bavishi et al. 2019] benchmark suite from its artifact [1] as a case study. The benchmarks are sourced from StackOverflow questions containing the `dataframe` tag. Each benchmark contain the input data frames, additional arguments, the expected data frame output, the list of Pandas API methods to be used in the program, and the number of method calls in the final program.

Bavishi et al. [2019] define *smart operators* to generate candidates and train neural models from a graph-based encoding on synthetic data to rank generated candidates. For a baseline, they consider an enumerative search synthesis engine that naïvely enumerates all possible programs using the methods specified in the benchmark. This narrows down the search space to a permutation of 1, 2, or 3 method calls specified upfront, instead of search over all supported Pandas API. In contrast, Absynthe works like enumerative search, but large classes of programs are eliminated by abstract interpretation of partial programs, or terms are constructed guided by the abstract semantics. Unlike SyGuS, all benchmarks in AutoPandas have only one input and output example. The synthesis goal is a multi-argument Python method that given the specified input produces the desired output.

The evaluation of AutoPandas benchmarks uses the same Absynthe core as the SyGuS evaluation. We wrote a test harness in Python that loads the AutoPandas benchmarks written in Python and communicates with Absynthe core running as a child process. The Absynthe core is responsible for doing the enumerative search, while eliminating programs using abstract interpretation. Any concrete program generated by Absynthe is tested in the host Python interpreter. These operations are performed as inter-process communication over Unix pipes between the host Python harness process and the child Absynthe Ruby process. This allows the testing of generated programs in the host Python process, saving the overhead of launching a new Python process and importing Pandas packages (about 1-3 seconds) for every candidate. If the input/output examples

[1] GitHub: https://github.com/rbavishi/autopandas

are satisfied the synthesis problem is solved, else control is returned back to Absynthe which searches and sends the next candidate for testing.

Abstract Domains. The abstract domains used for AutoPandas benchmarks are:

(1) **Types.** A domain to represent the data type of the computed values (Figure 2c).
(2) **Columns.** A domain to represent dataframes as a set of their column labels (Figure 2a).

Our Python harness infers the data types and the column labels from the input/output examples and the Absynthe core constructs the abstract domain values from PyType and ColNames domains respectively. These individual domains are combined pointwise using the product domain PyType $\times$ ColNames, and Absynthe soundly applies the individual abstract semantics to compute values in the same product domain. The types domain in Absynthe is a wrapper around types from RDL [Foster et al. 2020], a type system for Ruby. Absynthe uses RDL as a library to build the PyType class (the ty field holds an RDL type as shown in Figure 2c). This allows us to reuse prior work that defines nominal types, generic types, finite hash types, singleton types, and their subtyping relations. We define the semantics for these RDL types for the Python language in an abstract interpreter PyTypeInterp to handle features such as standard method arguments, optional keyword arguments, and singleton types as arguments (like `int`). We define the concretization function μ over these types, for example, nominal types can be concretized by all constants of the correct type from the set of constants or the singleton types are concretized to the singleton value itself. The semantics of the type domains are defined in terms of the PyType wrapper that calls into the relevant RDL methods. The example implementation of these domains in § 2 is a simplified version of these domains.

In practice, the AutoPandas benchmarks have input/output examples that are not just data frames, but also integers, Python lambdas, and method references (such as `nunique` from the Pandas library). Absynthe is soundly able to abstract these into the relevant domains. For types, integers become `Integer` and lambdas are inferred as a type `Lambda`. When these values are lifted to the columns domain, they are represented as $\bot$ as these are not data frames, thus there is no way to soundly represent their column labels. Additionally, Absynthe infers a set of constants from the input/output examples as well. It adds any string or numeric row and column labels of the data frames, in addition to any string or numeric standalone values passed as arguments. This set is used to synthesize the constants during the application of the S-Val rules.

Results. Table 2 shows the results of running the AutoPandas benchmarks through Absynthe. The numbers are collected on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16 GB RAM, with a timeout of 20 minutes (consistent with the timeout of Bavishi et al. [2019]). The *Name* column shows the name of the benchmark, *i.e.,* the StackOverflow question ID from which the problem is taken. The *Depth* column shows the length of sequence of method call chain in the final solution. The AutoPandas benchmarks are tuned to synthesize programs with a chain of method calls, where the bulk of the time spend is in synthesizing arguments to these method calls. This is characteristic of the Pandas API which accepts many arguments, often optional keyword arguments. The *Time* column shows the median of 11 runs along with the semi-interquartile range, where – denotes that a benchmark timed out. The *Size* lists the synthesized program size as number of AST nodes. Note that, this number is affected by both the depth of the synthesized program (the number of method calls) and the number of arguments to those methods. *# Tested* lists the number of concrete programs generated by Absynthe that were tested in the Python interpreter. Finally, *AP Neural* and *AP Baseline* shares the benchmarks solved by the AutoPandas neural model and naïve enumeration to aid in comparison with Absynthe. Two benchmarks, *SO_12860421* and

Table 2. Results of running AUTOPANDAS benchmarks through ABSYNTHE. The *Depth* column shares the longest chain of method calls in the synthesized solution; *Time* lists the median and semi-interquartile range of 11 runs for time taken to synthesize a program; *Size* lists the number of AST nodes in the synthesized solution; *# Tested* reports the number of concrete Python programs tested; *AP Neural* and *AP Baseline* shares the benchmarks that AUTOPANDAS neural model and naïve enumeration could synthesize. The benchmarks denoted with a * were a part of the artifact, but not reported in the paper [Bavishi et al. 2019]. Benchmarks highlighted in blue and yellow shows the benchmarks only synthesized by ABSYNTHE and AUTOPANDAS respectively.

Name	Depth	Time (sec)	Size	# Tested	AP Neural	AP Baseline
SO_11881165	1	0.20 ± 0.00	6	40	✓	✓
SO_11941492	1	13.84 ± 0.04	5	2507	✓	✓
SO_13647222	1	-			✓	✓
SO_18172851	1	0.42 ± 0.00	3	70		
SO_49583055	1	3.77 ± 0.01	6	272		
SO_49592930	1	0.22 ± 0.00	3	21	✓	✓
SO_49572546	1	1.50 ± 0.01	3	548	✓	✓
SO_12860421*	1	686.50 ± 1.68	11	1537521		
SO_13261175	1	283.12 ± 0.39	11	237755	✓	
SO_13793321	1	5.70 ± 0.04	6	413	✓	✓
SO_14085517	1	216.14 ± 0.38	7	12844	✓	✓
SO_11418192	2	0.10 ± 0.00	5	11	✓	✓
SO_49567723	2	-			✓	
SO_49987108*	2	-				
SO_13261691	2	65.17 ± 0.17	3	22322	✓	✓
SO_13659881	2	0.21 ± 0.00	6	45	✓	✓
SO_13807758	2	54.92 ± 0.26	6	3144	✓	✓
SO_34365578	2	-				
SO_10982266	3	-				
SO_11811392	3	6.88 ± 0.03	4	921		
SO_49581206	3	-				
SO_12065885	3	0.24 ± 0.00	6	286	✓	✓
SO_13576164	3	-			✓	
SO_14023037	3	-				
SO_53762029	3	545.62 ± 0.91	9	229233	✓	✓
SO_21982987	3	-			✓	✓
SO_39656670	3	-				
SO_23321300	3	-				

SO_49567723, are marked with a * as these were found in the AUTOPANDAS artifact were not reported in the paper.

ABSYNTHE solves 17 programs, the same number of programs as AUTOPANDAS neural model. However, the set of synthesized programs by both tools are different with a significant overlap. Benchmarks listed in Table 2 without any highlight shows the benchmarks that were synthesized by both tools. Benchmarks highlighted in blue were synthesized only by ABSYNTHE but not by AUTOPANDAS. Likewise, benchmarks highlighted in yellow are the benchmarks synthesized only by AUTOPANDAS but not by ABSYNTHE. The time taken to synthesize the programs is largely dictated by how the abstract semantics prunes the space of programs, hence it is proportional to the number of concrete programs generated and tested. The fact that, for the same program size, the number of AST nodes in the method arguments (the difference between size and depth) is indicative of solving time shows that synthesizing arguments is indeed the bottleneck of this benchmark suite.

For example, like *SO_11811392* and *SO_12065885* the type system quickly narrows down the search space, and the solution uses API methods that have 0 or 1 arguments only, making the arguments synthesis quick.

Discussion. Absynthe solves a harder synthesis problem because it does not use the list of methods to be used as provided in the specification. Instead, Absynthe uses the complete set of 30 supported Pandas API for every benchmark. Approximately, this gives us a choice of permutations of size 1, 2, or 3 (depending on the depth of the final solution) from 30 methods, without considering arguments from those methods. In contrast, the baseline enumerative search *AP Baseline* comparison limits the search to only the Pandas API methods that will be used in the final solution. Typically this limits the search space to 1, 2, or 3 methods as given in the specification. In other words, under naïve enumeration, Absynthe explores a strictly larger set of programs than AutoPandas baseline.

In the benchmarks where Absynthe failed to synthesize a solution, it falls back to term enumeration as the abstract domain was not precise. More specifically in the benchmarks with depth 3, Absynthe could do better by jointly reasoning about values in relational abstractions between multiple arguments of the same method. We plan to explore support for relational abstractions in future work. The neural model trained by Bavishi et al. [2019] is good at guessing the sequences that are potentially likely to solve the synthesis task. It, however, does not take into account semantics of the program, thus eliminating impossible programs from being considered. This shows up in *SO_18172851* and *SO_49583055* where both enumerative search and neural models failed, but Absynthe succeeds. Moreover, any updates to the neural model would need to be addressed with a new encoding or a retraining of the model on new data, a potentially resource consuming process. However, exploring the synergy of guidance from abstract interpretation combined with neural models similar to Anderson et al. [2020] to rank *sound* program candidate choices is an interesting future work.

6 RELATED WORK

General Purpose Synthesis Tools. SemGuS [Kim et al. 2021] has the same motivation as Absynthe to develop a general-purpose abstraction guided synthesis framework. However, SemGuS requires the programmer to provide semantics in a relational format as constrained horn clauses (CHCs). While CHCs are expressive and have dedicated solvers [Komuravelli et al. 2016], correctly defining semantics as a relations is prohibitively time-consuming and error-prone. Moreover, SemGuS performs well in proving unrealizability of synthesis problems, but it has limited success in synthesizing solutions. In contrast, Absynthe is a dedicated synthesizer that is geared towards synthesizing programs based on executable abstract semantics. Absynthe can be thought of as an unrealizability prover if coarse-grained semantics, the focus of Absynthe, is sufficient to prove unrealizable. SemGuS also supports under-approximate semantics, which is an interesting future work in the context of Absynthe. Rosette [Torlak and Bodík 2014] and Sketch [Solar-Lezama 2013] are solver-aided languages that use bounded verification using a SMT solver to synthesize programs written in a DSL. In contrast, Absynthe relies on abstract interpretation to guide search, so it can reason about unbounded program properties. There has been parallel work in synthesis using Christiansen grammars [Ortega et al. 2007] that allows one to encode some program semantics as context-dependent properties directly in the syntax grammar. However, an abstract interpreter-based approach gives Absynthe more semantic reasoning capabilities (like polymorphism).

Domain-specific synthesis. SyGuS [Alur et al. 2013] being a standard synthesis problem specification format, has seen a variety of solver approaches. CVC4 [Reynolds and Tinelli 2017] is a general-purpose SMT solver that has support for synthesizing programs in the SyGuS format. CVC4 has complete support for theory of strings and linear integer arithmetic, so it performs better

than Absynthe (which is guided by simple abstract domains) for SyGuS. However, Absynthe's strength is generalizability to other kinds of synthesis problems as demonstrated in synthesis of AutoPandas benchmarks (§ 5.2). DryadSynth [Huang et al. 2020] explores a reconciling deductive and enumerative synthesis in SyGuS problems limited to the conditional linear integer arithmetic background theory. Some of their findings has been adopted by Absynthe (§ 4). EuSolver [Alur et al. 2017b] is an enumerative solver that takes a divide-and-conquer approach. It synthesizes individual programs that are correct on a subset of examples, and predicates that distinguishes the program and combines these into a single final solution. Absynthe is close to EuSolver, as it is also based on enumerative search, but it is also guided by abstract semantics as well. We plan to support synthesizing conditionals in future work.

Past work solves synthesis problems using domain specific abstractions such as types and examples [Frankle et al. 2016; Osera and Zdancewic 2015], over-approximate semantics on table operations [Feng et al. 2017], refinement types [Polikarpova et al. 2016], secure declassification [Guria et al. 2022], abstract domain to verify atomic sections of a program [Vechev et al. 2010], and SQL equivalence relations [Wang et al. 2017a]. These abstraction can be designed as a domain and an abstract evaluation semantics can be provided to Absynthe for synthesizing such programs. However, Absynthe being a general purpose synthesis tool, will not have domain specific optimizations. We plan to explore Absynthe as platform deploying domain specific synthesis in future work.

Abstraction-guided Synthesis. Simpl [So and Oh 2017] combines enumerative search with static analysis based pruning, which is similar to Absynthe. However, the program search in Absynthe can be parameterized by a user provided abstract interpreter allowing the user to write specifications and semantics in a domain fit for the task-at-hand. Additionally, Absynthe can infer abstract values for the holes in partial programs, thus guiding the search using the abstract semantics (Figure 6). Blaze [Wang et al. 2017b] is very similar to Absynthe as it uses abstract semantics to guide the search. It adapts *counterexample guided abstraction refinement* to synthesis problems by refining the abstraction when a test fails, and constructing a proof of incorrectness in the process. However, it starts with a universe of predicates that is used for abstraction refinement, which is a requirement Absynthe doesn't place on users. FlashMeta [Polozov and Gulwani 2015] is similar, but requires the definition of "inverse" semantics for operators using *witness functions*. Absynthe, however, requires only the definition of forward abstract semantics and attempts to derive the inverse semantics automatically where possible.

Learning-based approaches. There has been a recent rise of learning based approaches to make program synthesis more tractable. AutoPandas [Bavishi et al. 2019] is an example of applying neural models to rank candidate choices constructed by other program generation methods (*smart operators* in AutoPandas' case). DeepCoder [Balog et al. 2017] trains a deep neural network to predict properties of programs based on input/output examples. These properties are used to augment the search by an enumerative search or SMT solvers. Absynthe is complementary to these approaches and does not use machine learning. In future, we plan to explore extensions to Absynthe that reorders the program search order using a model learned on program text *and* abstract semantics. EuPhony [Lee et al. 2018], on the other hand, uses an approach inspired by transfer learning to learn a *probabilistic higher order grammar*, and uses that in enumerative search to synthesize solutions. Probe [Barke et al. 2020] learn a probabilistic grammar *just-in-time* during synthesis. Their key insight is that many SyGuS programs that pass a few examples have parts of the syntax that has higher likelihood to be present in the final solution. In contrast, Absynthe is complementary to the approach of learning probabilistic grammars; abstract domains can prune the space of programs, while the grammar can assign higher weights to the terms that should be enumerated earlier. We leave exploring the synergy between these approaches to future work.

7 CONCLUSION

We presented Absynthe, a tool that combines abstract interpretation and testing to synthesize programs. It accepts user-defined lightweight abstract domains and partial semantics for the language as an input, and enables guided search over the space of programs in the language. We evaluated Absynthe on SyGuS strings benchmarks and found Absynthe can solve 77% of the benchmarks, most within 7 seconds. Moreover, Absynthe supports a pay-as-you-go model, where the user only pays for the abstract domain they are using for synthesis. Finally, to evaluate the generality of Absynthe to other domains, we use it to synthesize Pandas data frame manipulation programs in Python from the AutoPandas benchmark suite. Absynthe performs at par with AutoPandas and synthesizes programs with low specification burden, but no neural network training costs. We believe Absynthe demonstrates a promising design choice for design of synthesis tools that leverage testing for correctness along with lightweight abstractions with partial semantics for search guidance.

DATA AVAILABILITY STATEMENT

The latest version of the tool Absynthe is publicly available on GitHub [2]. A snapshot of Absynthe, along with source code, benchmarks used in the paper, supporting scripts and instructions to reproduce our results in § 5 are available as a Docker image artifact [Guria et al. 2023].

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their helpful comments. This research was supported in part by National Science Foundation awards #1900563 and #1846350.

REFERENCES

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. https://ieeexplore.ieee.org/document/6679385/

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017 (EPTCS, Vol. 260)*. 97–115. https://doi.org/10.4204/EPTCS.260.9

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 227:1–227:29. https://doi.org/10.1145/3428295

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 168:1–168:27. https://doi.org/10.1145/3360594

[2] https://github.com/ngsankha/absynthe

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*. Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 422–436. https://doi.org/10.1145/3062341.3062351

Jeffrey Foster, Brianna Ren, Stephen Strickland, Alexander Yu, Milod Kazerounian, and Sankha Narayan Guria. 2020. *RDL: Types, type checking, and contracts for Ruby*. https://github.com/tupl-tufts/rdl

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 802–815. https://doi.org/10.1145/2837614.2837629

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. *Artifact for "Absynthe: Abstract Interpretation- Guided Synthesis"*. https://doi.org/10.5281/zenodo.7824175

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: type- and effect-guided program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, 344–358. https://doi.org/10.1145/3453483.3454048

Sankha Narayan Guria, Niki Vazou, Marco Guarnieri, and James Parker. 2022. ANOSY: approximated knowledge synthesis with refinement types for declassification. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 15–30. https://doi.org/10.1145/3519939.3523725

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1159–1174. https://doi.org/10.1145/3385412.3386027

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449. https://doi.org/10.1145/3192366.3192410

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 229–238. https://doi.org/10.1145/2254064.2254092

Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. 2007. Christiansen Grammar Evolution: Grammatical Evolution With Semantics. *IEEE Trans. Evol. Comput.* 11, 1 (2007), 77–90. https://doi.org/10.1109/TEVC.2006.880327

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodík. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 65–78. https://doi.org/10.1145/3297858.3304059

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Mukund Raghothaman and Abhishek Udupa. 2014. Language to Specify Syntax-Guided Synthesis Problems. *CoRR* abs/1405.5590 (2014). arXiv:1405.5590 http://arxiv.org/abs/1405.5590

Jeff Reback, jbrockmendel, Wes McKinney, Joris Van den Bossche, Matthew Roeschke, Tom Augspurger, Simon Hawkins, Phillip Cloud, gfyoung, Patrick Hoefler, Sinhrks, Adam Klein, Terji Petersen, Jeff Tratner, Chang She, William Ayd, Richard Shadrach, Shahar Naveh, Marc Garcia, JHM Darbyshire, Jeremy Schendel, Torsten Wörtwein, Andy Hayden, Daniel Saxton, Marco Edward Gorelli, Fangchen Li, Matthew Zeitlin, Vytautas Jancauskas, Ali McMaster, and Thomas Li. 2022. *pandas-dev/pandas: Pandas 1.4.4.* https://doi.org/10.5281/zenodo.7037953

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

Andrew Reynolds and Cesare Tinelli. 2017. SyGuS Techniques in the Core of an SMT Solver. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017 (EPTCS, Vol. 260)*, Dana Fisman and Swen Jacobs (Eds.). 81–96. https://doi.org/10.4204/EPTCS.260.8

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10422)*, Francesco Ranzato (Ed.). Springer, 364–381. https://doi.org/10.1007/978-3-319-66706-5_18

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 327–338. https://doi.org/10.1145/1706299.1706338

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (dec 2017), 30 pages. https://doi.org/10.1145/3158151

Received 2022-11-10; accepted 2023-03-31