University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

Hasan Selman Kara & Patrick Burkhalter

Data Wrangling Using Programming by Example

Extending String Manipulation with Known-Entity-Translations

Bachelor's Thesis

School of Engineering
University of Applied Sciences and Arts Northwestern Switzerland (FHNW) Brugg-Windisch

Supervision

Simon Felix

Customer

Shouldcosting GmbH

March 2019

Abstract

Millions of people need to analyze raw data of varying quality to support their job function. In a lot of cases the present data needs to be refined and processed from its raw form to a more structured format in order to analyze and visualize it. This process is *time-intensive and error-prone*. Usually tools like Microsoft Excel or OpenRefine are used to improve that process. However, any non-trivial data wrangling task remains inaccessible to non-programmers. Furthermore, existing tools only cover the most common use cases, i.e. they are *business agnostic*.

Programming by example (PBE), a subfield of program synthesis, allows for a natural and intuitive user interaction mechanism. Users can specify their intent by providing *input output examples* for string transformations.

Using PBE for data wrangling is *not a novel idea*. Flash Fill is a PBE feature in Excel and Microsoft's PROSE framework shows the potential of PBE for data wrangling. However, these existing solutions cannot be integrated into custom software.

Therefore, we developed a custom PBE system for data wrangling tasks. We describe a custom domain-specific language (DSL) for complex string manipulations. We present a data structure to represent a large number of programs built in our DSL. We introduce a custom synthesis algorithm that populates our data structure with programs to transform string inputs to their intended outputs.

Our PBE system *outperforms the baseline in every test set*. The state of art system *PROSE performs better than our system in one case only*. With our PBE system the user only needs to provide 1.06% (baseline 2.98% and PROSE 1.24%) of the outputs, all other outputs are independently processed by the system. Furthermore, our synthesis algorithm converges rapidly and takes a *fraction of a second* to synthesize programs. Additionally, by extending the general string manipulation PBE system with known-entity-translations we showcase how extensible the system is for business-specific requirements. The synthesizer has been integrated and deployed to an existing custom data wrangling tool of our customer. This report serves as a guide for others to determine the feasibility of developing a custom PBE system.

Keywords: Program Synthesis, Programming by Example, Data Wrangling.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms and Abbreviations

CNF	Conjunctive Normal Form
CSP	Constraint Satisfaction Problem
DIC	Data Integrity Checker
DSL	Domain-Specific Language
ERP	Enterprise Resource Planning
PBE	Programming by Example
PROSE	Program Synthesis using Examples
PSI	Predictive Saving Identifier
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
SVM	Symbolic Virtual Machine
SyGuS	Syntax-Guided Synthesis
SyGuS-Comp	SyGuS Competition
SyGuS-IF	SyGuS Input Format

Chapter 1

Introduction

Millions of people need to analyze data in various forms to support their job function. In a lot of cases the data present needs to be refined and processed from its raw format to a more structured format in order to analyze and visualize it. We refer to that process as *data wrangling*.

Data wrangling is a time-intensive task. Today, people use software such as Microsoft Excel or OpenRefine to help with this process. These tools offer a lot of power and flexibility, but they have some downsides.

Firstly, they have an accessibility problem for non-programmers. A supposedly simple task such as extracting the string "300CHF" from an input "W42-**300CHF**-Joe" is non-trivial. One possible Excel formula would look like the following:

```
=MID(A2, SEARCH("-",A2) + 1, SEARCH("-",A2,SEARCH("-",A2)+1) - SEARCH ↵
("-",A2) - 1)
```

Writing such a program would be challenging for a programmer inexperienced in the Excel programming model — let alone for an analyst without a programming background. These users do not want to become developers, they just want to have their data in an appropriate format to suit their needs.

Secondly, existing tools are *business agnostic*, as they were developed to cover the most common use cases. Excel does not know anything about the business field the user is working in. This means for example that Excel cannot easily translate known entities such as different industry norms, e.g. transform a DIN value to ISO.

PBE systems (Programming by Example) can help to make data wrangling a less dreadful experience by being business aware and accessible to any user. PBE is a subfield of program synthesis, where the specification comes in the form of input output examples [2]. Given certain specification — in our case string input output examples — a program synthesizer deduces the desired intent and synthesizes a program that satisfies the specification. The synthesized program can then be applied on other inputs. Programming by example allows a user to create programs without programming conventionally [3].

In this report we present a custom-built PBE system for data wrangling. We showcase the reasons to develop a custom PBE system and how our system can be extended to new business needs.

1.1 Objective

One objective of our work is to make programming more accessible to non-programmers — namely data analysts at Shouldcosting GmbH. An approach to achieve this "democratization of program-

ming" is by building systems that allow for natural and intuitive interaction mechanisms for users to specify their intended tasks so that even non-programmers can perform programming tasks [4].

There are already projects based on PBE in the domain of data wrangling and string manipulation. These and other program synthesis projects are discussed in Section 1.4.1. Special mention deserves the work by Sumit Gulwani and the PROSE (Program Synthesis using Examples SDK) research team at Microsoft — which is lead by Gulwani. Microsoft PROSE SDK is a general purpose program synthesis framework for automatic programming or data wrangling from input output examples. Gulwani's PBE work led to the Flash Fill feature of Microsoft Excel, which is the inspiration behind this project.
Sadly, Microsoft's PROSE SDK is not commercially available[1]. Furthermore, integrating Flash Fill — which runs inside Excel — with our customer's current product (see Section 1.2) is not feasible.

Thus, we need to research and develop a new product with the following properties:

- **Commercially available** The product must be available for commercial use.
- **Reproducible** Other people facing a similar problem can use our report to create their own PBE system.
- **Extensible** The system can be extended for different business needs. In our work we enhance our general string transformation PBE system with known-entity-translations. This requirement is a direct business need of our customer who wants to translate string values between different material norms.

1.2 Context

Shouldcosting identifies cost-saving potential in the incoming goods department of companies. Their customers provide them with data from different sources such as their ERP systems. Analysts at Shouldcosting use a custom-built software called PSI (Predictive Saving Identifier) to analyze the customer data and to find saving potentials.

Most customers come from the industrial sector. They store their data in insufficient ways, i.e. inhomogeneous data is stored that later makes computational analysis difficult. Therefore, the data needs to be pre-processed before feeding it into the PSI. To assist in that process Shouldcosting commissioned the DIC (Data Integrity Checker) project at FHNW. The result is a web application — as seen in Figure 1.1 — that helps the user visualize and decrease the variance of a given data set.

This previous product is used in production at Shouldcosting. Our project is an extension of the old DIC with a PBE functionality. This means we inherit the existing code base and extend it.

1.3 Interaction Model

Simply developing a program synthesizer is not enough to satisfy user needs. Considering and choosing an appropriate interaction model is crucial for user adoption [5]. Users want to communicate their intent in a simple way and as intuitively as possible.

In this chapter we describe the interaction model used in DIC for the new PBE feature. As seen in Figure 1.1, the DIC displays the contents of an uploaded data set (a CSV file) as a table and allows multiple actions that can be applied on a selected column. In order to ensure seamless integration, we simply added two more possible actions that can be performed on the data set —

[1] Clarified after an e-mail inquiry with Microsoft (03.10.2018)

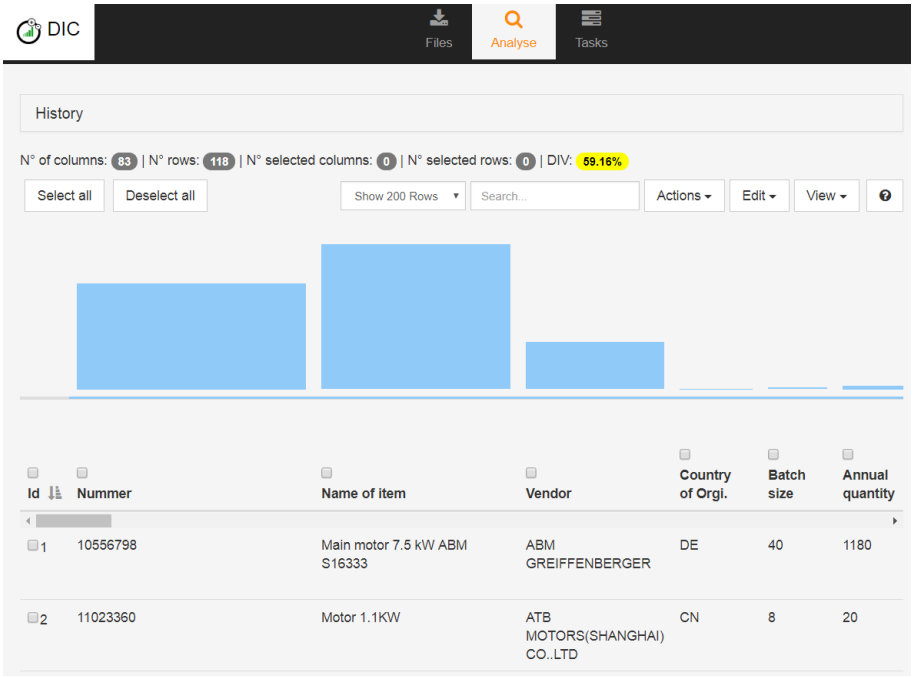


Figure 1.1: The analysis page of the first version of the DIC web application. Here the user can perform various actions on the displayed data set such as “Reduce Text Variance” or “Prewash”.

`Add PBE` and `Start PBE` — instead of adding a completely new page just for PBE. Thus, the new PBE feature is integrated into DIC and can be used like all the other data manipulation operations.

The difference between `Add PBE` and `Start PBE` is that the former adds for a selected column a new PBE column, whereas the later starts the PBE process for two selected columns — a normal column as the input and the a PBE column as the output.

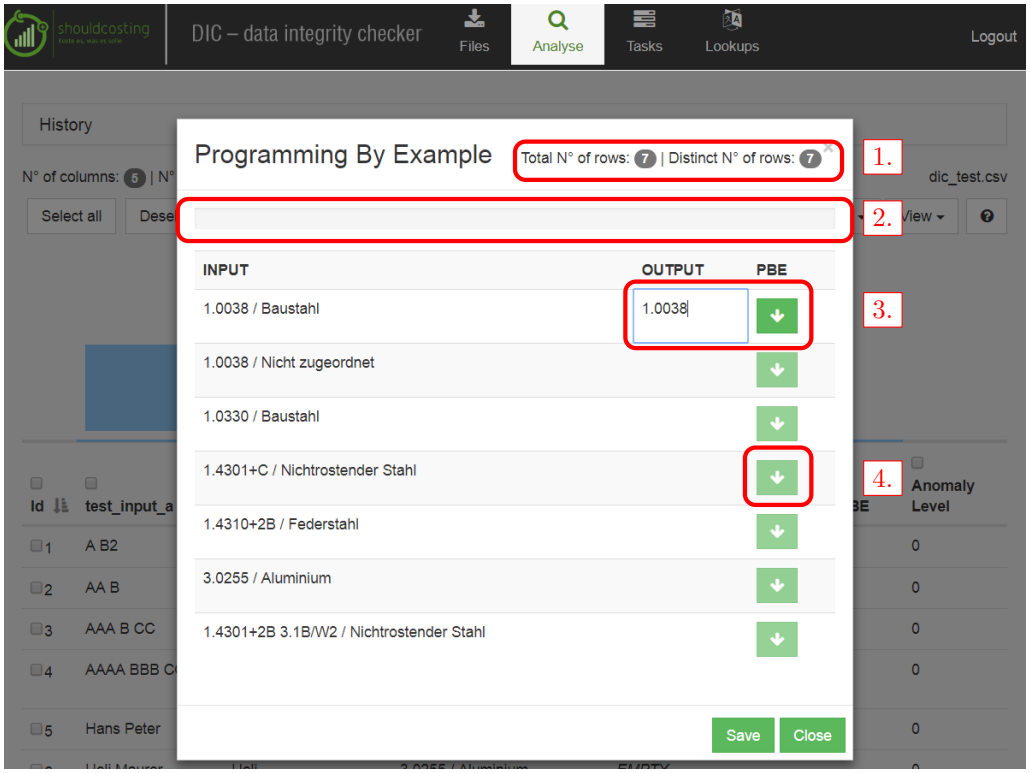


Figure 1.2: The new PBE pop-up shows the interaction model with the PBE system. (1) Shows the number of rows and the number of distinct rows, as the pop-up visualizes a distinct view over the data set because applying different programs on the same input does not make sense. (2) A progress bar shows how long it takes until a synthesizer timeout occurs. Sometimes synthesizing a program takes longer than acceptable for a user. Therefore, we give a visual feedback for how long the user has to wait at most. (3) Entering an example output by hand enables the PBE button. (4) As no output is provided, PBE cannot be started from this index.

The `Start PBE` action opens a pop-up (see Figure 1.2) where the user can enter a output example for a given input. Clicking the downwards arrow starts the program synthesizer. As input, the synthesizer takes all examples that are entered by hand and that are above the clicked arrow. The synthesized program is applied on each input below the clicked arrow that does not already have an output provided by the user.

The rationale behind this interaction model is that a user will start the data wrangling from the top, provide some examples, start the synthesizer and proceed to check whether some outputs below are wrong. If an erroneous output is found, the user can fix the output, restart the synthesizer and repeat the process. Thus, we can safely assume that all outputs above the clicked arrow are correct and do not have to be edited again. Therefore, we increase user trust in the system by not performing unexpected actions.

1.4 State of Research

Program synthesis is an active research topic and is considered by some to be the Holy Grail of computer science. Instead of telling a computer *how* to solve a problem, we tell a synthesizer *what* we want to solve. This chapter describes the us of program synthesis, the components of program synthesizers — user intent, program space and search technique — and their challenges.

1.4.1 Synthesis of Programs in Many Domains

Before we dive deeper into possible ways to achieve program synthesis, we look at the different domains it is currently used in. In its most general formulation, every problem is amenable to synthesis if one can give the problem an executable semantic [6]. Such problems include:

1. String manipulations [7, 8, 3]
2. Data wrangling [9, 10]
3. Memory model checker [11]
4. Hints for solving education problems [4]
5. Network protocols [12]
6. Cell biology [13]

   ...

Despite all these domains being vastly different, the components used in their program synthesizer are similar.

1.4.2 User Intent Specification

For a program synthesizer to deduce the users intent it has to be specified somehow. In this subsection we present three different specification mechanisms. These mechanisms differ in various aspects. Some specifications are more ambiguous than others. The ease of use, i.e. how hard it is to provide a specification, differs for each mechanism. Another aspect is how easy a synthesized program can be verified for correctness against the given specification.

Formal Specification Given a user-provided formal specification, an automated theorem solver is applied to construct a proof which is used to deduce the corresponding program. Therefore, by using a formal specification the user can specify the intent as detailed as desired and the generated proof can be used to verify the synthesized program. However, giving a complete formal specification is in many cases as complicated as writing the program itself [14].

Concrete Input Output Examples Input output examples provide an exact description of what the output should look like. Given one or more such input output examples a synthesizer deduces a pattern for similar inputs. This method is commonly used — e.g. in Flash Fill — as its usability is great. The drawback is that it leaves the user intent *under-specified* as the specification is very ambiguous. Also, the only possible solution for exact verification is to use the same input output examples on the synthesized program to check correctness.

Multimodal input Instead of limiting specification mechanisms to a single option, we can combine multiple options. Furthermore, by going beyond the traditional requirement of composing a syntactically correct sequence of instructions in a finite set of language instructions, a new paradigm for interactive programming using multimodal natural input is facilitated [14].

Examples for possible multimodal user inputs include examples, demonstrations, natural language, keywords, and sketches.

1.4.3 Program Space

The program space defines which components can be used by the synthesizer to create a program which fulfills the given requirements. This search space, also called hypothesis space, can either be predefined or dynamically extensible.

A predefined program space is a domain-specific language (DSL) defined by the programmer of the synthesis system. The developer uses the knowledge of the domain where the synthesizer is used to find a compromise between expressiveness of the language features and the number of features. A case where a fixed DSL is used is the Flash Fill feature in Microsoft Excel [7] which is used for string transformation.

An example for a dynamically extensible program space is the Automated Feedback Generation for Introductory Programming Assignments [15]. This paper describes a tool for programming assignments at university where the instructor can define models to predict errors made by students. These error models are used by the synthesizer to find bugs in the students code and provide a meaningful error message, thus reducing the workload for teaching assistants.

For our program synthesizer a static DSL is used. The reason is that our customer provides us with the domain knowledge needed for the DSL and there is no need for dynamic elements in the program space.

1.4.4 Search Technique

"In its most general formulation (for a Turing-complete programming language and an arbitrary constraint) program synthesis is undecidable, thus almost all successful synthesis approaches perform some kind of search over the program space." [14]

This search over the program space can be conducted in a number of ways. In this subsection we present three different search techniques.

Enumerative search The most simple approach to find valid programs is to brute force all possible combinations of elements in the program space and to test them against the specification. This simple approach is infeasible for almost all use cases [16].

The more sophisticated approaches use pruning strategies to control the size of the possible programs. Possible implementations of this are the different approaches to tree search like top down, bottom up or bidirectional [14].

Furthermore, Gulwani presents a way to enumerate possible programs in a graph. This approach is primarily constructed for string transformations. It only enumerates programs which fulfill at least a part of the specification and represents them as edges in a graph. After the graph is populated the paths that lead from the source to the target are valid programs [7].

Stochastic search The stochastic search approaches the problem of finding programs in the hypothesis space with probability.

One possibility is to represent all programs as a node in a graph where each edge represents a single change to a program which leads to the other program. A size function then is used to apply the Metropolis-Hastings Algorithm on this graph [14]. This algorithm interprets the graph as a Markov Chain and uses Monte Carlo sampling to estimate the best program defined by the size function [17].

Genetic Programming can be used to iteratively make random changes to programs and continue with the programs that have the best score in a given fitness function. Furthermore, it allows sharing promising code fragments between different breeds of programs [16].

Using machine learning is another method to program synthesis. A possible approach is to define a context-free grammar where the probability of occurrence is trained for each grammar rule. These weighted rules are used to decide which grammar elements should be used for the program [18]. A flaw in this search technique is that the contextual information of the input cannot be used [14].

Lastly there are two different deep learning approaches. With program induction a neural network is trained to be used as a program directly. Alternatively a neural network can be used to produce a program like the other synthesis algorithms [14].

Constraint Solving The search space is limited by providing constraints to the possible solution. Finding valid solutions in the search space is a *constraint satisfaction problem* (CSP). CSPs are expressed in the following form. "Given a set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints, find values of the variables that satisfy every constraint." [19]

CSPs are usually solved with SMT-solvers (satisfiability modulo theories) instead of the low level SAT-solvers (boolean satisfiability). "SMT-solvers are frontend to SAT-solvers, i.e. they translate input SMT expressions into conjunctive normal form (CNF) and feed SAT solvers with it." [1] Figure 1.3 describes how to solve an example linear equation system with the Z3 SMT-solver.

$$\left.\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y - 4z &= -2 \\ -x + 0.5y - z &= 0 \end{aligned}\right\}$$

```
from z3 import *
x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model() # -> [z = -2, y = -2, x = 1]
```

Figure 1.3: Solving a simple equation with Z3 SMT solver in python [1]

Remark. A system of linear equations can hold certain constraints which might not have solution, i.e. the problem is *unsatisfiable*.

Real-world applications can have thousands of constraints. In general, solving constraint satisfaction problems are NP-complete (Cook-Levin theorem). Therefore, SAT solvers prune the search space as early as possible.

Encoding a problem in SMT — let alone directly in SAT — is very time-intensive and nontrivial. In order to make program synthesis more accessible to programmers several frameworks are available. These frameworks allow solver-aided programming. Programmers can specify synthesis problems in a higher-level programming language with new constructs for constraint solving. Such constructs enable embedding second-level subproblems into regular programs in the host language [14].

Rosette is an example of a solver-aided programming framework [20]. Its host language is Racket — a Lisp type programming language — which runs on a custom symbolic virtual machine (SVM). Programmers can define custom solver-aided domain-specific languages (SDSLs) which frees the developer from compiling their language into SAT/SMT constraints [21].

We considered using Rosette for our work. However, we chose not to follow this approach. One of our project objectives is to make program synthesis more accessible to other programmers, so that they can extend our solution. Solutions based on SAT/SMT remain mostly too difficult for

people not well versed in program synthesis — despite the best efforts in the space of solver-aided programming by Torlak et al.

Chapter 2

Problem Definition

Our main goal is to create a PBE system that performs best for the data used by Shouldcosting, not a general purpose PBE system. Shouldcosting provided us with a set of already processed data, which we use to build and measure the algorithm.

2.1 Data Analysis

In this section we will conduct a data analysis of the provided data set. The first aspect to consider is the variance. This strongly impacts the number of examples needed, as duplicate input output examples only need to be processed once. As seen in Figure 2.1 some of the inputs have no duplication at all and others have up to 99.5% duplication.

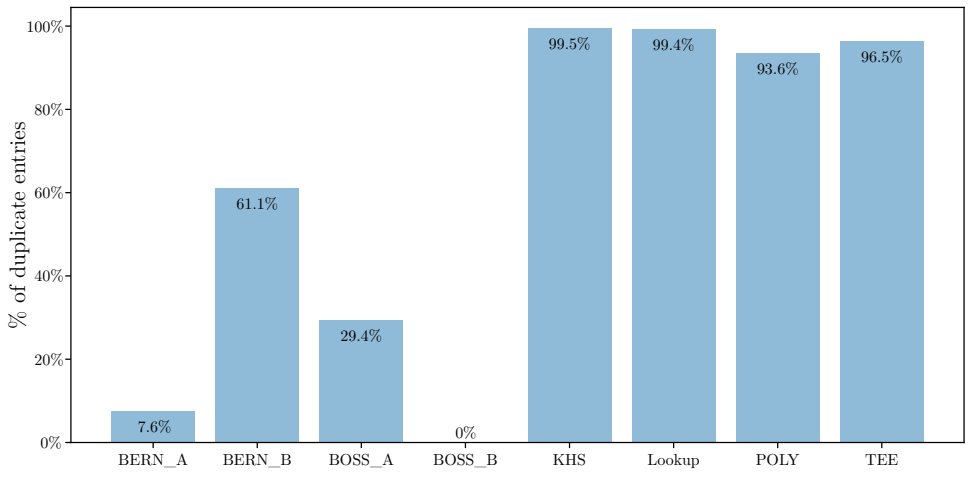


Figure 2.1: Percentage of duplicate records in test set.

Furthermore, we need to analyze the special characters used in the test sets. This is used to decide which special character the PBE system needs to be able to process. Therefore, we checked the data set provided by the customer and plotted the frequency of characters, shown in Figure 2.2. The chart does not include the test set “BOSS B”, as it contains only alphanumeric characters.

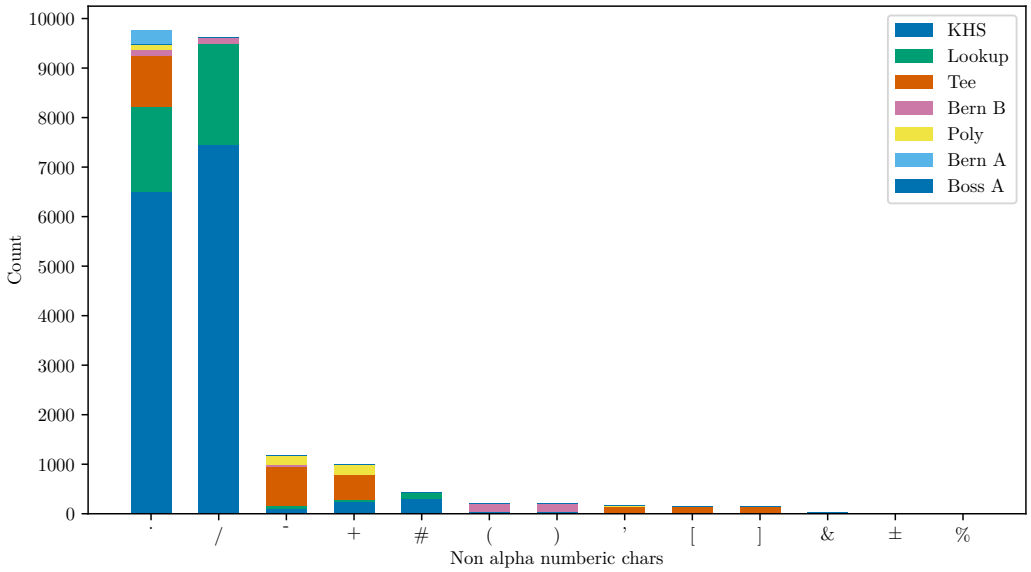


Figure 2.2: Distribution of non-alphanumeric characters in test sets.

Remark. Additionally we added a `Lookup` file which provides test cases for the known-entity-translation. This is necessary because these use cases are not contained in the provided data set. The basis of this `Lookup` file is an existing test file where we translated the material norms.

2.2 Examples

In this section we describe some concrete input output examples selected from the real-world data set provided by the customer. These examples illustrate what concrete use cases our PBE system must be able to handle. Furthermore, we see how heterogeneous the test set is.

Example 1. Most input output examples require some sort of string extraction, i.e. the intended output is constructed of one or more substrings from the input. Table 2.1 lists some string extraction examples.

Input	**Output**
1.0038 / Nicht zugeordnet	1.0038
1.4301+2B 3.1B/W2	1.4301
DC01+ZE (**1.0330**+ZE) / Stahl	1.0330
X8CRNIS18-9+C W-NR.**1.4305**	1.4305
GGG-60 W-NR.**0.7060**	0.7060
EN AW-2007 T4 [ALCU4PBMGMN]	EN AW-2007 T4
X 10 CrNi 18-8 (Rm 1100-1250) / SUS 301 (**1.4310**)	1.4310
1010325.B.00	1010325
0**370745**.B.00	370745
00**16415**.0.01	16415

Table 2.1: Multiple input output examples requiring some sort of string *extraction*.

Example 2. In some cases extracting substrings from input is not sufficient to construct the output because the input lacks certain characters used in the intended output. Common use cases for this requirement are for example adding punctuation or changing the capitalization. Table 2.2 lists the few string manipulation examples in our data set.

Input	**Output**
St37K	St37**k**
S235JR	**St**235**k**

Table 2.2: Multiple input output examples requiring some sort of string *manipulation*.

Example 3. One of the most desired requirements of our customer is the translation of known entities. Data in norm A must be converted into norm B. This could mean that *none* of the input is used to construct the output. Rather, the input is used a lookup key for its corresponding output. Therefore, this use case differs heavily from Example 2. As seen in Table 2.3, the output is completely different from the input.

Input	**Output**
EN AW-5754 / Aluminium	3.3535
1.4301+2B 3.1B/W2 / Nichtrostender Stahl	X5CrNi18-10

Table 2.3: Multiple input output examples requiring some sort of *lookup*.

Example 4. One program might not work on all inputs as intended as different kind of inputs might need completely different programs. However, there can be inhomogeneous data in the same data column. Consider the following list where the user intends to extract the bold substring from the respective strings.

1. **1.0036** / Baustahl
2. **1.0038** / Nicht zugeordnet
3. **1.4301** X5...810 / Nicht zugeordnet

   ⋮

31. DC01+ZE (**1.0330**+ZE) / Stahl

    ⋮

5032. DC01+ZE (**1.0331**+ZE) / Stahl

If the first synthesized program extracts the first six characters, the result will obviously not work for all inputs. Thus, the synthesizer must be able to apply a different program depending on the input.

Chapter 3

Programming by Example

In this chapter, we describe the language L that can model different string operations and what approach we use to handle strings. We present a data structure to succinctly represent a large set of expressions in the language. With the language and the data structure we are able to represent all possible programs for one input output example.

Figure 3.1 shows the high-level overview of our PBE system and how the different components interact with each other and the user.

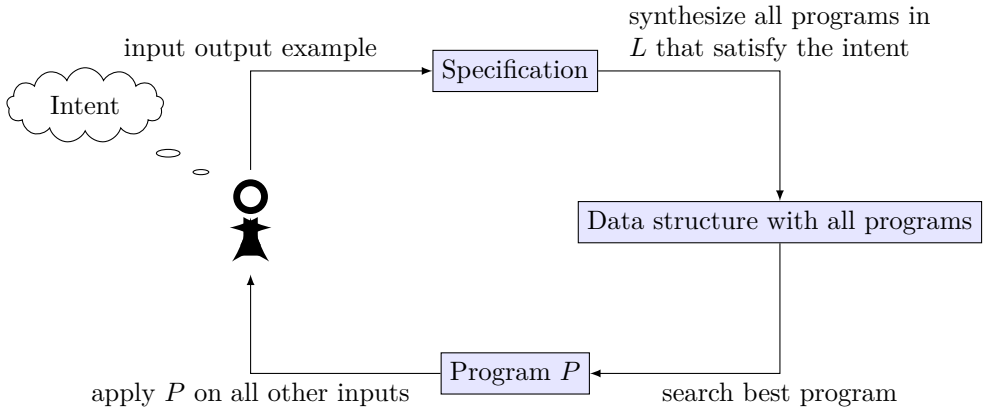
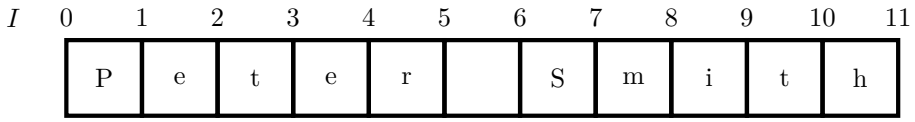


Figure 3.1: Overview of how our PBE system works.

Example 1. Given the following two input output pairs: ("Peter Smith" → "Peter") and ("John Doe" → "John"), we try to construct the output string from its corresponding input string.
Let's start with the output string of the first pair — "Peter". Next, we try to find substrings in the given input which are equal to *a part* of the output string. Figure 3.2 shows that there are many ways of using the input to get parts of the output. The most obvious one is to take the word "Peter" from the beginning of the input string. Another possibility is to only take a few letters, like "Pet" in the example. It is even possible to build the output string with single letters, where a letter can be used multiple times. It does not matter where the letter comes from, as seen by the letters "e" ant "t" in the example.



$$\begin{aligned}\text{“Peter”} &= \{I_{0-5}\} \\ &= \{I_{0-1}, I_{3-4}, I_{9-10}, I_{3-5}\} \\ &= \{I_{0-1}, I_{3-4}, I_{2-3}, I_{1-2}, I_{4-5}\} \\ &= \dots\end{aligned}$$

Figure 3.2: An output string can be assembled in numerous ways from its input string.

There are endless possibilities to represent the output string as a combination of substrings of the input string. The number of possibilities grows even more if the different kinds of definitions for one substring are taken into context, i.e. there are multiple ways to define the start and end index of a substring on a given string.

Example 2. We want to extract “Peter” with a single substring operation. We describe a list of possible programs to extract the intended substring from the input.

1. Substring from input string index 0 to 5
2. Substring the first word from the left
3. Substring the second word from the right
4. Substring from the start to the first space character

   ⋮

It is obvious that there are numerous programs that fulfill our intent *given only a single input output pair*. Many of these programs only work on the given example, i.e. they are too specific. To find more general programs, we create sets of possible programs for *two input output pairs*. By intersecting both sets we have a much smaller list of programs that fit both examples.

Example 3. Given the programs that satisfy the input output pair “Peter Smith” → “Peter”, many of these programs do not evaluate to the correct output for the pair “John Doe” → “John”. The program which takes the first 5 characters is not feasible on the second pair. Only the programs which select the first word (or the second to last etc.) remain in the set of possible programs.

Remark. Intersecting two sets of programs which share no programs results in an empty set. However, there are examples where we have a set of input outputs where we have to apply different programs. Thus, for certain cases we have to abandon the approach of intersecting sets of programs. A technique to solve this problem is called *conditional application* and is discussed in Section 4.4.

3.1 Domain-Specific Language

Substrings are only one possible operation applied to the input string to generate the output string. The set of combinations of these operations is called the *search space*, as these operations are the building blocks which are used to synthesize a program.

When developing a synthesizer we need to strike a good balance between expressiveness and efficiency of the search space of possible programs. On the one hand, the space of the programs

should be large/expressive enough to include programs that users care about. On the other hand, the space of the programs should be restrictive enough so that it allows for an efficient search, and it should be over a domain of programs that enables efficient reasoning [16].

To achieve this balance and to have means to fit the search space, we present a DSL (domain-specific language) L which only contains a simple set of string operations tailored to our needs. We describe a modified and simplified version of a DSL for string processing proposed by Gulwani [7]. The DSL contains a set of operations that are applied on the input string and allow us to model programs.

String structure To conduct complex string operations, e.g. getting all characters between two braces, we need to have insight into the structure of a given string. A *token* either represents some special character (e.g. '`(`', '`.`', '`#`', etc.) or a character class that matches a sequence of *one or more* characters (e.g. all alphanumeric characters).

We can choose any token, as long as it fulfills two properties. The chosen tokens are *disjunct* and need to have *at least one occurrence*. This allows efficient enumeration of all possible *SubStr* operations as there are no overlapping match ranges. Thus, we can describe the strings structure with only a single token combination.

For better readability, we reference tokens either by representative names or abbreviations. For example, we refer to the token matching all numeric characters as `Numeric` or `N`.

- `Start = ^`
- `End = $`
- `Alphabetic = A`
- `Numeric = N`
- `Space = S`
- `Leading Zero = 0`
- `Open Brace = OB`
- `Close Brace = CB`
- `Open angular bracket = OA`
- `Close angular bracket = CA`
- `Dot = .`
- `Slash = /`
- `Hyphen = -`
- `Everything Else = E`

We cannot add a token for each special character as there are hundreds of thousands of Unicode characters. Therefore, we use an `Else` token, that matches everything except all the other used tokens.

Special tokens — apart from the ones displayed in Figure 2.2 — are:

- **Start Token** Matches the beginning of the whole string i.e. that "eter" does not match this token, whereas "Pet" does.
- **End Token** Vice versa matches the end of the whole string.
- **Leading Zero Token** In our test set there are a number of examples where leading zeros have to be removed. This token can have conjunctions with the `Numeric` token, i.e. the matches overlap. Therefore, `Numeric` token must not match leading zeros!

Token Sequence A single token — for example `Numeric` — matches many substrings on a string. If we want to match a very specific substring a single token is too ambiguous. Thus, we introduce the concept of *token sequences* or short $TokenSeq$. A $TokenSeq$ is a sequence $r = \{T_1, .. , T_n\}$ of tokens $T_1, .. , T_n$. A $TokenSeq$ is constructed by merging all its tokens into a new token. The

merging of the tokens is done by concatenating the regex patterns of $T_1, .. , T_n$ to a *single new regular expression*.

Example 1. Given the two tokens T_1 := `Alphabetic` and T_2 := `Hyphen`, the *TokenSeq* $r = \{T_1, T_2, T_1\}$ and the string `"John-Doe - ABC.XYZ"`:

T_1 matches `"John-Doe - ABC.XYZ"`
T_2 matches `"John-Doe - ABC.XYZ"`

Whereas r only matches: “John-Doe - ABC.XYZ”

Representing Tokens Each token can be represented by a regular expression. This has the following benefits:

1. The `Else`-Token can be trivially represented by negating all other used tokens.
2. A token sequence can be dynamically built by concatenating the regex pattern of each token to a new pattern.
3. Getting the total number of matches of a *TokenSeq* on a string or the c-th position of a match is trivial. Conventional regex engines in language like Java, C# and others return all regex matches and their ranges on a string.

CPos operation $CPos(k)$ is a position expression and evaluates to the k^{th} index in a given string from the left side (or right side), if the constant k is non-negative (or negative). We denote the last index of a string with $CPos(-\infty)$.

$$[\![CPos(k)]\!]s = \begin{cases} |s|, & \text{if } k = -\infty \\ k, & k \geqslant 0 \\ |s| + k, & \text{otherwise} \end{cases} \tag{3.1}$$

Pos operation In many cases evaluating the same constant position is not sufficient to express the users intent. For example, we have a data set of first and last names and we want to extract the last name. As the start position of the substring depends on the length of the first name, we cannot use a constant position, i.e. CPos does not work. Thus, we have to dynamically evaluate the start position of the substring based on the structure of the input string.

In order to represent relative positions we use $Pos(r_1, r_2, c)$, where r_1 and r_2 are token sequences and c is a constant non-zero integer. Pos evaluates to an index t on a string s such that r_1 matches some *prefix* of $s[t_1 : t]$ and r_2 matches some *suffix* $s[t : t_2]$. Furthermore, t is the c^{th} such match from the left side (or the right side) if c is positive (or negative) — similar to $CPos$. If not enough matches exist, an exception is thrown.

Example 2. Given the string “Peter Smith” and the operation $Pos(r_1, r_2, c)$, where $r_1 = \{$`^`, `A`, `S`$\}$, $r_2 = \{$`A`$\}$ and $c = 1$. The first line in Figure 3.3 shows how the tokens r_1 and r_2 are matched if they are used independently. The second line shows how the index t is matched if r_1 is the *prefix* and r_2 is the *suffix*.

SubStr operation $SubStr(p_1, p_2)$ returns the substring of the input between the indexes p_1 and p_2. p is either a $CPos$ or a relative Pos. A $SubStr$ operation can be applied on every string, yet p_1 or p_2 *can evaluate to an illegal position* for some strings. For example, taking the substring from index 7 to 10 on the string `"ABC"` will not work, as the *index is out of bounds*. Thus, the SubStr operation can throw an exception.

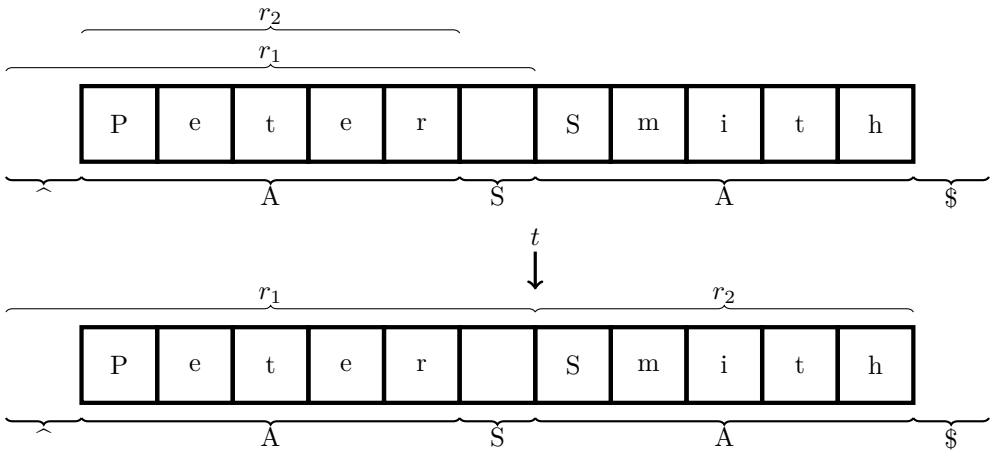


Figure 3.3: Visualize how ($[\![Pos(\{\hat{}, \texttt{A}, \texttt{S}\}, \{\texttt{A}\}, 1)]\!]$ "Peter Smith") evaluates to index t.

ConstStr operation $ConstStr(s)$ returns a constant string s, determined at the creation of the program. It is mostly used for delimiters, prefixes or suffixes which are not part of the input string.

Lookup operation $Lookup(tab, col_{in}, col_{out})$ is needed for the translation of known entities from existing, user-provided lookup tables. It consists of a *tab*, which denotes the table that was used as a lookup, and two col_{in} and col_{out}, which denote the input and output column.

The operation searches the input string for a value present in the input column. If a value is found the value from the output column at the same index is returned. If no value from the input column is found in the input string, an empty string will be returned.

Concatenating Operations In most cases a single operation (or expression) is not enough to build the output string from input string s. Therefore, we introduce *trace expressions* which are an ordered list of operations made from our language L. Each expression is applied on s and the results are concatenated to a single string.

$$[\![TraceExpression(e_1, \ldots, e_n)]\!]\ s = Concatenate([\![e_1]\!]\ s, \ldots, [\![e_n]\!]\ s)$$

Summary Now that we have defined each element in our DSL, we can build complete programs in our custom language L.

$$\begin{aligned}
\text{Constant String } s &:= ConstStr(s) \\
\text{Substring } &:= SubStr(p_1, p_2) \\
\text{Position } p &:= CPos(k) \mid Pos(r_1, r_2, c) \\
\text{Constant Position } &:= CPos(k) \\
\text{Relative Position } &:= Pos(r_1, r_2, c) \\
\text{Regular Expression } r &:= T_1, .. , T_n \\
\text{Token } &:= T \\
\text{Lookup } &:= Lookup(tab, col_{in}, col_{out}) \\
\text{Table Identifier } &:= tab \\
\text{Column Identifier } &:= col
\end{aligned} \tag{3.2}$$

Example 3. In this example we showcase how powerful our language is for string manipulations. Consider the following example:

Input	Output
March 2019	3. 2019
April 2019	4. 2019
June 2019	6. 2019

One possible trace expression to achieve this transformation looks like the following:

$$\begin{aligned} \text{Transform month name to number } e_1 &\equiv Lookup(Months, Name, Number) \\ \text{Dot at the end of month number } e_2 &\equiv ConstStr(.) \\ \text{Space and last word } e_3 &\equiv SubStr(Pos(\{\texttt{A}\}, \{\texttt{S}\}, -1), CPos(-\infty)) \\ TraceExpression &\equiv \{e_1, e_2, e_3\} \end{aligned}$$

3.2 Data Structure

To store the set of possible programs efficiently we use a DAG (directed acyclic graph). Each node represents the index between two characters of the output string. Therefore, each edge represents a substring of the output string, as seen in Figure 3.4. In this example the output string is again "Peter", which leads to node indices from 0 to 5.

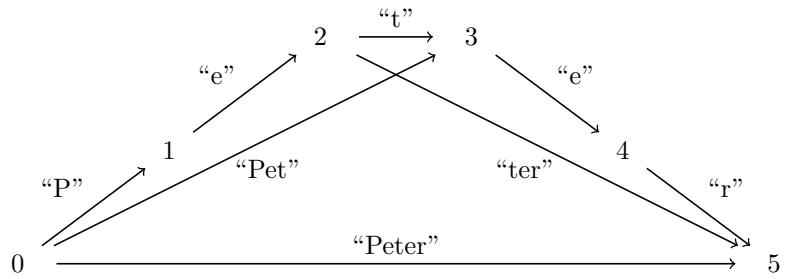


Figure 3.4: Directed acyclic graph with a sample of edges where each edge represents a substring of the output string.

Instead of the substring of the output string, the edges are populated with expressions from our DSL that evaluate to the respective substring. How the algorithm populates this graph is shown in Section 4.1. To enumerate all the programs that transform the input string to the output string, all possible combinations of DSL expressions on a path from 0 to 5 can be concatenated.

Possible Programs $CPos(k)$ is simplified to k

- $0 \rightarrow 5$
  - $SubStr(0, 5)$
  - $ConstStr(\text{"Peter"})$
- $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$
  - $SubStr(0, 1) + SubStr(1, 2) + \ldots + SubStr(4, 5)$

- $ConstStr(\text{“P”}) + SubStr(1, 2) + \ldots + SubStr(4, 5)$
  $\vdots$
- $ConstStr(\text{“P”}) + ConstStr(\text{“e”}) + \ldots + ConstStr(\text{“r”})$

The usage of DAG allows us to store the operations on the edges without the need to enumerate all possible trace expressions preemptively. This is useful when modifying the graph after its initial creation. Furthermore, this data structure allows us to intersect the common programs of two DAGs.

Chapter 4

Synthesis Algorithm

In this chapter we describe precisely how a program is synthesized. First, we show how to build a DAG containing the information to generate a program for one input output pair. Then we describe how to efficiently intersect two DAGs to find programs that match two input output pairs. After that, we show how to enumerate and rank the possible programs to find the best fitting one. We specify a mechanism to distinguish between different inputs and apply different programs depending on learned conditions. Lastly, we show how limiting the set of operations can optimize the algorithm. The core concept of the algorithm is derived from Gulwani [7].

4.1 Building Possible Programs

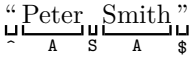
How to generate all possible programs for an example is a crucial part of the PBE system. Given an input output pair we start by looping over each substring s of the output string. Each s represents the intended value that must be returned by the operations on the edge in the DAG. This means that if we enumerate all possible expressions for each of these substrings we have a complete DAG. For each edge we now start to search for expressions in our language L which evaluate to the given substring.

Finding ConstStr expressions for s We add the expression $ConstStr(s)$ to our set of expressions on the edge.

Finding SubStr expressions for s First, we search the input string for occurrences of s. For each occurrence we denote the index where the occurrence starts with k. We now know that there is a $SubStr(CPos(k), CPos(k + |s|))$ that evaluates to s.

To find the equivalent relative positions Pos, we enumerate all possible sequences of tokens from the position to the left until the start of the input string. If we do the same thing on the right of the position until the end of the input string, we have two sets of regular expressions and each combination leads to a valid Pos. Before creating the Pos, the c value must be derived, as it denotes the c-th occurrence of the patterns in the input string.

Example 1. Given the string "Peter Smith" and the index between "Peter" and the space character, there are 12 ways to describe this position. The string can be represented as the following sequence of tokens: Start (`^`), alphabetic (`A`), space (`S`), alphabetic (`A`), end (`$`).



On the left side of the index the token sequence is either `A` or `^A`. On the right of the index the token sequences are `S`, `SA` or `SA$`. The list of all possible positions is the Cartesian product of the left and the right side. In this case all combinations of token sequences only occur once. Therefore, the possible c values are either 1 or -1.

Possible representations of Pos

- $Pos(\texttt{A}, \texttt{S}, 1)$
- $Pos(\texttt{A}, \texttt{S}, -1)$
- $Pos(\texttt{\^{}A}, \texttt{S}, 1)$
- $Pos(\texttt{\^{}A}, \texttt{S}, -1)$
- $Pos(\texttt{A}, \texttt{SA}, 1)$
- $Pos(\texttt{A}, \texttt{SA}, -1)$
- $Pos(\texttt{\^{}A}, \texttt{SA}, 1)$
- $Pos(\texttt{\^{}A}, \texttt{SA}, -1)$
- $Pos(\texttt{A}, \texttt{SA\$}, 1)$
- $Pos(\texttt{A}, \texttt{SA\$}, -1)$
- $Pos(\texttt{\^{}A}, \texttt{SA\$}, 1)$
- $Pos(\texttt{\^{}A}, \texttt{SA\$}, -1)$

All the combinations of the Pos generated with k and $k + |s|$ can be used to create a valid $SubStr$.

Algorithm 4.1 and Algorithm 4.2 specify how the $SubStr$ expressions are generated in more detail.

Algorithm 4.1 Generate $SubStr$ expressions

```
1: function GenerateSubStr(s: Substring of output, input: Input string)
2:     subStrings ← {}
3:     for each k s.t. k is the start of substring s in input do
4:         y1 ← GeneratePositions(k, input)
5:         y2 ← GeneratePositions(k + |s|, input)
6:         for each pos_l in y1 do
7:             for each pos_r in y2 do
8:                 subStrings := subStrings ⋃ {SubStr(pos_l, pos_r)}
9:             end for
10:        end for
11:    end for
12:    return subStrings
13: end function
```

Algorithm 4.2 Generate Pos and $CPos$ expressions

```
1: function GeneratePositions(i: index of position, input: Input string)
2:     allPos = {CPos(i), CPos(−(|input| − i))}          ▷ Add constant positions
3:     for each TokSeq_l s.t. TokSeq_l is a Token Sequence from k_1 to i for any k_1 < i do
4:         for each TokSeq_r s.t. TokSeq_r is a Token Sequence from i to k_2 for any k_2 > i do
5:             TokSeq_lr ← TokSeq_l ++ TokSeq_r          ▷ Concatenate the sequences
6:             c ← i is the c-th occurrence of TokSeq_lr in input
7:             c_Max ← Number of occurrences of TokSeq_lr in input
8:             allPos := allPos ⋂ {Pos(TokSeq_l, TokSeq_r, c)}
9:             allPos := allPos ⋂ {Pos(TokSeq_l, TokSeq_r, −(c_Max − c))}
10:        end for
11:    end for
12:    return allPos
13: end function
```

Finding Lookup expressions for s The string s is compared to all values in all lookup tables. If the value is found the values of all columns of the corresponding row are searched in the input string. If one of this values is found, a new *Lookup* expression with the references to the table and columns is added to the set of expression for the edge.

4.2 Intersecting Programs

To find a program that satisfies two input output examples we intersect the DAGs of both pairs. To do this we first build the Cartesian product of the nodes of both graphs. This means we create a new node for each combination of the nodes from the two graphs. The set of possible operations on an edge is determined by the intersection of the two sets of the corresponding edges in the old graph. For example the edge (1/2) → (4/5) would have the intersections of the sets of (1) → (4) and (2) → (5).

Figure 4.1 shows two DAGs with their edges populated with expressions. The DSL expressions are simplified to a set of single uppercase letters. In this example we intersect only letters that are the same. The intersection of "AB" and "BC" would therefore result in "B".

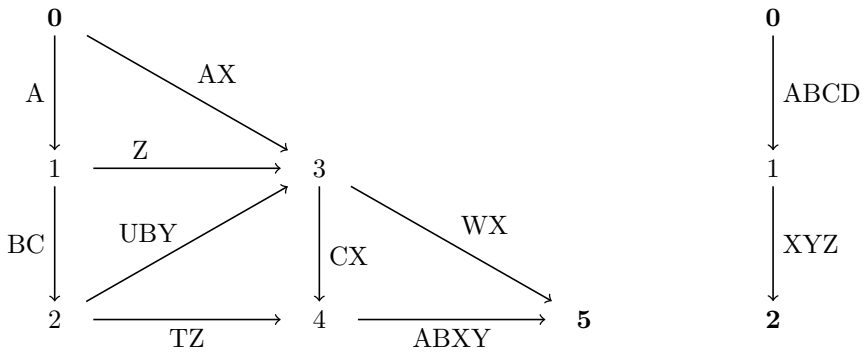


Figure 4.1: Two graphs before intersection. Source and target nodes are bold.

The intersection of two DAGs is built by calculating their Cartesian product. The resulting graph looks like a matrix with the first graph acting as rows and the second graph acting as columns. Figure 4.2 shows the intersected graph of the two graphs in Figure 4.1 — the source (0/0) and target (5/2) nodes are bold.

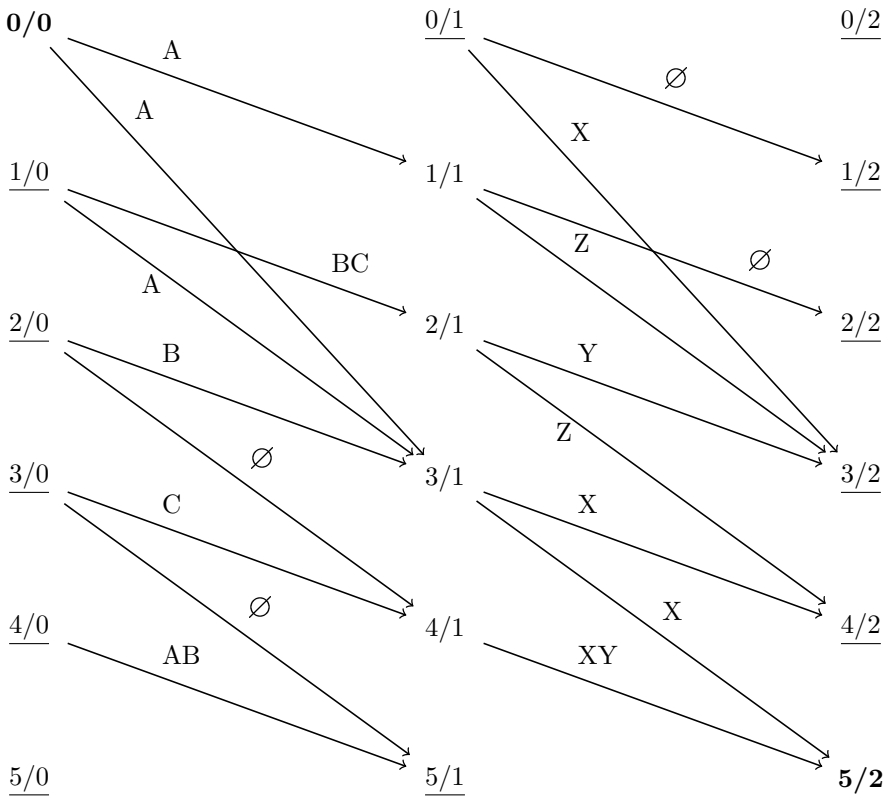


Figure 4.2: Full graph after building the Cartesian product of two graphs and intersecting the edges. The nodes are arranged in a matrix where one graph represents the rows and the other the columns.

The problem with this new graph is that its node size is the product of the sizes of the two original graphs. This does not scale very well, since the number of nodes grows quadratically. Therefore, we need to optimize the result of the intersection.

Because all the graphs are acyclic, there are no edges in horizontal or vertical direction. As we are only interested in paths that lead from source to target this characteristic allows us to discard some nodes. No node on the "border" of the matrix (underlined) can lead from source to target, except for the *source* and *target* nodes itself. We can discard all these nodes and the edges connected to it. Furthermore we can discard all edges that contain an *empty set* of expressions, as they cannot be used to build a program. The removed elements are grayed out in Figure 4.3

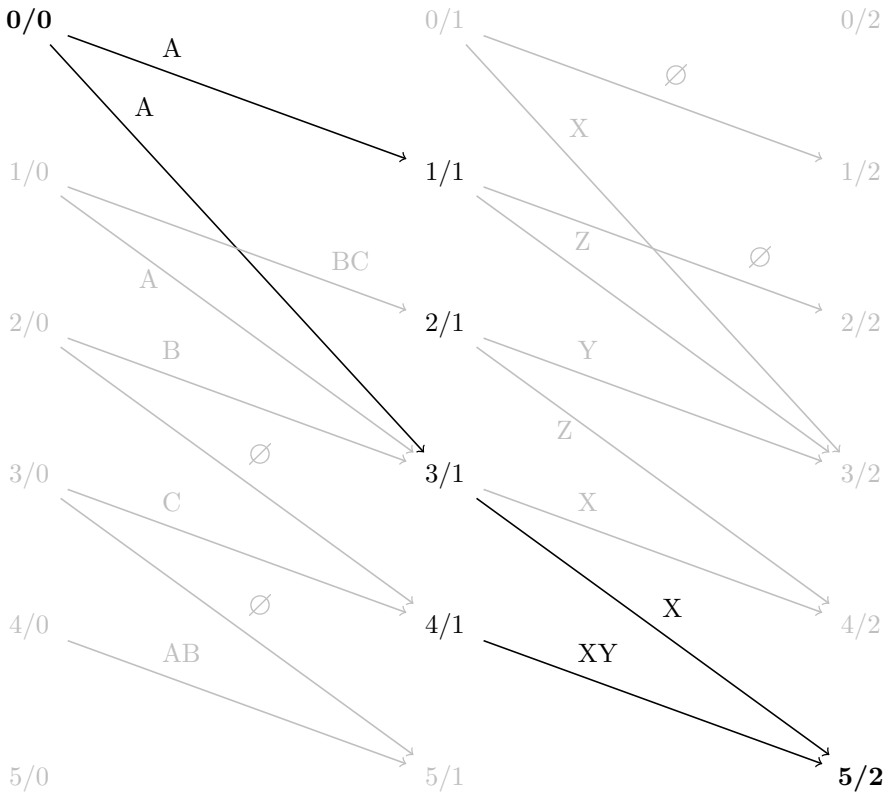


Figure 4.3: Graph with border nodes and empty edges removed

The second step in optimizing the size of the graph is to remove all nodes with only *ingoing* or *outgoing* edges, except for source and target. These nodes, and their connecting edges, cannot be part of the path. After the nodes (1/1) and (4/1) are removed from the example, the graph is reindexed to its minimal size. The end result of this process is shown in Figure 4.4. The only possible program in this example would be "AX".

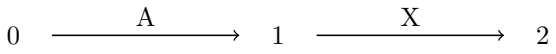


Figure 4.4: End result of the intersection of two DAGs.

The allowed intersections of our DSL elements are shown in Equation (4.1). All other intersections return $\varnothing$.

$$\begin{aligned}
SubStr(p_1, p_2) \cap SubStr(p_1', p_2') &= SubStr(p_1, p_2) \text{ iff } p_1 = p_1' \wedge p_2 = p_2' \\
CPos(k) \cap CPos(k') &= CPos(k) \text{ iff } k = k' \\
Pos(r_1, r_2, c) \cap Pos(r_1', r_2', c') &= Pos(r_1, r_2, c) \text{ iff } r_1 = r_1' \wedge r_2 = r_2' \wedge c = c' \\
r \cap r' &= r \text{ iff } T_1 = T_1' \wedge T_2 = T_2' \wedge \ldots \wedge T_n = T_n' \\
ConstStr(s) \cap ConstStr(s') &= ConstStr(s) \text{ iff } s = s' \\
Lookup(tab, col_{in}, col_{out}) \cap Lookup(tab', col_{in}', col_{out}') &= Lookup(tab, col_{in}, col_{out}) \\
&\quad \text{iff } tab = tab' \wedge col_{in} = col_{in}' \wedge col_{out} = col_{out}'
\end{aligned} \tag{4.1}$$

4.3 Finding and Ranking Programs

As mentioned before, all combinations of expressions on paths that lead from source to target are programs which fulfill the intent set by the examples. The most simple solution would be to take just the first program. This has the drawback that we cannot prefer more generic programs over specific programs. Therefore, we have to rank the programs according to some metric. Each DSL operation has a cost which is used to find simpler programs that are not unnecessarily complex. However, computing the cost of each trace expression over each path is computationally expensive. This is due to the fact that the combination of all expressions grows exponentially with the length of the path.

We use a two-step approach to find an adequate program in reasonable time. First, we find the cheapest path to enumerate its programs and then we choose the program with the lowest cost from this set.

Finding the cheapest path to enumerate is a *shortest path problem* where the weight of an edge is the number of expressions. The weight of a path is the product of all weights of its edges, as this represents the number of possible programs on this path. This heuristic allows us to find a set of operations that is cheap to enumerate but also contains many promising programs. As the graph is acyclic and already sorted in topological order of the indices, finding the shortest path is trivial and can be done in linear time.

For the cheapest path we now enumerate all possible programs and calculate the cost according to Equation (4.2). The *ConstStr* and *CPos* operations are heavily penalized such that a more generic program is preferred. The main reason is that a program with only one pair of examples should use a *SubStr* operation with relative *Pos* whenever possible, as the chance of a *ConstStr* operation to fit the next example pair is relatively low. On the other hand the *Lookup* operation has a negative cost to incentivize the synthesizer to use the additional information given by the lookup tables.

$$\begin{aligned} Cost(SubStr(p_1, p_2)) &= Cost(p_1) * Cost(p_2) \\ Cost(CPos(k)) &= 15 \\ Cost(Pos(r_1, r_2, c)) &= Cost(r_1) * Cost(r_2) \\ Cost(r) &= |T_1, T_2, \ldots, T_n| \\ Cost(ConstStr(s)) &= 250 \\ Cost(Lookup(tab, col_{in}, col_{out})) &= -10 \end{aligned} \tag{4.2}$$

4.4 Conditional Application

The problem with the solution so far is, that there is no valid program when the intersection of two DAGs results in an empty DAG. To combat this, we partition the DAGs such that each partition yields a program when all its DAGs are intersected. Then we have to find a classifier, which when applied to a new input, tells us the partition from which the program should be taken.

To partition the programs we choose a simple iterative approach. The DAG of the first input output pair builds the first partition. If the DAG from the second input output pair yields at least one program when intersected with the first DAG, it is added to the partition. Otherwise, we create a new partition with the second DAG. We continue to add new input output pairs to the first partition and create new partitions when the intersection does not result in a program that satisfies the requirements. If a new input cannot be classified it is discarded.

To find a classifier that fits exactly one partition we use a *one versus all* approach for each partition. This means that we search a classifier that matches all inputs from the given partition and none

of the inputs from all other partitions. The classifier itself is in the *disjunctive normal form*. Each conjunction in the classifier matches some of the inputs from the partition and none of the inputs from all other partitions. This represents the algorithm which we use to find the classifiers as well. While there are still inputs which are not covered by one of the conjunctions we search for a set of predicates which match for some of the not yet classified inputs and none of the inputs we want to exclude. As predicates we use the $Match(r, c)$ function and its negation. This allows us to classify the inputs according to the $TokenSeq$ r and the number of its occurrences c. The set of possible predicates are all $Match(r, c)$ that match any of the input. Algorithm 4.3 and Algorithm 4.4 show the algorithm to find a classifier for a partition.

Algorithm 4.3 Generate Classifier

```
1: function GenerateClassifier(includes: input strings to include, excludes: input strings
   to exclude)
2:     classifier ← {}
3:     includesToClassify ← includes
4:     while |includesToClassify| > 0 do                    ▷ Loop for conjunct expression
5:         excludesToClassify ← excludes
6:         classifyingInputs ← includesToClassify
7:         oldIncludesToClassify ← includesToClassify
8:         conjunct ← {}
9:         while |excludesToClassify| > 0 do                ▷ Loop for disjunct expression
10:            oldExcludesToClassify ← excludesToClassify
11:            predicate ← GetPredicate(classifyingIncludes, excludesToClassify)
12:            if predicate = {} then
13:                return {}                                ▷ No classification possible
14:            end if
15:            conjunct := conjunct ⋃ {predicate}
16:            let classifyingInputs be s.t. only values not matching the predicates remain.
17:            let excludesToClassify be s.t. only values not matching the predicates remain.
18:            if oldExcludesToClassify = excludesToClassify then
19:                return {}                                ▷ No classification possible
20:            end if
21:        end while
22:        if oldIncludesToClassify = includesToClassify then
23:            return {}                                    ▷ No classification possible
24:        end if
25:        classifier := classifier ⋃ {conjunct}
26:    end while
27:    return classifier
28: end function
```

Algorithm 4.4 Get the predicate with the biggest impact on the partitioning

```
1: function GetPredicate(includes: input strings to include, excludes: input strings to exclude)
2:     bestScore ← −1
3:     bestMatch ← {}
4:     for each match s.t. match is a match(r, c) in includes or excludes do
5:         predicate := match                                   ▷ Positive match
6:         let matchInclude be the number of includes that predicate matches.
7:         let notMatchExclude be the number of excludes that predicate does not match.
8:         score := matchInclude * notMatchExclude
9:         if score > bestScore then
10:            bestScore := score
11:            bestMatch := predicate
12:        end if
13:        predicate := ¬match                                  ▷ Negative match
14:        let matchInclude be the number of includes that predicate matches.
15:        let notMatchExclude be the number of excludes that predicate does not match.
16:        score := matchInclude * notMatchExclude
17:        if score > bestScore then
18:            bestScore := score
19:            bestMatch := predicate
20:        end if
21:    end for
22:    return bestMatch
23: end function
```

4.5 Synthesizer

In this section we present how all components interact with each other. The public interface of our program synthesizer consists of a method `Synthesize` that takes an input output example and returns a program p which contains a list of partitions. Each partition contains a classifier to determine if a string matches the partition (see Section 4.4) and a trace expression to transform the input. Thus, when p is applied on an input string a matching partition is searched and its trace expression applied on the string. If none of the partitions in p match the input, we return the input.

Listing 4.1 shows the structure of a sample program. The program can be read like a "if-elseif" statement in conventional programming languages. For example, given two input string "0078225.1.09" and "1004425.A.00" the first partition will match the first string and return "78225", because the expression extracts everything after the leading zeros until the first dot. Whereas the second string will match the second partition and return "1004425", because the expression extracts everything from the start to the first dot.

```
1 Program[
2   CASE( ((P, 1, {^, 0+, N-0}) )
3     TraceExpr(SubStr(Pos(TokenSeq(^, 0+), TokenSeq(N-0), SimpleInt(1)), Pos( ↵
      TokenSeq(N-0), TokenSeq(.), SimpleInt(1))))‘,
4   CASE( ((P, 1, {^, N-0, .})) )
5     TraceExpr(SubStr(Pos(TokenSeq(^), TokenSeq(^, N-0), SimpleInt(1)), Pos( ↵
      TokenSeq(N-0), TokenSeq(.), SimpleInt(1))))‘
```

```
6 ]
```

Listing 4.1: Sample program structure with two partitions.

`Synthesize` works by first building the DAG g from the input output example (see Section 4.1). The first time `Synthesize` is called a new program is created with a single partition. This first partition is initialized with a default classifier that matches every string. Note that applying a trace expression on a string might fail. The default classifier is changed once a second partition is added.

If the first program already exists, we iterate over its partitions and intersect g with the DAG g_{pi} from the partition. If the result is empty, we continue with the next partition, i.e. g cannot be part of the current partition. Else we update the current partition by setting the g_{pi} to the intersected DAG, finding and setting a new trace expression from the intersected DAG (see Section 4.3) and adding the input to the partitions list of inputs.

If none of the partitions match the new DAG, we add a new partition to p. We use the `GenerateClassifier` to generate a classifier that matches none of the other classifiers (see Algorithm 4.3). Note that adding a new partition might fail, because `GenerateClassifier` can return an empty set. In that case we simply discard the new input output example.

4.6 Optimization

In this section we describe what optimizations we made in the implementation of the algorithm.

According to our algorithm we create the *Pos* elements for all sequences of tokens left or right of the position. We analyzed the token sequence lengths of the programs synthesized during the benchmark and all the token sequences generated while populating the DAG. The final program only used token sequences with 1 to 4 tokens. The initial DAGs had tokens sequences with a length of up to 39. Figure 4.5 shows the distribution of both lengths.

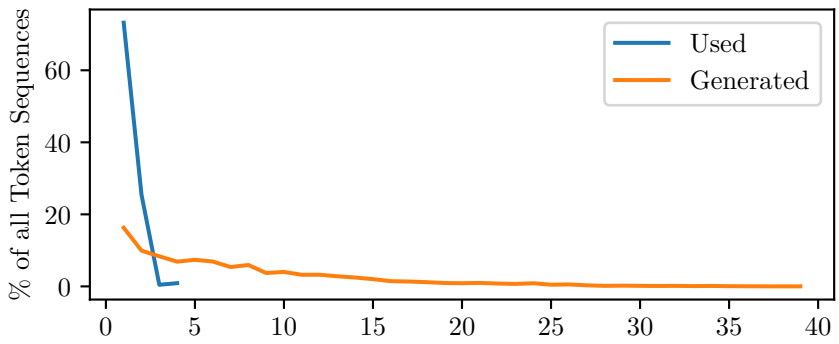


Figure 4.5: Distribution of the length of token sequences

The results in Figure 4.5 show that it is clearly inefficient to generate all token sequences. Thus, an upper limit is needed. Furthermore, a similar problem occurs when building the predicates for the conditional application. These are token sequences as well. However, we use a scoring system to find the best predicate, shown in Section 4.4. Therefore, we do not need an upper limit of the length but a preferred length of the token sequence. This only applies if two predicates are equally effective, in this case the predicate with the length closer to the preferred length is taken.

A third variable in the decision for the token lengths is if the character and number tokens should be split or if an alphanumeric token should be used. We evaluate this variables according to a

benchmark score. How this score is built exactly is shown in Section 5.1. The scores of the PBE system for combined token and for split tokens are shown in Table 4.1.

		Max token length									
		1		**2**		**3**		**4**		**5**	
		w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Preferred predicate length	**1**	98.40	98.59	98.75	98.73	98.75	98.37	98.75	98.73	98.75	98.73
	2	98.60	98.63	98.79	98.90	98.79	97.94	98.79	98.90	98.79	98.90
	3	98.58	98.60	**98.80**	**98.94**	**98.80**	98.31	**98.80**	**98.94**	**98.80**	**98.94**
	4	98.58	98.60	**98.80**	**98.94**	**98.80**	**98.94**	**98.80**	**98.94**	**98.80**	**98.94**
	5	98.58	98.60	**98.80**	**98.94**	**98.80**	**98.94**	**98.80**	**98.94**	**98.80**	**98.94**

Table 4.1: Score without (with) distinction of characters and numbers left (right)

As seen in Table 4.1, the score converges in the lower right corner of both tables. This shows that there is no improvement possible by increasing the length further. We see that the best score is achieved by using the character and number token separately. As each of the lower right corner result is the same score, we use the lowest possible token length because that means fewer calculations are needed. The final algorithm uses a maximal token length of 2 and a preferred predicate length of 3.

Chapter 5

Results and Discussion

A PBE system can be measured in different dimensions depending on the objectives it should solve. We show how we decided to quantify our progress. Furthermore, we compare our system to a baseline score. The test set consists of real world data from Shouldcosting and generated data for the known-entity-translation (see Section 2.1).

5.1 Metrics

In this chapter we discuss different possible metrics to score PBE systems and present the algorithm used in the baseline and in our PBE system.

To measure and compare PBE systems we present the following three approaches:

- **Synthesis Time** How long it takes to synthesize a program.
- **Program Complexity** Has influence on the execution time of a program on an input.
- **Number of Interactions** The fewer examples needed for the synthesizer, the better.

Although these metrics are different they are non-conflicting, i.e. one can use all metrics simultaneously. However, we only use the number of interactive rounds to show the results of our PBE system (Section 5.2). There are two reasons. Firstly, it is simple to understand for people who are unfamiliar with program synthesis. Secondly, we compare the metric score with the score of the baseline algorithm to determine the success of our work. As the baseline algorithm always runs in constant time and the generated program is simply a key value dictionary, measuring synthesis time and program complexity does not make sense.

Baseline The most simple baseline algorithm — that still can be considered PBE — is to simply store each input output example in a key value map. Given an input string, its corresponding output is returned if the value is present in the map, else the correct output value has to be provided and the rule count is increased.
Remark. The test set contains a handful of inputs that are ambiguous, i.e. map to multiple outputs. In this case, the entry in the key value map is overwritten.

Unsurprisingly this approach yields a decent score if the data set contains a lot of duplication (Section 5.2).

Algorithm 5.1 Baseline Benchmark

Input: S: Set of (σ, s) pairs

$r := 0$ ▷ Number of rules
$M : K \to V^{\perp} \mid M[k \to v](x) = v$ for $x = k$, otherwise $M(x)$ ▷ Map or Dictionary
for each (σ, s) in S **do**
  if $M(\sigma) = \perp$ **then**
    $M[\sigma \to s] = s$ ▷ Add new input output example
    $r := r + 1$
  end if
end for
return $1 \,/\, |S| \cdot r$ ▷ Calculate benchmark score

Reference The benchmark of our PBE system is more complicated for two reasons.

Firstly, the order of the input output examples set has an impact on the benchmark score. The benchmark feeds the first example from the set to the synthesizer and applies it on the following inputs — similar to the integrated interaction model described in Section 1.3. Therefore, different programs are synthesized depending on which examples are used. Table 5.1 illustrates this issue with some example input output pairs. We solve this problem by creating 5 randomly shuffled versions of each test set.

Input	**Output**	**Result**	
1995_24_12	1995	1995	$\Rightarrow p_1 \equiv$ `SubStr(CPos(0), CPos(5))`
Bill Davis	Bill	Bill	
Dori Jones	Dori	Dori	
Alexander Cook	Alexander	Alex	$\Rightarrow p_2 \equiv \ldots$

Input	**Output**	**Result**	
Alexander Cook	Alexander	Alexander	$\Rightarrow \tilde{p_1} \equiv$ `SubStr(CPos(0), CPos(9))`
1995_24_12	1995	1995_24_1	$\Rightarrow \tilde{p_2} \equiv \tilde{p_1} \vee$ `SubStr(CPos(0), Pos(...))`
Bill Davis	Bill	Bill Davi	$\Rightarrow \tilde{p_3} \equiv \ldots$
Dori Jones	Dori	Dori	

Table 5.1: Depending on the order of input output pairs the generated programs are different.

Secondly, the baseline is always able to synthesize a program in a constant time. Our synthesizer might not be able to generate a program for a given example or synthesizing takes longer than acceptable for an interactive, user-facing application. Therefore, we introduce time out and exception handling in the reference benchmark (see Algorithm 5.2). This time out can be used to limit the time the synthesizer has to find a program.

Algorithm 5.2 Reference Benchmark

Input: S: Set of (σ, s) pairs

$r := 0$ ▷ Number of rules
$f := 0$ ▷ Number of fails
$p := \{\}$ ▷ Synthesized program
for each (σ, s) in S **do**
  if $p = \{\}$ **then**
    $p :=$ SynthesizeProgram(σ, *input*, p)
    if $p = \{\}$ **then continue** ▷ Continue loop iff no program is generated
  end if
  $\widetilde{s} := p(\sigma)$ ▷ Apply the current program on the input
  if $\widetilde{s} \neq s$ **then** ▷ Wrong output i.e. try to extend p
    $p :=$ SynthesizeProgram(σ, *input*, p) ▷ Extend current program
  end if
end for
return $1 - (r + f) \;/\; |S|$ ▷ Score

Algorithm 5.3 Helper function to synthesize program

function SynthesizeProgram(σ: Input, s: Output, p: Program) ⇒ Program
  try
    $\widetilde{p} :=$ Synthesize(σ, s) ▷ Call the stateful synthesizer
  catch TimeoutException ▷ Synthesizer can timeout
    $f := f + 1$
    return p ▷ Failed to synthesize a program
  end try
  $r := r + 1$ ▷ Increase rules count
  return $\widetilde{p}$ ▷ Return newly synthesized program
end function

5.2 Results

The best score the baseline benchmark achieved was 97.02%. This is due to the fact that the test sets show very little variance. Our PBE system however scored **98.94%**. This means that the user only needs to manually process 1.06% of the inputs. All other outputs can be processed by the PBE system. Figure 5.1 shows the score compared to the baseline for each test set.

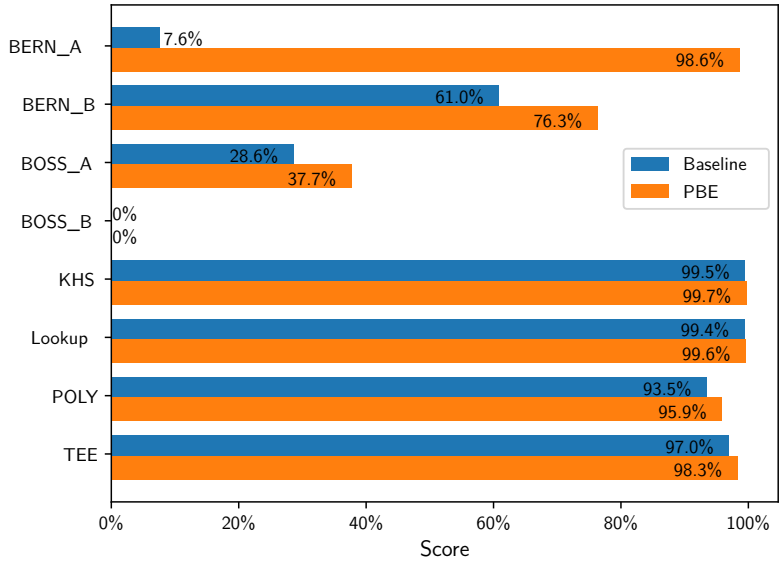


Figure 5.1: Benchmark results of the PBE system and the reference algorithm.

The `BERN_A` test set demonstrates why our PBE system outperforms the baseline. This set has almost no duplicate entries but all entries have the same pattern. We only need two examples to process the whole test set. One takes the string from start to the first dot. The other does the same but removes leading zeros from the string. The following examples represent the test set adequately:

- 1010325.B.00 → 1010325
- 1014835.A.00 → 1014835
- 0049205.0.00 → 49205
- 0077675.1.00 → 77675

The BOSS_B test set represents the cases where neither the baseline nor the PBE system achieve any advancement compared to processing each entry by hand. The set contains a use case where the user needs contextual information and the output cannot be deduced from the input. The contextual information cannot be taken from lookups as the outputs differ for the same inputs.

- AlMgSi → gehärtet
- AlMgSi → trovalisiert

Aside from the number of interactions we dissected the synthesis time. This is not a deciding factor for the optimization of the algorithm but only a conclusive analysis. Figure 5.2 shows the distribution of the synthesis time on all test sets. The results are measured three times (Intel Core i7-7700HQ, 32 GB RAM) and are averaged. The synthesis time refers to the time it takes to process one single input output pair. This includes the generation of the DAG, the intersection with the existing partitions and the classification of the input value.

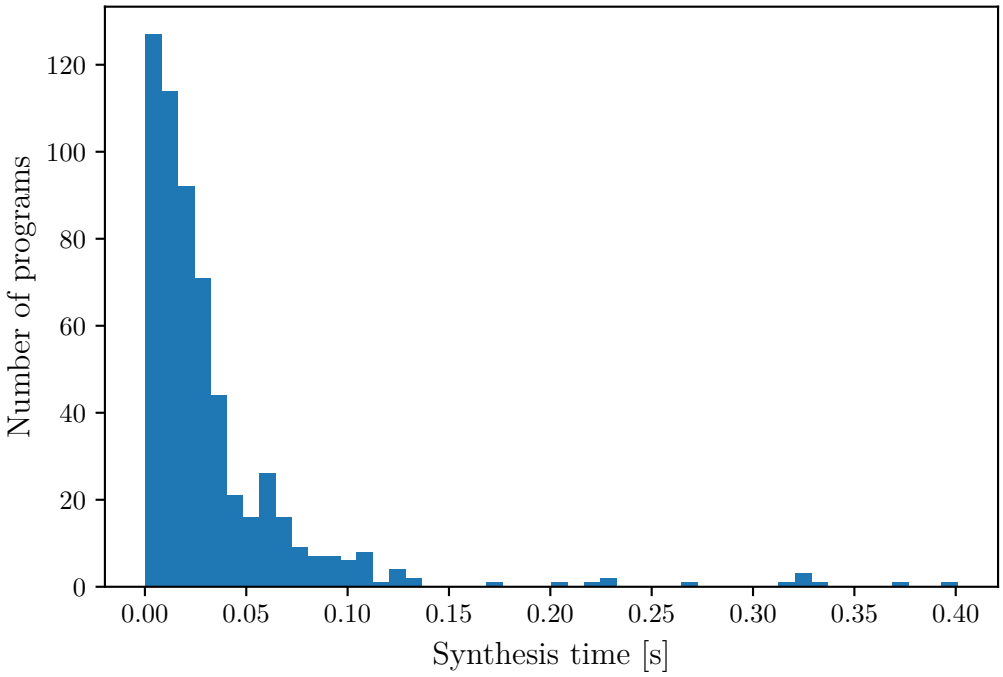


Figure 5.2: Histogram of synthesis time, $N = 584$

In Figure 5.3 the synthesis time is broken down into the different test sets. The graph shows the percentage of all the programs per test set that were synthesized in less time than the respective time on the x-axis.

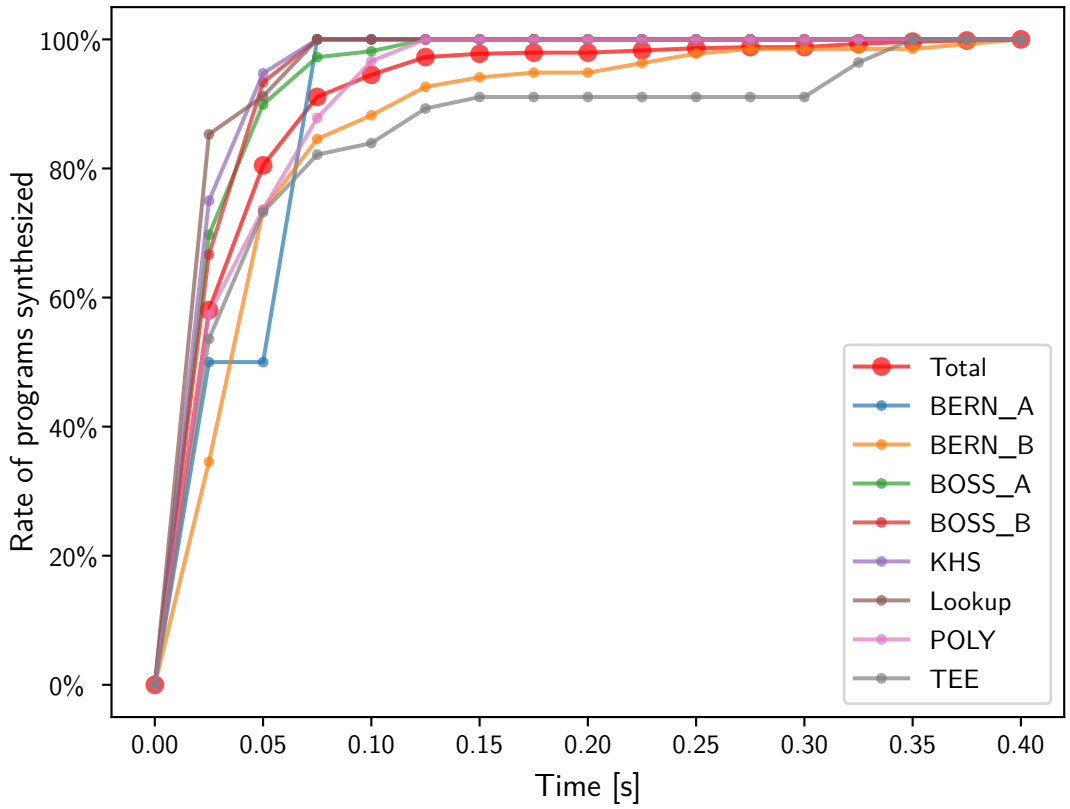


Figure 5.3: Synthesis time per test set

Figure 5.3 and Figure 5.2 shows that the synthesis time never exceeds 400 ms. The synthesis time is an important feature of a PBE system, as the maximal synthesis time for most real-life scenarios should not exceed 500 ms [22]. We use this 500 ms timeout restriction to compare our PBE system to the state of the art synthesizer: Microsoft PROSE. Figure 5.4 shows that our PBE system performs almost as good or even better than PROSE in all use cases. Considering time and resource disparity remarkably well in the two synthesizers, our system performs remarkably well.

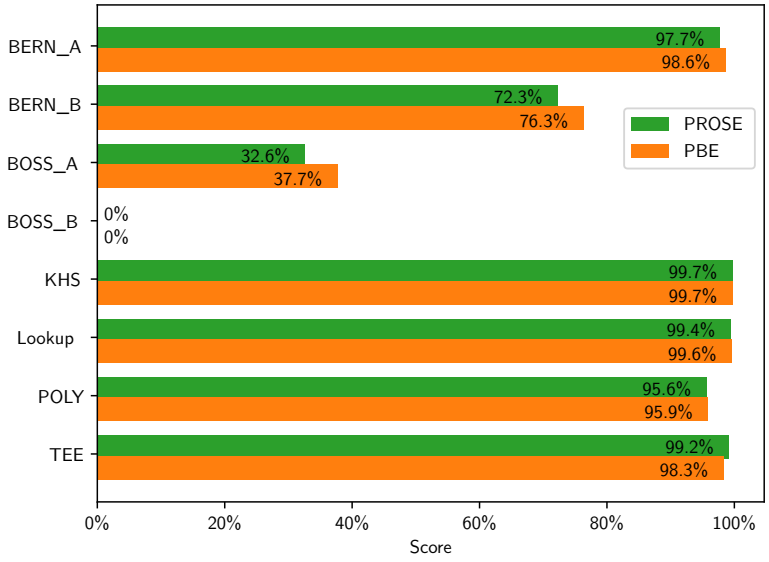


Figure 5.4: Benchmark results of our PBE system and the Microsoft PROSE.

Chapter 6

Conclusion

The PBE system was integrated into the data wrangling tool (DIC) of the customer. This enables string processing and known-entity-translations trough input output examples.

First, we built a prototype PBE system which could only extract simple substrings. Then we extended the system to handle more complex string operations and known-entity-translations. Lastly, we wrapped the system into a decision tree which allowed conditional application of the generated programs.

In the following sections we show the contributions of our PBE system and how it could be extended.

6.1 Contribution

With the PBE system we built we not only improved the data wrangling workflow of our customer but also obtained insight on how a PBE system can be built and adapted to given business needs. Our PBE system achieved a score of 98.94% whereas the baseline only scored 97.02%. Furthermore, a state of the art system scored 98.76% on our benchmark data set.

Simple string transformation We show a comprehensible algorithm for string transformation. This algorithm cannot only be recreated but also easily extended.

Business aware We describe how to tailor a PBE system to the business need of the customer. This knowledge of the problem domain allows for more specific, and in a given use case better, solutions than a general purpose PBE system.

Realizable Our results show that it is possible to build a conclusive PBE system in limited time. This is an important information for businesses, as many of the alternative PBE systems are not commercially available.

6.2 Outlook

There are many approaches to further extend the PBE system described in this thesis. The most promising in terms of accuracy or possible applications are listed below.

Ambiguity handling Currently we synthesize multiple programs that satisfy the user intent, yet — as seen in Section 4.3 — we only take best program. This is somewhat wasteful, since we never use the other programs. However, we can leverage them to enable easier interaction. The synthesizer can run a set of synthesized programs on each new input to generate a set of corresponding outputs for that input. The synthesizer can highlight the inputs that cause multiple distinct outputs for the user to take action.

Overlapping Tokens In our algorithm a string is represented by a single token sequence. This is very efficient during the building of the DAG. An alternative to this is using tokens which are not distinctive. This means that a character can be expressed by more than one token. A prominent example of this is using alphanumeric, character and number tokens. This allows more generic programs if needed but at the same time enables the specific distinction between characters and numbers.

Multiple Inputs Instead of just using one input per output, it is possible to use more than one input value. In this case, the *SubStr* and *Lookup* operation need an additional parameter which denotes the used input. This allows the combination of values from different inputs.

Extending DSL The PBE system is easily extendable for further DSL elements. New elements can be added to the set of possible operations on an edge of the DAG, similar to the *Lookup* operation. A possible new DSL operation can be a date translation which would transform a date to its corresponding week day.

SyGuS-Comp Syntax-Guided Synthesis Competition (SyGuS-Comp) is an annual program synthesis competition which allows solvers for syntax-guided synthesis problems to compete on a large collection of benchmarks [23]. The SyGuS-comp consists of four tracks: general track, conditional linear integer arithmetic track, invariant synthesis track, and — since 2016 — the programming by example track.

"The SyGuS problem is to find a function f that meets the specified syntactic and semantic constraints. The syntactic constraint is given as a grammar deriving a set Exp of expressions that captures the candidate implementations of f. The semantic constraint is a logical formula $Spec$ that captures the desired functionality of f" [24]. SyGuS problems are specified in the SyGuS input format (SyGuS-IF).

We did not participate in SyGuS-Comp, as our project goal is not to create a general purpose PBE system. Nor is the implementation of the SyGuS-IF in the scope of this project. Nonetheless, SyGuS is an interesting project and the only standardised method to compare different program synthesizers. Participating in the SyGuS-Comp would be interesting and can be a part of future work.

Bibliography

[1] D. Yurichev, *SAT/SMT by Example*, 2019. [Online]. Available: https://yurichev.com/SAT_SMT.html

[2] S. Gulwani, "Programming by examples (and its applications in data wrangling)," in *Verification and Synthesis of Correct and Secure Systems*. IOS Press, January 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/

[3] D. C. Halbert, "Programming by example," Ph.D. dissertation, 1984, aAI8512843.

[4] R. Singh, "Accessible programming using program synthesis," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA , 2014. [Online]. Available: http://hdl.handle.net/1721.1/93834

[5] T. Lau, "Why PBD systems fail: Lessons learned for usable AI," Florence, Italy, 2008.

[6] R. Bodik, "Program synthesis: Opportunities for the next decade." [Online]. Available: https://youtu.be/PI99A08Y83E

[7] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 317–330. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423

[8] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 816–827, Jun. 2016. [Online]. Available: http://dx.doi.org/10.14778/2977797.2977807

[9] S. Gulwani, "Data wrangling using programming by examples," July 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/data-wrangling-using-programming-examples/

[10] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 381–390. [Online]. Available: http://dl.acm.org/citation.cfm?id=645927.672045

[11] E. Torlak, M. Vaziri, and J. Dolby, "Memsat: Checking axiomatic specifications of memory models," *SIGPLAN Not.*, vol. 45, no. 6, pp. 341–350, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806635

[12] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," *SIGPLAN Not.*, vol. 48, no. 6, pp. 287–296, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462174

[13] A. S. Koksal, Y. Pu, S. Srivastava, R. Bodik, J. Fisher, and N. Piterman, "Synthesis of biological models from mutation experiments," *SIGPLAN Not.*, vol. 48, no. 1, pp. 469–482, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429125

[14] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: http://dx.doi.org/10.1561/2500000010

[15] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, pp. 15–26, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462195

[16] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '10. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1836089.1836091

[17] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970. [Online]. Available: http://www.jstor.org/stable/2334940

[18] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, "A machine learning framework for programming by example," *30th International Conference on Machine Learning, ICML 2013*, pp. 187–195, 01 2013.

[19] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557 – 581, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221798003646

[20] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: ACM, 2013, pp. 135–152. [Online]. Available: http://doi.acm.org/10.1145/2509578.2509586

[21] ——, "A lightweight symbolic virtual machine for solver-aided host languages," *SIGPLAN Not.*, vol. 49, no. 6, pp. 530–541, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594340

[22] O. Polozov and S. Gulwani, "Program synthesis in the industrial world : Inductive , incremental , interactive," 2016.

[23] S. Padhi. (2019) Syntax-Guided Synthesis Comeptition. [Online]. Available: http://sygus.org/

[24] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018. [Online]. Available: http://doi.acm.org/10.1145/3208071

University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

Title of work:

Data Wrangling Using Programming by Example

Extending String Manipulation with Known-Entity-Translations

Thesis type and date:

Bachelor's Thesis, March 2019

Supervision:

Simon Felix

Customer:

Shouldcosting GmbH

Students:

Name:	Hasan Selman Kara	Name:	Patrick Burkhalter
E-mail:	hasan.kara@students.fhnw.ch	E-mail:	patrick.burkhalter@students.fhnw.ch
Student-ID:	15-652-969	Student-ID:	15-652-209
Semester:	8.	Semester:	8.

Statement regarding plagiarism:

By signing this statement, we affirm that we independently produced this paper and adhered to the general practice of source citation in this subject-area.

Hasan Selman Kara, Windisch, 21.3.2019: _____

Patrick Burkhalter, Windisch, 21.3.2019: _____