Trinity: An Extensible Synthesis Framework for Data Science

Ruben Martins
Canergie Mellon University
rubenm@andrew.cmu.edu

Jia Chen
UT Austin
jchen@cs.utexas.edu

Yanju Chen
UCSB
yanju@cs.ucsb.edu

Yu Feng
UCSB
yufeng@cs.ucsb.edu

Isil Dillig
UT Austin
isil@cs.utexas.edu

ABSTRACT

In this demo paper, we introduce TRINITY, a general-purpose framework that can be used to quickly build domain-specific program synthesizers for automating many tedious tasks that arise in data science. We illustrate how TRINITY can be used by three different users: First, we show how end-users can use TRINITY's built-in synthesizers to automate data wrangling tasks. Second, we show how advanced users can easily extend existing synthesizers to support additional functionalities. Third, we show how synthesis experts can change the underlying search engine in TRINITY. Overall, this paper is intended to demonstrate how users can quickly use, modify, and extend the TRINITY framework with the goal of automating many tasks that are considered to be the "janitor" work of data science.

PVLDB Reference Format:
Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An Extensible Synthesis Framework for Data Science. *PVLDB*, 12(12): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Due to the messy nature of data in different application domains, data scientists spend a significant amount of their time performing data preparation and wrangling tasks, which are considered by many to be the "janitor work" of data science. In recent years, *program synthesis* has emerged as a powerful technology that can be used to automate many tedious aspects of data analytics. For example, program synthesizers have been used to automate spreadsheet programming [8, 4, 5], data migration [12], SQL programming [10, 13], and table and tensor transformations [3, 11].

However, one disadvantage of most existing program synthesizers is that they are tied to a very specific application scenario (e.g., number formatting in Excel [8]). While the domain-specific nature of program synthesizers has partially been responsible for their success, developing a useful program synthesizer currently requires an advanced degree in computer science. This paper aims to broaden the applicability of example-guided program synthesis (i.e., *programming-by-example (PBE))* by proposing a *general framework* called TRINITY that can be used to *quickly* develop *efficient* program synthesizers in a new domain. The TRINITY framework is parametrized by a domain-specific language (DSL) and can be easily applied to new application scenarios by providing a suitable DSL and its interpreter. The user can optionally provide (lightweight) specifications of DSL constructs to improve the efficiency of synthesis.

The TRINITY framework is based on recent advances in inductive program synthesis and incorporates the following salient features:

- *Customizability:* TRINITY can be easily adapted to new application domains by providing a suitable DSL (and its semantics) for the target application scenarios.
- *Efficient synthesis:* TRINITY is based on an efficient synthesis algorithm that combines search and lightweight deduction. Specifically, TRINITY performs a search over the space of DSL programs but uses the semantics of the DSL constructs to prune large parts of the search space using lightweight deductive reasoning based on Satisfiability Modulo Theory (SMT) solvers.
- *Usability:* To address the fundamentally incomplete nature of specification in programming-by-example, TRINITY gives users fine-grained control over inductive bias by providing so-called *preference predicates* that constrain the space of synthesized programs.

In what follows, we give a high-level usage overview of the TRINITY framework (Section 2) and then discuss how different users can use, modify and extend TRINITY (Section 3).

2. USAGE OVERVIEW

Figure 1 shows how different users can interact with TRINITY. We split users into three categories: *end-user*, *advanced user*, and *synthesis expert*.

End-user. From the perspective of end-users, TRINITY provides a rich PBE environment that can be used to automate various tasks that arise in data science. In particular, current synthesizers in TRINITY can generate list and table

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

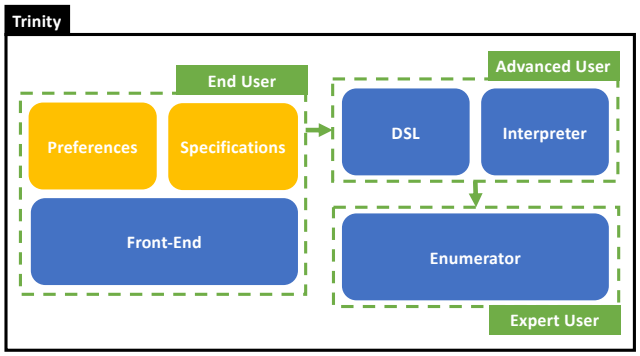

Figure 1: Usage Overview

transformations, SQL queries, and programs that consolidate values from different sources. To perform a synthesis task, the end-user specifies her intent by using input-output examples. Additionally, the user can (optionally) also specify so-called *preference predicates* that introduce inductive bias to the synthesizer. For example, if the user thinks that the synthesized program should contain the `concat` function for concatenating two strings, she can provide this information to TRINITY by adding a predicate like `occurs(concat, 80%)`, where 80% indicates the user's confidence that the target program will contain the `concat` function. Given such a predicate, TRINITY will bias its search towards programs that contain at least one occurrence of the `concat` operator.

Advanced user. From the perspective of an advanced user (or domain expert), TRINITY can be used to build new synthesizers or extend the functionality of existing synthesizers. For example, to build a new synthesizer on top of TRINITY, the domain expert only needs to provide a domain-specific language (DSL) that is suitable for the target class of tasks. In addition to the syntax of the DSL and its interpreter, the user can also provide logical specifications for the DSL constructs in order to speed up synthesis. However, these logical specifications need not precisely specify the semantics of the DSL construct. For example, consider the `concat` operator that concatenates two strings `a` and `b` and returns a string `c`. We can encode the semantics of this operator using the logical specification `len(c)=len(a)+len(b)`, where `len` denotes the length of the string. Note that this specification gives a partial rather than complete description of what the `concat` operator does. Internally, TRINITY's synthesis algorithm uses such specifications to dramatically prune the search space of possible programs.

Synthesis expert. For users with some knowledge of program synthesis, TRINITY's synthesis engine can be customized to more efficiently search for programs that satisfy the user-provided examples. At its core, TRINITY's synthesis algorithm performs backtracking search over DSL programs but leverages program semantics to significantly prune the search space. Furthermore, the search algorithm in TRINITY is *conflict-driven* in that it can learn useful "lemmas" based on failed synthesis attempts (i.e., conflicts) [2]. An advanced user with some knowledge of program synthesis can further customize the underlying algorithm in TRINITY by either changing its search strategy or by integrating a statistical model (e.g., in the style of DeepCoder [1]) into the enumeration engine.

3. DEMO

In this section, we demonstrate three use cases of TRINITY: (i) Synthesizing programs for data wrangling tasks in R, (ii) extending the DSL for data wrangling tasks to synthesize new programs, (iii) changing the enumerator to incorporate statistical models trained from a corpus.

3.1 Data Wrangling in R

DEMO 1. *To get a sense of the data wrangling tasks that we can automate using* TRINITY*, consider a data frame reshaping tasked posted on StackOverflow* [1] *where a data scientist wants to convert data from long to wide format. To get help from other StackOverflow users, he provides the input data frame shown in Table 1 and the output data frame in Table 2.*

Table 1: Input table

Student	Grade	Score1	Score2
Greg	A	75	76
Greg	B	86	85
Sally	A	85	86
Sally	B	80	78

Table 2: Output table

Student	B_Score1	B_Score2	A_Score1	A_Score2
Greg	86	85	75	76
Sally	80	78	85	86

Given only *this simple input-output example,* TRINITY *can automatically synthesize the following R program using the* `tidyr` *library:*

```
df1=gather(input,Score,Grade,Score1,Score2)
df2=unite(df1,AllScores,Time,Score)
output=spread(df2,AllScores,Grade)
```

While not necessary for this relatively simple example, the user can optionally help the synthesizer by providing preference predicates*, such as* `occurs(gather, 60%)`*, which would cause the synthesizer to bias the search towards programs that contain the* `gather` *function from the* `tidyr` *library.*

DEMO 2. *As another example, consider the following scenario described on a StackOverflow post* [2] *where a user needs to collapse multiple columns of a data frame into a single column. In particular, the diagnosis columns (1-3) in Table 3 correspond to binary values that denote the presence or absence of the diagnosis. The user would like to reshape her data from the format shown in Table 3 to the format shown in Table 4 but fails to do so after many attempts as described in the StackOverflow post:*

"No matter how many times I read the documentation on reshape/reshape2/tidyr I just can't manage to wrap my head around their implementation." [2]

TRINITY *can help this user by automatically synthesizing the following R code using the* `tidyr` *and* `dplyr` *libraries:*

[1] `https://stackoverflow.com/questions/29775461`
[2] `https://stackoverflow.com/questions/29447325`

Table 3: Input table

ID	Diagnosis_1	Diagnosis_2	Diagnosis_3
A	1	0	0
A	1	0	0
B	0	1	0
C	0	1	0
D	0	0	1
E	0	1	0
E	1	0	0

Table 4: Output table

ID	Diagnosis
A	1
A	1
B	2
C	2
D	3
E	2
E	1

```
df1=gather(input,Patient,Value,-‘ID‘)
df2=separate(df1,‘Patient‘,
             into=c(’Patient’,’Diagnosis’))
df3=filter(df2,‘Value‘ > 0)
output=select(df3,‘ID‘,‘Diagnosis‘)
```

How does it work? The data wrangling synthesizer built into Trinity is based on a DSL (a subset of which is provided in Figure 2) that uses functions provided by the `tidyr` and `dplyr` libraries. As mentioned earlier, Trinity performs an enumerative search over the space of programs in this DSL but uses lightweight deductive (SMT-based) reasoning to significantly prune the search space [2]. In particular, each of the DSL constructs shown in Figure 2 come equipped with a logical specification that can be used to prove that no program of a certain shape will be consistent with the input-output examples. Such logical reasoning capabilities dramatically reduce the space of programs that need to be explored by Trinity.

As mentioned earlier, Trinity can also use preference predicates optionally provided by the user to guide its search. In addition to the `occurs` predicate discussed earlier, Trinity also supports other unary and binary predicates that can be used to constrain the search space. For instance, a predicate such as `is_parent(foo, bar)` indicates that the return value of `bar` is an argument of function `foo`. Internally, Trinity uses a Maximum Satisfiability Modulo Theory (Max-SMT) solver to convert these preferences into a set of *soft constraints* and prioritizes its search strategy to *maximize* the satisfaction of these constraints.

3.2 Extending the DSL for Data Wrangling

Demo 3. Trinity *is a modular synthesizer that allows the user to easily extend existing synthesizers or even create new ones. For instance, suppose that the user wants to extend the data wrangling synthesizer described in Section 3.1 by adding a new DSL function called* `summarise` *that aggregates values in a column. In this case, the user can extend the synthesizer by just adding this new* `summarise` *function*

```
TABLE → input | select(TABLE, LIST)
        unite(TABLE, COL, COL) | separate(TABLE, COL)
        spread(TABLE, COL, COL) | gather(TABLE, LIST)
        filter(TABLE, COL, OP, C)
 LIST → [1] | [1,2] | ...| [4,5]
  COL → 1 | ...| 5
   OP → < | = | >
    C → 0 | ...| 10
```

Figure 2: Part of the DSL for data wrangling tasks in R

to the existing DSL:

```
TABLE → summarise(TABLE, AGG, COL)
  AGG → min | max | mean | sum
```

Optionally, the user can also provide (coarse) specifications that will be used by Trinity *to prune infeasible programs. For instance, the user may provide the following specification for* `summarise`*:*

```
1  func summarise: Table r -> Table a,
2                  Aggr b, ColInt c {
3    row(r) < row(a);
4    col(r) <= col(a) + 1;
5  }
```

Here, lines 1 and 2 specify the signature of the `summarise` *function, which takes as input a table* `a`*, an aggregate function* `b`*, and a column* `c` *and returns a table* `r`*. The logical specification in lines 4–5 describes the relationship between the input and output table with respect to the number of rows and columns. Finally, the user also needs to provide an interpreter for the new DSL construct. For instance, the following “interpreter” for* `summarise` *is a simple wrapper that invokes the corresponding function in R:*

```
1  class MorpheusInterpreter(Interpreter):
2  def eval_summarise(self, node, args):
3    ...
4    _script =
5    ’{ret_df} <-
6     {table} %>% summarise({TMP} =
7     {aggr} (.[[{col}]]))’
8    .format(ret_df=ret_df_name,
9            table=args[0],
10           TMP=get_fresh_col(),
11           aggr=str(args[1]),
12           col=str(args[2]))
13   return robjects.r(_script)
```

How does it work? Given a DSL and an interpreter, Trinity can synthesize any well-typed programs that can be implemented in that DSL. Specifically, Trinity uses the provided interpreter to check if a given program is consistent with all of the input-output examples. In addition, the logical specifications provided by the domain expert allow Trinity to prune the search space by (a) determining whether a partial (i.e., incomplete) program has any completions that can satisfy the input-output examples, and (b) identifying a set of programs that are equivalent to previously rejected programs. By decoupling the semantics of the DSL from the inner workings of the synthesizer, Trinity allows users to build effective synthesizers on top of its underlying search engine without having to write a new synthesizer from scratch.

3.3 Modifying the Search Engine

DEMO 4. *Expert users can also customize the underlying search engine of* TRINITY *to further speed up their synthesizer. In particular, users can provide statistical models (e.g., deep neural network, n-gram) that can be used to predict the most likely programs. Then,* TRINITY*'s underlying search engine explores programs in this DSL by consulting the provided statistical model. That is, programs that are assigned a higher score by the model are prioritized during the search. Such a statistical model could be obtained in a variety of ways: For instance, if there are many existing programs in the target DSL, the user could train their model on an existing code corpus and enumerate programs according to their frequency in the corpus. Alternatively, the model could also be guided by the examples rather than existing code: For instance, one could train a deep neural network to predict which DSL constructs are likely to occur in the target program based on the input-output examples [1].*

How does it work? TRINITY views each program as an Abstract Syntax Tree (AST), and its search engine enumerates ASTs that correspond to well-typed programs in the DSL. The enumeration of ASTs is guided by the statistical model. Furthermore, TRINITY supports both explicit and symbolic search, and the expert user can integrate their statistical model into both of these search strategies. For explicit search, TRINITY performs backtracking search over ASTs and uses the statistical model to decide which AST node to expand next and how to expand it. For symbolic search, the predictions of the statistical model are expressed as preference predicates which are then encoded as soft constraints for a MaxSMT solver. Since the solver finds assignments that satisfy the maximum number of soft constraints, the programs that are enumerated first correspond to those deemed more likely by the model. Both the explicit and the symbolic search options have their respective advantages: For instance, while it is easier to incorporate the statistical model into the explicit search mode, the symbolic search option allows TRINITY to perform more aggressive pruning by learning new constraints from failed synthesis attempts.

4. RELATED WORK

The TRINITY system is related to a long line of work on programming-by-example (PBE) that has gained increasing popularity over the last several years [4, 2, 3, 10, 14]. In what follows, we discuss other related work in this space.

The existing synthesizers built on top of TRINITY are most closely related to other PBE systems that automate table transformations [10, 14, 6, 5]. However, we are not aware of any other synthesizers (other than the ones built on top of TRINITY) that automate data wrangling tasks.

TRINITY is a best viewed as a general framework for quickly developing program synthesizers for new domains. As discussed in Sections 3.2 and 3.3, advanced users can build new synthesizers on top of TRINITY by coming up with new DSLs for the target application domain or customizing TRINITY's underlying search strategy. Thus, TRINITY is not just limited to automating data wrangling tasks in R but can target other programming languages as well as other types of tasks. Viewed as a general synthesis framework, TRINITY bears resemblance to other synthesis frameworks such as Prose [7], Rosette [9], Blaze [11], and Neo [2]. Among these, the most closely related one is Neo. In particular, TRINITY is a second-generation version of Neo that extends it in several ways. First, TRINITY gives finer-grain control to users by allowing them to provide preference predicates to guide search. Second, the underlying search engine of TRINITY incorporates a MaxSMT solver that ensures maximal satisfaction over user preferences. Finally, it is much easier to build new synthesizers on top of TRINITY compared to Neo. TRINITY differs from other synthesis framework in that it combines conflict-driven backtracking search with deductive SMT-based reasoning. We believe that this design makes TRINITY quite extensible, as one can build efficient synthesizers on top of TRINITY by just providing coarse, easy-to-write logical specifications of DSL constructs.

5. CONCLUSION

We have presented TRINITY, a novel synthesis framework (written in Python) that makes it easy to build new synthesizers. We have also presented a program synthesizer built on top of TRINITY that simplifies tedious aspects of data analytics by automating a wide range of data wrangling tasks that arise in the context of R programming. The documentation and source code of the TRINITY framework is available at `https://fredfeng.github.io/Trinity/`.

Acknowledgments. This work is supported in part by NSF awards #1762299, #1762363, #1811865, and CMU/A-IR/0022/2017 grant.

6. REFERENCES

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

[2] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *PLDI*, 2018.

[3] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, 2017.

[4] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[5] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[6] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.

[7] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *OOPSLA*, 2015.

[8] R. Singh and S. Gulwani. Synthesizing Number Transformations from Input-Output Examples. In *CAV*, 2012.

[9] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

[10] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *PLDI*, 2017.

[11] X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. In *POPL*, 2018.

[12] N. Yaghmazadeh, X. Wang, and I. Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *PVLDB*, 11(5):580–593, 2018.

[13] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. In *OOPSLA*, 2017.

[14] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *ASE*, 2013.