Sara Sá Almeida

Bachelor of Science

Type-driven Synthesis of Evolving Data Models and APIs

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Costa Seco, Assistant Professor, NOVA University of Lisbon
Co-adviser: Bernardo Toninho, Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2020

Abstract

Modern commercial software is often framed under the umbrella of data-centric applications. Data-centric applications define data as the main and permanent asset. These applications use a single data model for application functionality, data management, and analytical activities, which is built before the applications.

Moreover, since applications are temporary, in contrast to data, there is the need to continuously evolve and change the data schema to accommodate new functionality. In this sense, the continuously evolving (rich) feature set that is expected of state-of-the-art applications is intrinsically bound by not only the amount of available data but also by its structure, its internal dependencies and by the ability to transparently and uniformly grow and evolve data representations and their properties on-the-fly.

The GOLEM project aims to produce new methods of program automation integrated in the development of data-centric applications in low-code frameworks. In this context, one of the key targets for automation is the data-layer itself, encompassing the data layout and its integrity constraints, as well as validation and access control rules. This work, which is integrated in GOLEM, will focus on the challenge of defining and evolving a rich data layer component by means of high-level operations, and investigate the ability to synthesize both database scripts and/or code for a data layer access component that correctly implements integrity constraints, business logic validation, and access control rules.

The output of the tool we intend to design and develop will synthesize database scripts and code for the data access layer that correctly transforms the system and its data. The prototype will be illustrated and integrated in the context of a product of the OutSystems company, interacting with other components that generate interface components, or dialog-based interfaces that build the synthesis specification.

Keywords: Program Synthesis, Type-driven Synthesis, Refinement Types, Data Models, APIs, Data Schema Evolution

Contents

Chapter 1

Introduction

Computer science has long evolved from the setbacks experienced at the beginning of its history, having to face new challenges now. When computing machines appeared, the focus was mainly on the actual construction of the machines rather than its software, these earlier computers were mostly built for military/government purposes [38]. At the beginning of the 1950s, computers started being produced commercially, even though not on a large scale and for a narrow audience. These computers had to be programmed using low-level machine-depend code, resulting from the fact that hardware capacity was very limited, the code had to be very efficient to use the resources wisely and computers were costly [38]. The difference between software and hardware was not clear, since producing software was basically writing machine code.

In the later years, advances in hardware technology [10, 19, 33, 38] permitted the construction of more powerful, reliable, and cheaper computers which resulted in an increase in computer's production. The availability of more computers propelled a growing community of programmers and the birth of many programming languages (e.g. FORTRAN, ALGOL, and LISP), which enabled to write code at a high-level of abstraction, that after would be translated into machine code [38]. At this time, programming did not consist of writing machine code directly but relied on the use of programming languages, which allowed for the production of software to become a separate activity than the production of hardware. The focus passed on to the design of languages and their compilers.

The increase in hardware capacity promoted the development of programming and the ability to do large scale software projects, but it also put some strain on the software community that could not match this evolution and it would later originate the so-called software crisis [8, 10, 19, 33, 38]. This crisis was characterized [8, 10, 19, 33, 38] by large software projects constantly being late and over budget, the code was complex and of a large size, which in turn resulted in a large number of bugs. Since software started to be

used for critical activities, bugs represented a high risk. There was also a lack of skilled programmers which did not permit to build complex, and correct software systems. The widely used testing techniques could not assure the total correctness of software. There was a duality between trying to reduce production/cost of software and reduce bugs, decreasing one would probably increase the other. The state of the situation, in the sixties, reached a point where projects were failing, manifested bugs, and producing code did not mean producing correct code. The software crisis needed to be addressed. The NATO Conference [10, 33, 38] in 1968 became a pivotal moment, which reunited professionals with different backgrounds to discuss possible solutions to the software crisis, there was a common understanding that there was a problem to be addressed. Software correctness was among the issues discussed, since computers were starting to be used for critical activities, it could even present risk to human lives. On this matter, there were two lines of reasoning [33, 38]: the academics (such as Dijkstra and Hoare) that agreed that, to prove program's correctness, its construction had to be treated as a science by applying mathematical reasoning and that testing techniques could not show that the software did not have errors, just that it conformed to its purpose; the practitioners who were interested in producing software that worked which then used testing techniques to prove that the software met its purpose.

As a form of incorporating rigorous mathematical reasoning into software, a new field emerged, program verification [33]. Program verification incorporates a body of techniques that prove properties, on programs, defined as formal specifications by applying mathematical reasoning. These techniques are used to prove program correctness. Hoare [20] claimed that program properties and execution results could be analyzed from the code itself with the application of deduction techniques, and so introduced an axiomatization, a logic, that enables us to reason about the correctness of statements through its pre-conditions and post-conditions. Dijkstra, that also believed that one should reason about program correctness through a mathematical perspective [10, 38], took this notion a bit further. While noting that formal specifications were being used to prove program correctness after they are built, presented an idea that these specifications can also be used to produce a correct program by construction that would not need to be proved correct afterwards. Which revealed to be a foundational work to the field of automatic programming/program synthesis.

Since Dijkstra's work had a considerable contribution to program synthesis, we decided to apply the reasoning explained in [11], to derive correct programs from specifications, to an example of our own so that we can get a better grasp of the mindset. In [11], Dijkstra presents a calculus for the derivation of programs from specifications. He introduces two examples in the paper. The first is based on a conditional statement and the second comprises an iterative program. Both are defined using guarded commands. The semantics is presented by means of weakest pre-conditions (predicate transformers) and a group of rules that, if applied correctly, allow the derivation of a correct program and

the awareness of having reached the goal [11].

Consider a very simple example from [11] to demonstrate the general idea. This consists of deriving a conditional statement from a specification. The program to derive, computes the maximum between two numbers, x and y, and performs $m := max(x, y)$. With fixed x and y, the output m of the program will either be x or y and will be greater or equal than both, which forms the specification (R) in Equation (1.1)

$$\text{R: } (m = x \textbf{ or } m = y) \textbf{ and } m \geq x \textbf{ and } m \geq y \tag{1.1}$$

The general mechanism consists in analyzing the specification in Equation (1.1) and selecting statements such that R holds. The procedure follows the general form of guarded commands, the operator wp is then applied to each selected statement so that we can derive the guard that should hold when the statement is executed, leading to an execution that terminates in a final state where R holds. If all the guards are obtained by the wp operator and these guards cover every possible input (so that abortion does not happen), the conditional statement will terminate in a final state where R holds. Following this reasoning, the author first tries the statement $m := x$ which makes R hold and uses wp to obtain its guard:

$$\begin{aligned} wp(\text{“}m := x\text{”}, R) &= (x = x \textbf{ or } x = y) \textbf{ and } x \geq x \textbf{ and } x \geq y \\ &= x \geq y \end{aligned}$$

With this first statement and the corresponding guard, the first draft of the program is the following:

$$\textbf{if } x \geq y \longrightarrow m := x \textbf{ fi}$$

The process is only concluded when any initial value of x and y makes a guard hold. For example, if the initial value of y is greater than x the program will abort, which means this is not the final program and there needs to be at least an extra guarded command. Repeating the same process for the statement $m := y$ produces the following program

$$\begin{aligned} &\textbf{if } x \geq y \longrightarrow m := x \\ &\square\ y \geq x \longrightarrow m := y \\ &\textbf{fi} \end{aligned}$$

The program is considered the final one, since for any value of the inputs there is always a guard that holds, so the program does not abort.

Concluded the explanation of the example above, now we can have a look at our own example solved by applying Dijkstra's reasoning, which shows that deriving an iterative program is slightly more challenging. Given that the paper describes a declarative system rather than an algorithm, it is instructive to illustrate the derivation of an iterative program, for a different example (than the one in the paper) to make the transitions between every step clearer. Thus, by analogy with the *gdc* example (which will be explained in Section 2.1.1), we now derive a program that computes the sum of all natural numbers from

0 to N. According to Dijkstra [11], to derive the iterative program regarding the sum, we must first establish an invariant relation and a variant function. The invariant property is a condition that must hold before and after every iteration. The variant function consists of a measure that decreases with each iteration, in order to achieve termination.

We begin by defining an auxiliary predicate, $Sum(r, n)$, that holds whenever r is the correct value of the sum of all natural numbers up to n. The inductive definition of the predicate $Sum(r, n)$ is thus defined by the two cases:

$$Sum(0, 0)$$
$$Sum(r + (n + 1), n + 1) \text{ if } Sum(r, n) \text{ with } r > 0 \wedge n > 0$$

This predicate will help defining the invariant relation P, while several formulations of P are possible, we choose to represent P as:

$$P : Sum(r, n) \wedge 0 \leq n \leq N \wedge r \geq 0$$

Since the final goal is to compute the sum of all natural numbers from 0 to N, for a fixed N, when the program terminates, $Sum(r, N)$ should hold and r will contain the final result. The general structure of this class of derived programs is:

"initialization of local variables"
do "guarded command set" **od.**

The general structure of the commands is $r, n := E1, E2$, since there are two variables in the invariant (r and n).

Firstly, regarding the initialization of the program variables, r and n are the variable names referenced in the invariant and should be given appropriate initial values. Considering our definition of the predicate and that $Sum(r, N)$ should hold when the program terminates, the variables r and n will both start at 0. This initialization guarantees that P holds before the first iteration. Therefore, the first statement of the program will look like $r, n := 0, 0$;

Secondly, the process of generating guarded commands is an iterative one. Deriving guarded commands until the final result can be calculated from $P \wedge \neg BB$ is the disjunction of all the guards of the program), meaning that the result can be calculated from the available guards since the iterative program ends in a state where none of the guards is true but the invariant is always maintained true. If the guards in the program cover the variable values defined in the invariant, then the result can be fully calculated. If some valid variable values are not considered by the guards, in some cases the result cannot be calculated, so the process of generating guarded commands for the program must continue.

To obtain the first guarded command, we must indicate that the values of r and n are placeholders, replacing r and n with E1 and E2 ($r, n := E1, E2$). The interesting guard

(B) will be the weakest pre-condition such that "$r, n := E1, E2$" can be executed while preserving P.

$$\begin{aligned}(\text{P and B}) &\implies wp(\text{“}r, n := E1, E2\text{''}, P)\\ &= (Sum(E1, E2) \land 0 \leq E2 \leq N \land E1 \geq 0).\end{aligned}$$

The expressions $E1$ and $E2$ have to be chosen so that the guard is expressed in terms of the program variables, and the expressions preserve the truth of P after the iteration. To achieve this, we will manipulate $Sum(r, n)$. Inspecting the second step of the inductive definition of the predicate, we observe that $Sum(r, n)$ is equivalent to $Sum(r+(n+1), (n+1))$. Previously we had that (P and B) $\implies$ $(Sum(E1, E2) \land 0 \leq E2 \leq N \land E1 \geq 0)$ and $P = Sum(r, n) \land 0 \leq n \leq N \land r \geq 0$ so substituting $Sum(r, n)$ with $Sum(r+(n+1), (n+1))$ will express the following $(Sum(r+(n+1), (n+1)) \land 0 \leq n \leq N \land r \geq 0$ and B) $\implies$ $(Sum(E1, E2) \land 0 \leq E2 \leq N \land E1 \geq 0)$.

From the predicates equivalence, we obtain that $r, n := r + (n+1), (n+1)$ and since $r, n := E1, E2$, that means that $E1 = r+(n+1)$ and $E2 = n+1$. Applying the weakest pre-condition operator (wp) with $r, n := r + (n+1), (n+1)$ will express the following $wp(\text{“}r, n := r + (n+1), (n+1)\text{''}, P) = (Sum(r+(n+1), (n+1)) \land 0 \leq n+1 \leq N \land r+(n+1) \geq 0)$. Extracting $n+1 \leq N$ from the expression, we will get the guard $n < N$.

The previous guard was found regarding the invariance of P, but it also has to guarantee the decrease of the variant function. A variant function t should be selected such that it decreases throughout the iterations. Considering the initial values of the variables, for example, n varies between the values 0 and N (starting at 0) so a measure for t could be $t = N - n$. Since n is increased on every iteration until it reaches the value N, t will always decrease.
Using $t = N - n$, we will get:

$$\begin{aligned}wp(\text{“}r, n := r + (n+1), (n+1)\text{''}, t \leq t_0) =\\ wp(\text{“}r, n := r + (n+1), (n+1)\text{''}, N - n \leq t_0) = (N - n - 1 \leq t_0)\end{aligned}$$

The smallest solution for the unknown t_0 is defined as tmin. This means that $tmin = N - n - 1$ and we can use the following definition of wdec: $wdec(S, t) = (tmin(X) < t(X))$ (more details about the relationship between wp and wdec in [11]). Applying the wdec predicate transformer (that denotes the weakest pre-condition so that executing the statement will leave the system in a final state that decreases t by at least 1), the condition is obtained by $wdec(\text{“}r, n := r + (n+1), (n+1)\text{''}, t) = (N - n - 1 < N - n) = (N - N - n - 1 < -n) = (-n - 1 < -n) = (n + 1 > n)$. This condition always holds, so this means that no further restriction will be imposed on the guard.

The first attempt at deriving the program can be expressed as the program below:

$$r, n = 0, 0;$$
$$\textbf{do } n < N \longrightarrow r, n := r + (n+1), (n+1) \textbf{ od.}$$

Since non BB is equal to $n \geq N$ then (P and non BB) $\implies Sum(r, N)$ and r will be the final result. The final result can be calculated from the generated guard since the guard covers all values of n less than N. The value of N is also added to the sum, even though the guard does not cover it, because the statement $r := r + (n+1)$ always adds the successor of n to the result. Which results in the sum of all values from 0 up to N. Non BB is equal to $n \geq N$, which is a range of numbers the program does not need to consider in the calculation. Once the final result can be calculated from the available guards, no further guard command search is needed, the above program is the final one.

As illustrated above, Dijkstra's work [11] presented the reasoning that an expert can follow to construct a correct program from a specification. The following years brought major improvements into the process, in terms of allowing non-experts to have alternative ways of defining the specification and automation of the process. Alternative forms of specification appeared such as input-output examples, natural language, partial programs, among others. Since then, many synthesizers [14, 27, 29] have been built that automate the process.

Nowadays, a widespread population has access to computers. A reality that before did not seem possible, when just a few computers existed. Personal computers uses now range from personal use to education/workplace purposes. Most of these end-users are not programmers or have little programming experience and may need to produce scripts to execute certain tasks. Programming is not an easy or fast activity for them to master, which creates the need to develop tools that enable the average end-user to be able to accomplish the desired tasks within little time. In addition to that, programming still has a major component of repetitive tasks that do not permit developers to focus on more important activities such as design or context related problems. Both the end-user and developers can benefit from automation. Automating programming can also help in decreasing programming errors as far as the synthesized code is concerned. In order to increase the automation in programming, program synthesis is an active research area that is concerned with generating a program from a high-level specification. The specification has various forms that are better suited depending on the problem and domain. Pairing the specification with effective program search techniques, program synthesis has the potential to make great changes in programming tasks. The range of applications varies from producing small complex programs, data manipulation tasks, program optimization/repair, among others.

1.1 Problem statement

Modern commercial software is often framed under the umbrella of data-centric applications. Data-centric applications define data as the main and permanent asset. Its data model is built before the applications, which drive the construction of applications around data and provide a single data model for application functionality, data management, and analytical activities. In contrast to traditional application-centric software, which produces data as a consequence of business activity. Moreover, since applications are temporary, in contrast to data, there is the need to continuously evolve and change the data schema to accommodate new functionality.

In this sense, the continuously evolving (rich) feature set that is expected of state-of-the-art applications is intrinsically bound by not only the amount of available data but also by its structure, its internal dependencies and by the ability to transparently and uniformly grow and evolve data representations and their properties on-the-fly.

The GOLEM project aims to produce new methods of program automation integrated in the development of data-centric applications in low-code frameworks. One of the fundamental goals of GOLEM is to reduce the need for programmers to explicitly write code where automation is possible. In the context of data-centric applications and programming, one of the key targets for automation is the data-layer itself, encompassing the data layout and its integrity constraints, as well as validation and access control rules.

This work, which is integrated in GOLEM, will focus on the challenge of defining and evolving a rich data layer component by means of high-level operations, and investigate the ability to synthesize both database scripts and/or code for a data layer access component that correctly implements integrity constraints, business logic validation, and access control rules. The information needed to generate the correct code for the data layer component can be digested from a rich type-based specification, using dependent and refinement types, and from the data stored in the data layer. For instance, a rich type specification for a data component can include various integrity and validation constraints (e.g. the type of a table field can enforce that it must be a positive integer or that the field can only be modified by users with a certain access permission). Moreover, a modification of a data schema that introduces a new data restriction may be deemed invalid due to some data that already exists in the database that violates the constraint (e.g. modifying an existing table field from integer to positive integer type will not be possible if the table contains an element with negative integers). The need for some adaptation function arises, one that corrects the data items that make the new schema invalid.

The output of this work will then integrate with other components that generate interface components, or dialog-based interfaces that interactively try to build the specification used here as input.

1.1.1 Contributions

Our work will make the following contributions:

- Definition of a core language for the specification of data schemas that captures integrity, business/logic, and security constraints. The language is used in the specification of the synthesis problem.
- Definition of a core language for data schema modifications, which is the target of our synthesis procedure.
- Implementation of a type-directed synthesis tool for creation/modification of data schemas, which also considers integrity, business/logic, and security constraints.

1.2 Document Structure

The following sections on the document start with a review of the main the concepts, techniques and applications in program synthesis (Section 2). Sequent sections, first introduce a summary of the key related work (Section 3), next a more detailed description of the proposed approach to the problem stated (Section 4) and conclude with the expected phases of the work plan (Section 5).

Chapter 2

Literature Review

We now introduce a short roadmap of the chapter. Program synthesis is the automatic or semi-automatic generation of a program from a high-level specification. Jha et al. mention that "Automatic synthesis has long been one of the holy grails of software engineering" [21]. Many communities have contributed to this research area throughout the years, such as programming languages, artificial intelligence, machine learning, and program verification.

The classical view of program synthesis has been the deductive approach, using logical specifications as the expression of user intent [1, 21, 30]. Early approaches to the derivation of programs used theorem-proving as a constructive procedure to build proofs and extract programs from those proofs [24]. These approaches were followed by techniques that perform synthesis by transforming specifications into correct-by-construction programs without the need of proofs [24]. All these approaches require complete specifications (e.g. logical formulas) which can prove difficult to express. This originated work that used partial specifications such as examples [18]. Recent advances in SAT and SMT solvers [1] stimulated further improvements of synthesizers, allowing the specification of synthesis constraints that can be passed to this solvers for verification. Current synthesizers often receive as input an implicit restriction (e.g. grammar) of the search space in addition to the specification to make the synthesis problem more tractable [1].

Abstracting from specific details, to find a program consistent with a specification, a program synthesizer has to search the program space using a certain search technique. Gulwani et al. characterized the program synthesis task in three dimensions: user intent, search space, and search technique [18].
These dimensions can be summarized as follows [18]:

1. The user intent is considered to be the high-level specification that expresses the desired program and can be specified in several ways such as logical formulas,

examples, traces, natural language, partial programs or even related programs.

2. The search space/program space defines all possible programs and can be over imperative or functional programs, regular or context-free grammars, succinct logical representations. To restrict the program space, a subset of a general-purpose or domain-specific programming language can be used or alternatively a specifically designed domain-specific language.

3. The search technique is used to navigate the search space to find a program that satisfies the specification. Many techniques can be employed such as deduction, enumeration, constraint solving, among others.

The key challenges in program synthesis are expressing user intent properly and the large search spaces which a synthesizer has to search to find a program that satisfies the specification [18].

Regarding user intent, there is a trade-off between the imprecision of using examples and the challenges of producing correct, complete specifications, as well as doing an adequate specification language. The specification used will also depend on the application in question.

Considering the search space, if it is insufficiently restricted, it may end up being too large for a synthesizer to be able to search it effectively. This is why many synthesizers in recent years have used several strategies (e.g using domain-specific languages (DSLs) or a subset of operators in a language) as a way of restricting the search space. This originated the so-called syntax-guided synthesis problem (SyGuS) [1], which is a community effort to formalize the core ideas behind these approaches as the SyGuS problem. In [1], they formulated and compared three previous approaches using enumerative, constraint-based and stochastic algorithms. The input is a background theory, a specification in the form of a logical formula and a grammar that restricts the possible candidates. The goal is to find a candidate program, constructed from the input grammar, that satisfies the specification according to the theory. In this work, several benchmarks with synthesis problems for different domains as well as an annually run synthesis competition were created. This competition has stimulated the appearance of new and better synthesizers [2].

The next sections will present the general approaches to program synthesis, being inductive synthesis 2.1 and deductive synthesis 2.2. We are aware that not all approaches fall exactly into the buckets of synthesis that generalizes from incomplete examples or synthesis that apply deductive reasoning to complete specifications. The same happens with synthesis techniques, a synthesizer may use one technique or apply a combination of techniques. For example, there are inductive synthesizers that apply deductive techniques. The following structure facilitates the presentation of the literature and presents an exploration through both paths, presenting synthesis techniques and applications.

2.1 Inductive Synthesis

Inductive Synthesis is generally characterized as the synthesis approach that receives examples as a specification [1, 30, 35]. These examples can be expressed in multiple forms such as input-output examples, tests, partial programs, etc. On one hand, examples have the advantage of being easier to express than, for example, complete logical specifications. On the other hand, they can be quite ambiguous, and so there is the possibility of having multiple semantically different candidate programs consistent with the specification [17]. In Section 2.1.1 we will see techniques used by inductive synthesizers to obtain a solution consistent with the specification and in Section 2.1.2 the resulting applications.

2.1.1 Synthesis Methods

In this section we present the main synthesis techniques: Enumerative, Constraint Solving and Stochastic Search. We will also mention a few other existing techniques.

Before going into the details of each approach, we highlight the key challenges. The following techniques may have to deal with concepts such as defining (and possibly restricting) the program space, dealing with ambiguity and involving the user in the synthesis process.

To mitigate the problem of large search spaces, many inductive synthesizers employ techniques such as restricting their program spaces by using DSLs, supplying a partial program (sketch) so that only unspecified parts have to be synthesized [35], or generate a program from a library of components [14, 21]. Restricting the program space to a DSL enables the incorporation of domain-specific knowledge into the synthesis process, even though the candidate programs will be restricted by the set of operators in the DSL.

Component-based Synthesis leverages a library of program components and formulates the synthesis problem as a composition/orchestration of a subset of the component library as a program. Approaches such as [21] start from a small set of components and, if they are insufficient, allow the developer to iteratively augment the component library until the synthesis is successful.

The partial program approach restricts the search space in the sense that the synthesizer only has to produce code for the unspecified parts (holes). In SKETCH [35], the holes are also restricted in the values they can take, which further reduces the possibilities.

These different methods of program space restriction inspired the SyGuS formulation [1]. In the competition for syntax-guided solvers, in 2016, a specific track was created for syntax-guided approaches that receive examples as the semantic specification instead of the original formulation with the logical formula [2].

In order to deal with the inherent ambiguity of the problem, there are synthesizers that apply ranking techniques to the set of programs consistent with the specification, ordering them according to some measure [17]. Since selecting a random program that is consistent with the examples may not be the best solution. These ranking functions

are of two general types: manual ranking functions which employ heuristics or learned ranking functions.

A simpler ranking technique is to enforce preferences over candidate programs. For instance in [39], candidate queries are ranked higher if they have a more natural structure, use predicates that are more common and cover many different constants.

Learned ranking functions use a large amount of training data to learn how to rank programs. MORPHEUS [14] uses around 15,000 code snippets from Stackoverflow to train a 2-gram model that allows to order program candidates based on the score from this model. FlashFill [34] applies a gradient descent method to learn a function that classifies programs as positive or negative, and ranks the positive higher.

Finally, with the increasing development of programming-by-example frameworks that are used by the general public [16, 34], there is the need to involve users in the synthesis process, to increase user's confidence in the results and resolve ambiguities together with ranking functions. Techniques that involve user interaction may help in that sense, such as querying the user during the synthesis process. For instance, the work in [26], introduced two user interaction models. The first allows the user to search over all DSL synthesized programs which are translated to natural language so that the user can pick the right one. The second asks questions to the user based on the synthesized programs that are consistent with the examples in order to refine the initial specification and repeat the process.

2.1.1.1 Enumerative

Enumerative Search works by enumerating all programs in the search space according to a certain order such as program size, complexity, etc [18]. A common strategy is to enumerate programs by size. The procedure iteratively synthesizes programs of increasing size, by applying generation rules to programs from previous iterations. We illustrate this approach with an example from [28], where the program space is defined by the grammar in (2.1).

$$\begin{aligned} \text{Start} &\rightarrow \text{String} \\ \text{Int} &\rightarrow 0 \mid 3 \\ &\mid (+\ \text{Int Int}) \\ &\mid (\text{str.indexof String String}) \\ \text{String} &\rightarrow \text{“ ”} \mid \mathit{input} \\ &\mid (\text{str.substr String Int Int}) \end{aligned} \tag{2.1}$$

The expressions generated by the grammar will be enumerated by height. The goal is to maintain a net of enumerated expressions, that is initially empty and grows with every iteration. In the example, with height 0, the first enumerated elements are literals and variables, being 0, 3,“ ” and input. To generate programs of height 1, the production rules from the grammar are applied to the previous elements of the net. For example

Int → (+ Int Int) is applied to all pairs of values Int and the programs (+ 0 0), (+ 0 3), (+ 3 0) and (+ 3 3) are added to the net. The iterative process continues in a similar manner, increasing the height and exploring all possibilities. The grammar used in this example is very simple, which results in a small amount of enumerated programs. However, with larger program spaces the number of enumerated programs will grow very large.

As listed above, this technique has the tendency to produce a large number of programs even if the program space is restricted in some way. In order to address this problem, several synthesizers apply pruning techniques, so that fewer programs are stored. One such pruning technique consists in the use of equivalence classes [28, 39]. At each phase, enumerated programs are grouped based on an equivalence metric, these groups are called equivalence classes. Each subsequent phase enumerates programs that can be constructed from the representative of each equivalent class. This technique effectively partitions the search space, avoiding the enumeration of different programs that are functionally equivalent.

Another approach that also considers program equivalence is observational equivalence (OE) [28]. This approach prunes programs that have the same output on all examples and is considered a more aggressive version of the previously described approach. Unlike with equivalence class pruning, which requires grouping of generated programs and computing representatives, OE pruning evaluates programs on the input examples and produces a set of outputs. If two programs at any stage of the procedure produce the same set of outputs, the latter one is discarded since it is deemed observationally equivalent. A disadvantage, in this case, is that a program that was discovered later and has the same output net of a program that was discovered earlier, will not be kept, in spite of potentially being a better fit to the problem [28].

There are alternative forms of pruning the search space that use deductive techniques, which mix enumeration and deduction. In [3], a divide and conquer strategy is used. Terms that are correct on some subset of the input are enumerated until the enumerated terms cover all the input. At this stage, predicates are enumerated until a conditional expression using the terms is found. An algorithm for decision tree learning is leveraged to find a decision tree that joins the terms and predicates into a tree consistent with the examples. This enables a faster enumeration process, considering fewer candidates. Other works such as [14] use SMT solvers for pruning the candidates. Components from a library are enumerated so that they form a sketch, similar to the one of the SKETCH system [35], representing multiple candidates, depending on how program holes are filled. Each component has a specification that is combined to find the specification of the sketch, which is then encoded into an SMT formula. The SMT solver discovers if the sketch satisfies the input-output examples. Note that pruning sketches results in pruning many possible candidates.

In summary, enumeration is a technique commonly used to find candidate programs

consistent with a specification. Pruning techniques help to greatly decrease the complexity of the enumeration process. Multiple approaches that do not uniquely use enumeration still have an enumeration step [3, 29].

2.1.1.2 Constraint Solving

Program verification is a research area that has close connections with program synthesis. While the former tries to prove a specification in a program, the latter tries to build a program from a specification. In this research area, the use of SMT (Satisfiability Modulo theories) solvers is recurrent across different verification tools [1].

SAT solvers determine the satisfiability of a formula but they cannot take into consideration background theories (that give an interpretation of predicates and functions) [5]. SMT solvers, are able to deal with the satisfiability of formulas given a background theory [5]. Advances in SAT solving [1, 5] allowed to build better SMT solvers.

As in program verification, program synthesis also takes advantage of SMT solvers. The use of such solvers allows to delegate the complexity of traditional techniques that search through all possible programs to the solver, which can often deal with the complexity of the task effectively [21]. The program space and specification are encoded into SMT formulas that are leveraged to the solver.

The SKETCH system [35], besides introducing the idea of sketches, also introduced the notion of a counterexample-guided inductive synthesis algorithm (CEGIS). This algorithm separates the synthesis and validation of programs into different stops. Using an SAT-based inductive synthesizer to produce candidate implementations from a set of inputs and a bounded model-checker to validate the candidates and produce counterexamples. The main idea behind this algorithm is that a small set of inputs can represent the correct program, using inputs that represent specific situations (which the author calls "corner cases"). Thus, only a few iterations will be necessary to produce the correct program if it exists. The algorithm starts with a random input from which it synthesizes a candidate implementation that is submitted to the validation procedure. If the program is the desired one it is accepted, otherwise a counterexample is produced. Each counterexample is added to the inputs in order to synthesize a new candidate and start a new iteration until the validation procedure accepts the output. By separating the synthesis and the validation process, newer or more adequate validation procedures can be used in other instantiations of the algorithm.

This work inspired another approach that, like CEGIS, separates these concerns. The so-called Oracle-guided synthesis [21] uses an SMT solver to synthesize a program from a library of components and two oracles, an I/O oracle and a validation oracle. The I/O oracle substitutes the need for a complete specification, since it does not require a large set of input-output examples. When queried on any input, it returns the output on the desired program. What differentiates this approach from the previous one is that besides not needing counterexamples, it attributes a higher cost to querying the validation oracle

so that it only allows to query it once [21], and so avoids querying the validation oracle on every iteration like CEGIS. This is possible by defining two types of constraints: one that enables the synthesizer to produce a well-formed candidate and another that checks that, given a candidate program, there is not another program respecting the given examples that produces a different output on any given input. This allows the algorithm to only query the validation oracle when the program is specific enough that it would not return a different answer from another candidate program. The iterative process starts with a random input and queries the I/O oracle to get the output. If it can generate a candidate from the set of components, it checks if it is specific enough, if it cannot, the procedure terminates. If the program is not specific enough it obtains an input for which two non-equivalent programs generate different outputs, requests the correct output from the I/O oracle, starting a new iteration to synthesize a new candidate with this added example. When a program that returns always the same output as other candidates is found, the query oracle is queried and decides if it is the correct program or if there is no solution. The general architecture of the syntax-guided synthesis problem (SyGuS) [1] was also instantiated using the CEGIS algorithm. Synthesizing candidates not just through constraint solving but also using techniques such as enumerative and stochastic approaches.

Regarding the encoding of SMT formulas for program synthesis [18], the specification and program space restrictions are encoded into one formula (SMT formula). The SMT solver finds an instantiation of the variables that makes the formula true. Each instantiation that makes the formula true corresponds to a correct program. There are approaches [21] that encode the constraints directly, which is a complex task and then they have to map the solution back to a program [37]. Solver-aided languages [35, 37] help in this task by providing high-level programming languages with constructs that are then encoded into the SAT/SMT formula by the framework. SKETCH [35] provides a language for specifying sketches leaving holes in the sketch for the synthesizer to fill. The sketches are later encoded into SAT formulas by the language compiler. ROSETTE [37] is a solver-aided language, used as an extension of the Racket language, adding to the language four query constructors. The constructors can be used to verify an implementation, synthesize code, localize bugs and to call an oracle. When used in an application, they are compiled directly into the constraints and developers don't have to do it themselves.

2.1.1.3 Stochastic Search

Stochastic techniques sample programs from the program space according to defined metrics to guide the search [1, 18, 32]. The use of a Markov Chain Monte Carlo (MCMC) sampler together with a cost function over the program space is an example [32] of an application of the technique. A cost function is defined according to the desired properties of the program. In the case of program optimization in [32], the cost function considers the similarity of the program by comparing to the program to be optimized and

the performance improvement. The goal is to guide the search with the cost function and minimize the cost of the obtained program. MCMC samplers obtain samples from probability density functions and sample programs with a higher probability more often [32]. The MCMC sampler in the case of [32] uses the cost function and returns programs with a low cost. In the formulation in [1], with a different metric, a score is attributed to each expression and measures how much the expression satisfies the specification. If the expressions have a higher score, they have a higher probability of being sampled.

Genetic programming [22] is also considered a stochastic technique that begins with an initial population (of programs) and applies operations inspired by biological evolution such as mutation and crossover that continuously alter these programs until they satisfy a given fitness function. This fitness function may be defined by tests, input-output examples, or other properties [18].

2.1.1.4 Other techniques

We now mention other techniques related to the use of machine learning and neural networks.

There are several possible uses of machine learning in the program synthesis task. Earlier we mentioned the use of learned ranking functions to rank candidate programs. Considering the search technique part, we will mention examples such as decision tree learning [3] and guiding the search with learned probabilistic models [23]. When we discussed the enumerative technique and the combination with deduction, we mentioned the divide and conquer strategy. The divide and conquer strategy [3] enumerates terms and predicates to construct a decision tree that represents the program. To build this decision tree it uses a decision tree learning algorithm. The algorithm [3] first determines if there is a term that applies to all examples which could represent a tree with only one node. If it does not find such a term, it tries to find a predicate to split the terms with respect to the examples, based on an information gain heuristic. It continuously tries to build a tree that will cover all the examples and represent the final program.

On another note, machine learning techniques can also be used to guide the search procedure. Many search approaches do not consider certain programs more likely than others, which results in searching for several candidates that will not satisfy the goal. One possible approach [23] is to formulate the problem as a syntax-guided synthesis problem which is restricted by a grammar and extend the grammar with a probabilistic model. The model is trained on previously solved program synthesis tasks. The solution is found by performing a weighted enumeration on the model through the A* algorithm. The enumeration is ordered by probability, starting with higher probability.

When using neural networks for automatic program learning, the categories are divided into program synthesis and program induction [9]. The approaches in the first category use a neural network that given the input-output examples builds an actual program. The ones on the second category, learn how to map the inputs to outputs from

the set of input-output examples and then are able to provide the correct outputs given new inputs. Neural network techniques can handle errors (or noise) in the examples specification, in contrast to methods using traditional synthesis techniques which is an advantage [9].

Approaches such as guiding the search with a learned probabilistic model and the use of neural networks need a large amount of training data which is not the case in the traditional programming by examples techniques that need only a few input-output examples [9, 23] . That may complicate the task of using machine learning and neural networks if such amount of training data is not available. This can be solved by synthesizing training data, such as in [18].

2.1.2 Applications

The spectrum of synthesis tools that inductive synthesis is able to produce is very wide. We mention a few applications, such as: bit manipulation, data manipulation, queries and program deobfuscation, optimization and repair.

The main use of inductive synthesis is to be able to produce a program from a set of examples or partial programs. In low-level programming and other areas that require the direct manipulation of bits, producing efficient bit manipulation programs is not an easy task even for expert developers [21]. This is the reason why the synthesis community has provided solutions to tackle this problem. For example, if the developers have a general idea of the solution but not of the exact details, they may provide a program with the general structure and the synthesizer will produce those low-level details [35]. Another example is to provide a set of input-output examples of the expected behavior of the program and the main bit manipulation operators [21]. Within the work of formalizing the syntax-guided problem [1], the authors created several benchmarks with synthesis problems, within these there were bit manipulation and bit-vector problems. These benchmarks inspired the appearance of more synthesizers for these kinds of tasks.

Programming-by-example has had a major influence in the field of data wrangling. Nowadays large amounts of data that have to be manipulated, stored, and analyzed are produced and available. Data scientists have to analyze all of this data and draw conclusions from it. However, data preparation can often take 80% of their time [14]. Data wrangling concerns activities of data manipulation, such as data extraction and transformation. FlashFill [34] is a feature incorporated in Excel 2013 that automates string transformations. From one example of a transformation, the system suggests how to transform other inputs. WREX [12] is an extension for the Jupyter Notebook that generates code for data transformations such as string/number/date transformations. In contrast to FlashFill, WREX generates the code for data scientists to analyze and FlashFill only shows the possible solution. Considering table manipulations, both FlashExtract [16] and FlashRelate [4] are tools that extract structured data from semi-structured data.

FlashExtract extracts structured data from text/log files and webpages. FlashRelate extracts relational data from spreadsheets. Since data is being stored and transmitted in several formats that are often unstructured, these tools help to automatize that task.

While the works above focus on manipulating data from spreadsheets, there are also tools that synthesize general table manipulations such as MORPHEUS [14]. With the input and output table information, MORPHEUS synthesizes the program that does the transformation on the table. It is capable of doing operations such as table reshaping and table consolidation.

Another kind of table manipulation programs that can be synthesized are queries [13, 39, 41]. Queries consult/insert/delete data on database tables. Specifying the query intent through examples enables non-expert users to be able to query database information. Some synthesizers focus explicitly on SQL queries [39, 41].

Synthesis is not only used to produce programs from scratch, but it also can take a complete program as specification and perform tasks such as program deobfuscation, optimization or repair. Obfuscated programs are programs that perform malicious actions, namely malware [21]. Since it is hard to understand deobfuscated code and many approaches deobfuscate the program manually, the work in [21] permitted to automatize this task. From the obfuscated initial program, the synthesizer produces a deobfuscated and much simpler program. Starting from an inefficient program, synthesis is also used for program optimization [32]. Performance constraints enable the synthesizer to return an optimized program. The repair of programs [22] is also possible, for example, starting with a program that has a bug and a group of tests, it is possible to synthesize a program that will pass those tests.

2.2 Deductive Synthesis

Deductive synthesis is the net of approaches that take logical formulas as a specification of the desired program [24, 25]. Providing logical formulas as a specification is often viewed as a challenging task, which can often be out of reach of the average end-user [18]. The advantage that these complete specifications have is that they fully specify the constraints on input and output, unlike examples that only specify a portion of the functionality. In Section 2.2.1 we will see deductive synthesis methods and in Section 2.2.2 refer to some applications.

2.2.1 Synthesis Methods

The main lines of work in deductive synthesis span over derivation from specifications, extracting programs from proofs and type-directed approaches. Initial approaches to program synthesis applied deductive theorem-proving techniques to obtain programs [24]. A program could be extracted from the deductive proof of a theorem. Later, newer techniques involving specification transformation or rewriting appeared [24]. More recently,

specific type-directed [27, 29] reasoning has been used for when the specification defines the types of input and output. In addition, using solvers has permitted to automate previously intricate tasks [36].

2.2.1.1 Derivation from specifications

Deriving programs from specifications consists of applying systematic rules to the logical specification of the desired program in order to produce a program that satisfies the specification. The construction of correct programs is based on rules that preserve correctness. Dijkstra's seminal work [11] was foundational in this area. A non-automated way of thinking about the derivation of programs from specifications. In the work, Dijkstra [11] presented a syntax of programs using guarded commands, with semantics given by weakest pre-conditions. The concept of weakest pre-conditions was inspired by Hoare [11, 20], which reasons about statement correctness through pre-conditions and post-conditions. If before executing the statement, the pre-conditions are true, it means that if the statement terminates it will terminate in a state where the post-condition holds. It is not guaranteed that the statement execution terminates. Dijkstra introduced the concept of weakest pre-conditions, weakening the restrictions on the pre-conditions, but still guaranteeing the correctness of the result. The operator used is $wp(S, P)$, denoting the weakest pre-condition for the statement S with the post-condition P. The weakest pre-condition is the necessary pre-condition that should hold so that when the statement is executed, it terminates in a state that verifies the post-condition. Considering that the goal was to derive alternative and iterative programs from specifications, the work presented rules to derive these using weakest pre-conditions and how to prove termination.

Let us consider an example [11] from Dijkstra's work that requires the establishment of a invariant relation and a variant function so that an iterative statement can be derived. The invariant relation states that on every iteration a certain relation between the program variables holds. The variant function guarantees the termination of the loop. The aim of the program is to calculate the greatest common divisor (gcd) between two positive numbers. The final relation for a fixed X and Y (the inputs) is established as $x = gcd(X, Y)$. This is the specification of the problem. The invariant chosen in the example that is used to derive the iterative program that calculates the gcd is the following

$$\text{P: } gcd(X, Y) = gcd(x, y) \textbf{ and } x > 0 \textbf{ and } y > 0$$

The author doesn't mention how to choose the invariant, just that this was the one chosen and not necessarily unique. This invariant defines the program variables x and y, such that they will always be positive and the gcd of these variables will always be equal to the gcd of the variables received as input.

The reasoning behind the construction of the iterative program is to inspect the invariant and extract program statements from it. The statements chosen have to preserve the validity of the invariant. This is not an automated process, requiring domain knowledge

to conduct the derivation. The weakest pre-condition operator (wp) is applied to each statement to find its guard. It discovers the weakest pre-condition that must be satisfied so that, if the statement is executed, it will terminate and the post-condition will hold. To the guard obtained by applying wp, another operator $wdec$ is also applied to make sure that the guard guarantees the termination and the decrease of a measure t by at least 1. This allows for the construction of an iterative program that is guaranteed to terminate if none of the guards are true or, if any guard is true, guarantees that the statement will be executed and terminate verifying the post-condition. This reasoning allows us to construct a correct terminating program from a specification.

The first step is to define the initial values of the program variables. From the invariant, it is possible to understand that they should start with the values of the input X and Y. Obtaining the initial statement $x := X; y := Y;$ for the program.

The second step considers the main structure of the iterative program. A general statement is to indicate that x and y are placeholders by the statement $x, y := E1, E2$ and apply wp.

$$\begin{aligned}(\text{P and B}) \implies & wp(\text{“}x, y := E1, E2\text{''}, P) \\ & = (gcd(X, Y) = gcd(E1, E2) \textbf{ and } E1 > 0 \textbf{ and } E2 > 0).\end{aligned}$$

As explained previously, applying wp to the statement should give the guard of that statement, however the guard obtained here is not computable. Since there is no gcd function available that allows the computation of this guard. Thus we are required to mathematically manipulate the gcd function in a way that maintains the relation $gcd(X, Y) = gcd(E1, E2)$.

Using knowledge from the gcd function [11], it is known that $gcd(x, y) = gcd(x - y, y)$, so the statement $x := x - y$ could be derived from this equivalence.

From this statement, applying $wp(\text{“}x := x - y\text{''}, P) = (gcd(X, Y) = gcd(x - y, y)$ **and** $x - y > 0$ **and** $y > 0)$, the guard $x > y$ is obtained, which guarantees the invariance of P. The measure chosen as the variant in this case is $t = x + y$ and, applying the $wdec$ operator, the condition $y > 0$ is obtained. This guarantees that the process terminates decreasing the measure t by at least 1. The condition $y > 0$ is guaranteed by P so no further restriction on the guard is needed.

A first draft of the desired program is

$$\begin{aligned}& x := X; y := Y; \\ & \textbf{do } x > y \longrightarrow x := x - y \textbf{ od.}\end{aligned}$$

This iterative program contains the initial program statements, the loop, the statement and the guard that was derived. We now reason about the correct termination of the derived program. Dijkstra defined that the iterative statement ends in a state where none of the guards are true and the invariant is true, so in that state the program must be able to calculate the final result x such that $x = gcd(X, Y)$. However, the program constructed until now does not account for the case when y is bigger than x, it cannot calculate the result.

Considering that $gcd(x,y) = gcd(x,y-x)$, the same procedure explained previously can be refined considering the statement $y := y - x$ which will generate the guard $y > x$. Thus results in the program

$$
\begin{aligned}
&x := X; y := Y;\\
&\textbf{do } x > y \longrightarrow x := x - y \textbf{ od.}\\
&\square\ y > x \longrightarrow y := y - x \textbf{ od.}
\end{aligned}
$$

The reasoning about correct termination is repeated. Considering BB as the disjunction of the guards, if the program is in a terminal state where P and non BB holds, we must check if the result can be calculated. Since non BB is $x = y$ and $gcd(x,x) = x$, the result can be fully calculated given that the variable x takes as initial value X which would be the result for equal values of input. Thus, the program can calculate the *gcd* for every positive number of x and y. The construction of the program is considered complete.

After this example, we have shown the complex reasoning required to manually derive a simple iterative program from a total specification. This is why it is compelling to mention a more recent approach that shares the same mindset of obtaining programs that are correct by construction but takes advantage of the current advances in technology to automate the process. The work developed in [36] formulates program synthesis from the program verification perspective leveraging solvers to discharge constraints. The input to the problem is a logical specification, a description of the domain of expressions/guards, and restrictions on the resources the program can use. The algorithm uses these data to create a program where the statements, guards, invariants, and ranking function are not specified. The ranking function here has the same goal of the variant function previously explained, as does the invariant. The synthesis consists of defining verifiable constraints on the program that when given to a verification tool, can discover the missing parts. The authors mention that the insight [36] behind the approach is that when trying to prove partial correctness of a loop, program verification tools synthesize an invariant. If the tools can synthesize the invariant, guards and statements can also be synthesized. To synthesize the program and the proofs of partial correctness and termination (invariant and ranking function), three constraints are defined on the program: a safety constraint to prove that the program produces correct results; a well-formedness constraint so that guards and statements inferred correspond to an actual valid program; and, a progress constraint to ensure termination. Proving these constraints with a verification tool will produce the guards, statements, invariant, and ranking function. The synthesizer does not only consider a logical formula as specification but also other restrictions on expressions/guards and resources. Thus, the process of producing a program with a loop, that had so many steps and was not automated in Dijkstra's work [11] was reduced to verifying three constraints using a verification tool. Before, specifying an invariant and a variant was not an easy task but also analyzing the invariant to extract program statements was not straightforward. This shows a major improvement in synthesis techniques.

2.2.1.2 Extracting programs from proofs

Theorem-proving synthesis techniques are based on extracting the desired program from the proof of theorems extracted from the correctness specification. In the early stages of program synthesis, these techniques were broadly used [24, 25]. Programs would not need debugging or verification since they were guaranteed to already satisfy a given specification. Early work that used resolution-based theorem proving had difficulties using mathematical induction, resulting in not being able to represent iterative or recursive loops [24]. In the early '70s, Manna et al. [25] demonstrated in a general manner how to use theorem-proving techniques to extract recursive and iterative programs. To introduce loops in the program, the principle of mathematical induction has to be used in the proof. They pointed out that the induction principle used greatly affects the form of the obtained program and that mechanical theorem proving at the time probably could not solve many of the simple proofs explained in their work. At this time, a few synthesizers that use theorem-proving already existed. Theorem-proving systems used axioms or rules of inference to store information, each with advantages and disadvantages. In [25], the authors mention that the use of both in a system could be useful. Even though newer techniques appeared that involve transforming the program's specification instead of proving a theorem, there was still work [24] on theorem-proving methodologies. This work was backed by the idea that approaches that do not use theorem-proving directly, still involve some part of it, as for example to prove termination of the constructed program and so there is no point in doubling the work.

To illustrate the theorem-proving method, we consider a simple example from [25] that does not have loops: the goal is to construct a program that returns the maximum between two numbers. We omit an example with loops for the sake of simplicity.

Firstly, the authors outline the general structure of the problem. The specification of the desired program is given by input and output conditions defined by the predicates $\varphi(x)$ and $\psi(x,z)$, respectively. The program constructed receives the input x satisfying the input condition $\varphi(x)$ and calculates the output z such that the output condition $\psi(x,z)$ holds. The process of constructing the program is done through the proof of the theorem $(\forall x)[\varphi(x) \supset (\exists z)\psi(x,z)]$ extracted from the specification. Which in practice states that for every input that satisfies the input condition there exists an output that satisfies the output condition. Proving this theorem is proving that an output satisfying the previous specification exists and the program can be extracted from this proof. When the input condition is true, the theorem is defined by $(\forall x)(\exists z)\psi(x,z)$, just stating that there exists an output for every possible input.

Secondly, the authors point out how this example is specified and proved. In this case the input condition is true. There is no constraint on the inputs considered. Any two numbers can be considered. The program receives the inputs x_1 and x_2 and returns the output z. The output condition is $\psi(x_1,x_2,z) : (z = x_1 \vee z = x_2) \wedge z \geq x_1 \wedge z \geq x_2$. This output condition specifies the goal of the program to be produced: the maximum between two

numbers will be one of the inputs received and it is larger or equal to both inputs. So z takes either the value of x_1 or x_2.
Consequently, the program that satisfies this specification will be extracted from the constructive proof of the following theorem

$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \vee z = x_2) \wedge z \geq x_1 \wedge z \geq x_2]$$

Which states that for any input x_1 and x_2, there exists an output which is equal to one of the inputs received and it is the maximum value between them. If this output exists, it means that the desired program can be constructed. The first step in the example is to turn the theorem into Disjunctive Normal Form (DNF).

$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \wedge z \geq x_1 \wedge z \geq x_2) \vee (z = x_2 \wedge z \geq x_1 \wedge z \geq x_2)]$$

Then, having $(u = v) \supset (u \geq v)$ as an axiom, the formula can be simplified. In the first disjunct there is $z = x_1$ and $z \geq x_1$. Applying the axiom, this is reduced to $z = x_1$. If z is equal to x_1, it is also bigger or equal to x_1. The same goes for the second disjunct. The resulting formula is

$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \wedge z \geq x_2) \vee (z = x_2 \wedge z \geq x_1)]$$

The proof is made by case analysis. If $x_1 \geq x_2$, then z is substituted by x_1 and the first disjunct holds. If $x_2 < x_1$, the opposite happens. In both cases the theorem holds. The program extraction is described as: the substitution of the output variable results in an assignment statement and case analysis results in conditional statements, with a branch per option. This means that the resulting program can be expressed by an if-then-else, **If** $x_1 \geq x_2$ **then** $z = x_1$ **else** $z = x_2$.

As exemplified, theorem-proving techniques prove a theorem extracted from the specification of the desired program. In these proofs we can find the desired program. To automate this process, the theorems should be passed into automatic theorem-provers instead of proved manually.

2.2.1.3 Type-directed Synthesis

In cases where the specification is defined by the types of inputs and outputs, type-directed approaches are used. When we define a program by the types of inputs and outputs that our program should have, we are imposing input and output conditions. One way to approach this could be to enumerate all program candidates and check if their type matches the specification [29]. In this way, candidates that do not typecheck are rejected, but still a lot of combinations are considered. Current type-directed techniques [27, 29] usually work by decomposing the problem into subproblems, where each subproblem is considered individually and then the solutions are combined. The type information of the problem is passed into the subproblems so that each subproblem finds a solution that will agree with the general specification and considers fewer combinations.

Simple types are inhabited by a large number of programs, even though many do not exactly fulfill the intent that the developer wished for the program. Given this, many approaches use extra information in their synthesis process such as input-output examples [27] or add predicates to types (refinement types) [29], allowing to better constrain the program. Independently of using examples or refinements, these are propagated to the subproblems as a way of restricting the possible solutions for each.

We examine an example (Eqs. (2.2) to (2.5)) from [29] and explain how SYNQUID, the tool developed by Polikarpova et.al, finds the desired function. The program is a function that receives a number n and a value x, and returns a list with n copies of x. The specification is shown in Equation (2.2):

$$replicate :: n : Nat \rightarrow x : \alpha \rightarrow \{List\ \alpha \mid len\ v = n\} \tag{2.2}$$

The name of the function is replicate, the type signature specifies the types of the two inputs and the type of the output. As mentioned by the authors, Nat is defined by the refinement type $\{v : Int | v \geq 0\}$ which expresses all integers larger or equal to 0. Both the input n and the output have refined types. Refinement types are types restricted by a predicate. From a synthesis perspective, a predicate on the input type acts as an input condition: the accepted values of n have to be positive. On the other hand, the refinement on the output acts as an output condition: the goal is to return a List that has a length equal to n. This specification also makes use of richer types: the parametric polymorphic type in input x, and the dependent, function type, allowing the type of the result to mention the inputs. Crucially, the elements in the output list, List α, must be the value x because the system must produce a polymorphic replicate function α [29].

Without looking at the details yet, lets examine how could a function with this type signature be built. This is a function that outputs a list defined by the algebraic datatype shown in (2.3).

$$\begin{aligned} &\textbf{termination measure len}\ :: List\ \beta \rightarrow Nat \\ &\textbf{data}\ List\ \beta\ \textbf{where} \\ &Nil :: \{List\ \beta \mid len\ v = 0\} \\ &Cons :: \beta \rightarrow xs : List\ \beta \rightarrow \{List\ \beta \mid len\ v = len\ xs + 1\} \end{aligned} \tag{2.3}$$

The list composite type defines two possible constructors for a list, Nil or Cons. The function will basically have two possibilities to build a list. It either constructs a list with size 0 or bigger. In the specification, n defines the length of the desired list. So a natural way to decompose this problem would be in two subproblems depending on the value of n. The specification of the original problem could be refined into considering n equal to 0 and n larger than 0 in each subproblem, respectively. After solving the solution of both subproblems independently, the solutions would be combined into a solution of the general problem. On a final note, since the constructor Cons receives a value and adds it to a list and the goal is to create n values in the list, this indicates that this constructor will be called several times.

Now we can see how SYNQUID synthesizes the replicate function. To solve this problem, besides the function types, it also needs a collection of components that are used during the synthesis process. SYNQUID [29] works by either by decomposing the problem and trying to find a solution for each part or by picking components from the collection. The initial set of components has the functions 0, inc, dec mentioned in (2.4) and the list datatype mentioned above in (2.3). The list datatype has a termination measure to ensure that recursion on lists terminates, which resembles the approach used by Dijkstra with the variant function in the example mentioned previously in deriving from specifications. With loops or recursion, termination must be guaranteed.

$$\begin{aligned} &0 :: \{Int \mid v = 0\} \\ &inc :: x : Int \rightarrow \{Int \mid v = x + 1\} \\ &dec :: x : Int \rightarrow \{Int \mid v = x - 1\} \end{aligned} \tag{2.4}$$

The specification in (2.2) is decomposed into subproblems. In this case, only the subproblem of the type of output will be considered, because this specifies precisely the solution to the problem [29]. The variables $n : Nat$ and $x : \alpha$, and the function $replicate$ are added as components (the function name is added because of recursion). To ensure termination, the version of replicate in the group of components has the input n restricted to natural numbers strictly smaller than n instead of being just a positive number, so n works as the variant function in this case. Considering the subproblem, we mentioned that we could observe two options: Nil would satisfy the initial specification with n equal to 0 and Cons with n larger than 0. Here, the synthesizer does not assume these options, instead adding an unknown predicate to the set of components and searching for a component that satisfies the original specification under this predicate. In a general manner, it resorts to a solver to discharge a constraint problem and assign a value to the predicate. If this value is true, the candidate solves the specification in its hole. If not, it solves under a refinement of the specification (and searches for another solution to complement). The solver may also indicate that the enumerated candidate is not the right one. At the end, SYNQUID finds that Nil satisfies the specification under $n \leq 0$ and finds that for $n > 0$ Cons is the solution. The procedure continues originating the solution in (2.5).

$$\begin{aligned} replicate = &\lambda n.\lambda x.\textbf{if } n \leq 0 \\ &\textbf{then } Nil \\ &\textbf{else } \text{Cons } x \text{ (replicate (dec } n) \ x) \end{aligned} \tag{2.5}$$

To consider the situation of having simple types together with examples to define the goal type instead of the refinement types, we take an example from [27]. The goal is also to construct a list. The goal function is defined with types and input-output examples. Since none of the list constructors agreed with all examples, the problem ended up being decomposed into two subproblems. The group of examples that originally defined the problem was divided into the subproblems in order to define their behavior.

The same happened with the refinements in the example above that turned into $n \leq 0$ for a subproblem and $n > 0$ for another.

To summarize, type-directed approaches start with a specification in terms of types. These approaches do not try to solve the whole problem at once, instead they decompose it into parts and leverage type information (and possibly other useful information such as examples or refinements) into each part to prune candidates. Once every part is solved, the solution is put together. This results in a decrease of the amount of candidates considered.

2.2.2 Applications

We have realized that deductive work emphasizes more the type of programs that are synthesized rather than the functionality of the program itself. In contrast to inductive synthesis that it is more goal-oriented, it focuses more on the concrete applications as mentioned above in Section 2.1.2. Deductive synthesis has applications such as producing imperative, iterative, recursive and functional programs. Loop-free programs are simpler to produce than recursive or iterative programs. Recursive/iterative programs require using mathematical induction in theorem-proving [24, 25]. In Dijkstra's work [11], to build an iterative program, there was the need to define a variant function to prove termination and a invariant relation that holds on every iteration. Demonstrating that recursion/iteration terminates brings an extra difficulty into the problem. A great number of type-direct approaches synthesize functional programs. For example, sorting algorithms, manipulations on trees and lists [29].

Chapter 3

Summary of Key Related Work

As mentioned in Section 1.1.1, this work will encompass three key conceptual developments (a core, type-driven specification language for data schemas and their evolution that is amenable to effective automated synthesis techniques) and their subsequent implementation. We now summarize the most significant pieces of work from the literature that relate to data-oriented specifications, program evolution and synthesis.

3.1 Specifying Data Schemas

The work on nested relational calculus (NRC) [6] allows the development of database query languages with typeful operations. Traditional query languages do not permit queries on structures such as nested relations or collections such as bags and lists, which are allowed in NRC [6]. Considering a database as a collection of records, NRC provides operations for sets and records separately, while providing a language where these may be freely combined in structures (nested relations). The operations on sets can be generalized to collection types such as bags and lists. Moreover, the authors present a language for iterating over collections and consider the well-defined fragments of the language. To reason about well-defined fragments, the authors introduce collection constructs and use the paradigm of structural recursion and operations for constructing sets, in order to have a language that manipulates collections of records. The work also presents a calculus and functional algebra for the language.

Our core, type-driven specification language for data schemas will be heavily inspired by the work on nested relation calculus, leveraging the existing technology on reasoning about databases as collections of records from a programming language perspective. Our work will likely consist of extending existing approaches with forms of (potentially stateful) type refinement [15] that allow us to specify rich constraints on data schemas.

3.2 Program Synthesis and Data-centric Synthesis

As mentioned above, our aim is to use type-directed synthesis techniques. Current type-directed approaches often use type information jointly with additional data, such as examples [27] or predicates added to the types [29], to further restrict the search space.

In the context of functional programming, in [29] the synthesis goal is specified using polymorphic refinement types, from the type information the synthesizer will produce a recursive function that provably satisfies the specification. They use predicates from a decidable logic that permit automatic verification (synthesis). The synthesis procedure works by either choosing a component from a library of components or by decomposing the problem into subproblems, and then merging the solutions into a final one. When decomposing the problems into subproblems, type information is propagated into the subproblem. To solve the subproblems, the synthesizer relies on picking components from a defined library which then uses to perform refinement type checking and refinement inference by resorting to an SMT Solver. To perform refinement type checking there is the need to solve a subtyping constraint with an unknown refinement type. Every enumerated component is submitted to refinement type checking which in turn concludes if the component does or does not satisfy the specification, or if it satisfies the specification under a refinement, this makes the algorithm generate a conditional and look for a solution with the negated refinement. Crucially, the work [29] reports that its synthesis procedure is better suited to synthesize programs that manipulate list/trees and data structures, which may be suited for our goals when we consider database tables as lists of records.

Using input-output examples instead of predicates as additional information, the type-directed approach in [27] also is inserted in the context of functional synthesis. The work [27] defines the desired function by its type and input-output examples, it works either by diving the problem into subproblems refining the goal type and examples or by enumerating well-typed terms and using type-checking to verify the candidate. Using the input-output examples enables the construction of a refinement tree that represents all possible refinement of the examples and constraints the generated code. Even though both approaches are type-directed, our approach is closest to [29] since it uses refinement types to specify the synthesis goal. Propagating refinement types top-down allows to better restrict the possible candidates than propagating examples, since examples do not fully specify the behavior (while easier to provide at times). Both works [27, 29] allow to greatly prune candidates dividing the problem of finding a solution in subproblems propagating type and other information top-down, this allows to solve each subproblem only considering this restricted information and find/evaluate candidates at this level, instead of enumerating all possibilities and then evaluating. Our goal will be to synthesize code for data schema transformations, which the works do not directly address.

3.3 Synthesis and Program Evolution

Some work focus on synthesizing evolutions/changes [31, 40]. In the context of data-driven applications. The work [40] addresses the problem of database schema refactoring over the software life cycle under the lens of program synthesis.

As applications evolve over time, various changes to the original database schema design are required, either for performance reasons or simply to support new features. Moreover, changes to the DB schema typically also require changing various parts of the application layer, and both the schema and the application changes must necessarily preserve the previous semantics in order to maintain the original functionality. These aspects combined make this a particularly challenging and error-prone task, especially when changes in the schema involve splitting and merging relations or moving attributes across different tables.

The work [40] aims to simplify this process by considering a synthesis procedure that given a so-called database program (i.e. a set of SQL DB transactions), the source schema and a target schema, synthesizes a new database program over the new schema that is semantically equivalent to the original program, ensuring no behavior is lost as the DB schema evolves. At a high-level of abstraction, their approach first attempts to produce a value correspondence that maps attributes in the original schema to the new one (e.g. based on attribute names). Given such a correspondence, a program sketch is generated (i.e., database program where some of the tables, attributes, or boolean constants are unknown), representing the space of all programs that may be equivalent to the original program, respecting the value correspondence. Finally, an instantiation of the sketch that is equivalent to the original program is computed via enumerative search, using a novel notion of minimum failing inputs to dramatically prune the search space.

On one hand, the approach has advantages such as no need for user input, being sound since the output is provably equivalent to the original and guarantees that it finds an equivalent program if it exists. On another hand, it can only handle schema changes that are expressible in terms of value correspondence (e.g. merging two columns into a single column and using operations to extract original values in a query would not be possible), extending the system to account for richer changes is hard due to very strong guarantees of program equivalence and it does not support database programs with if statements or loops. Given this, the search techniques may be useful, though this is slightly limited compared to what we want to do that is closer to type-driven synthesis, even if they are not traditionally applied to this context.

The work [31] addresses the problem of program synthesis for program evolution in the context of class replacement in object oriented (i.e. Java) programs. Throughout the software life cycle, there is often a need to refactor applications to use updated versions of libraries or to migrate to different libraries with a similar feature set. In this setting, ensuring that the application behavior is undisturbed by the library change is both non-trivial, since libraries are not necessarily backwards compatible, and error-prone, since

the chosen replacement class can differ in terms of internal data representation, interface signature or even underlying functionality. The work aims to automate this error-prone library replacement in applications by synthesizing an adapter class for a given replacement class, producing a class that is equivalent to the original.

The synthesis procedure, dubbed MASK, takes as input the original class and a replacement class, producing methods that implement the interface of the original class using the replacement. In order to ensure that the behavior of the original class is preserved, their framework symbolically executes all methods of both classes to construct equivalence predicates that relate equivalent states in the original and replacement classes, subsequently requiring synthesized methods and constructors to return equivalent values and leave objects in equivalent internal states. This allows for methods to be synthesized in isolation but guaranteeing that any sequence of invocations produces equivalent behavior. Given such equivalence predicates, MASK synthesizes an adapter sketch [35] of the original class, containing sketches for each class method. Finally, a so-called sketch harness is generated, which contains correctness checks that ensure the behavioral equivalence of the original and generated classes.

This work is the first of its kind, requires little to no user input and gives semantic correctness guarantees. The approach has drawbacks such as using symbolic execution to generate the predicates makes use of array fields or recursive data structures unsound. Similarly, methods with loops that depend on symbolic conditions or recursive calls are unrolled only up to some constant, limiting the correctness of the approach. Generics are severely limited and programs may only use instantiated generics (i.e. cannot adapt a generic class, only classes that use instantiated generic classes).

Chapter 4

Approach

Our technical approach will define all aspects of the program synthesis problem. We must elucidate how the specification is expressed, how the program space is defined and the synthesis technique. The proposed approach was already introduced and now we will give a more detailed description.

We will begin by defining a core language for the specification of data schemas that can also capture integrity, business/logic, and security constraints. The language will enable us to define rich type-based specifications, using dependent and refinement types, the refinement types are used to enforce constraints which are defined by predicates. The specification language will work at a meta-level, defining, not only new types, but also stateful modifications to existing stores, meaning that the existing values of that type stored in a database must be modified accordingly. For instance, the refinement type $\{x : int \mid key(x)\}$ uses the predicate $key(x)$, which may specify that x is a key (i.e. it is a unique value). This can be used to specify the production of a database table with the following (incomplete) specification:

$$inputTypes \rightarrow \{\mathbf{col_1} : \{x : int \mid key(x)\}, \mathbf{col}_2 : \{y : int \mid refTable(y)\}\}$$

The above specification, where the input types are not specified for the sake of simplicity, can be a specification that creates a table with two columns ($\mathbf{col_1}$ and $\mathbf{col}_2$). The type for $\mathbf{col_1}$ enforces that the values of the column must be keys (i.e. unique), whereas the type for $\mathbf{col}_2$, through predicate $refTable(y)$, explicates that the values must be driven from some other table in the data schema.

With a different choice of predicates, other kinds of constraints can be defined. In order to investigate which predicates are more useful in our synthesis context, we will go through a requirement elicitation phase. When defining stateful modifications, not just

introducing new types, the specification also needs to take into consideration the current state of the data schema that is about to be modified.

We then plan to define the transformation on the types of the data schema by the construction of a core language of modifications, which will be used as the target language for our synthesis procedure. The language will be inspired in languages like nested relational calculus [6], which define language operations regarding types, where a relational database consists of sets of records, allowing to define operations for records and sets. The language may be defined separately or integrated in the specification language through the use of specialized meta-programming primitives in the style of refinement kinds [7]. Refinement kinds [7] enable to provide a specification with predicates over the domain of types, similarly to refinement types that use predicates over the domain of values of a type. The resulting system is amenable to type-directed meta-programming, since it introduces primitives to manipulate types as data, while also being able to express data constraints. Meta-programming techniques make it possible for a program to receive programs and types as input data, which makes it useful for our approach since for doing modifications we will need the types of modifications and the data stored in the data layer.

Given a rich type-based specification using dependent and refinement types, and data stored in the data layer, our tool will apply a type-directed approach to synthesize both database scripts and/or code for a data layer access component that correctly implements integrity constraints, business logic validation, and access control rules. Our tool will integrate with other components that generate interface components or dialog-based interfaces for an easier expression of user intent, these components will interactively try to build the specification (expressed in the defined specification language) used as input for the synthesis procedure. The procedure will use the specification expressed as input and output types of the data schema, and produce code in the modification language (target of the synthesis). The synthesis process will follow in a similar way as current type-directed approaches such as [29], decomposing the specification into subproblems while propagating type information and recurring to SMT solvers [5] to do refinement type checking of possible solutions. Composing the solution to all subproblems, originates the final solution.

The output of the tool we intend to design and develop will synthesize database scripts and code for the data access layer that correctly transforms the system and its data. The prototype will be illustrated and integrated in the context of a product of the OutSystems company.

Chapter 5

Work Plan

The literature review, elaboration of the dissertation plan, and the work plan for the next phase is described chronologically in the gantt chart in Figure 5.1. In order to accomplish the proposed work, the work plan is as follows:

- Our first goal is to design a specification language for data schemas (structured or semi-structured) with refinement types. To achieve that, we have to start by a requirements elicitation phase to understand which refinements are better suited in this context. Once the requirements are established, we may start the design of the typed specification language (Task 2).
- We will design a modification language that establishes the available transformations on the types of the data schema. This may be a separate language or integrated in the specification language through the use of specialized meta-programming primitives within the style of refinement kinds [7] (Task 3).
- Once we can specify the synthesis problem specification (specification language) and the synthesis target (modification language), we will apply a type-directed synthesis technique to produce the specified transformations (Tasks 4 and 5).

  A first phase on this process may be to synthesize code from a specification with simple types, ensuring that the process is indeed generating transformation code from the specification.

  A second phase would take the basic mechanism of synthesis with simple types and further improve the procedure to start from specifications with properties (e.g. security) defined by refinement types, and synthesize code that guarantees the satisfaction of those properties (statically or dynamically).

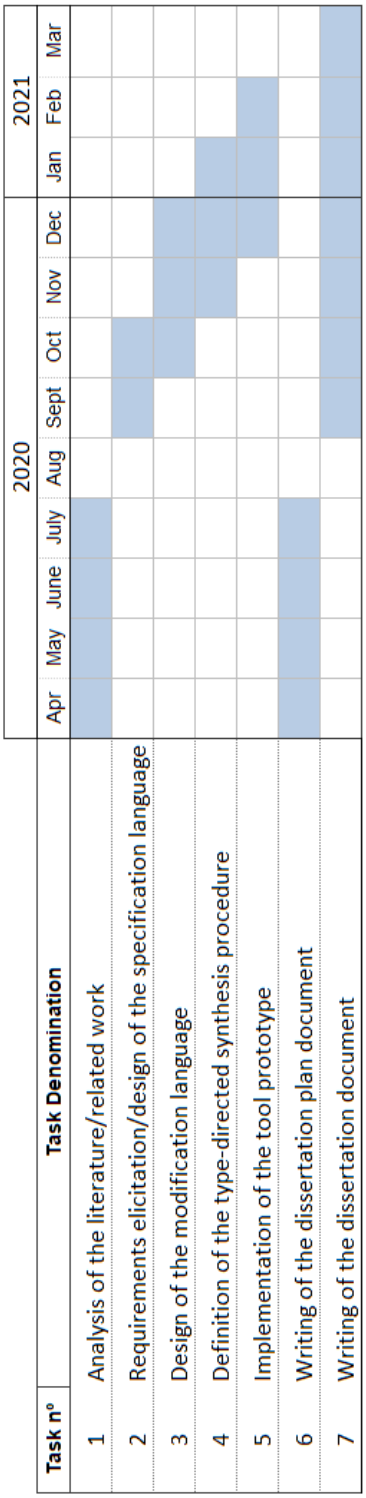

Figure 5.1: Gantt Chart

Bibliography

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. "Syntax-Guided Synthesis." In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2013, pp. 1–17.

[2] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. "SyGuS-Comp 2016: Results and Analysis." In: *Electronic Proceedings in Theoretical Computer Science* 229 (2016), 178–202. ISSN: 2075-2180. DOI: 10.4204/eptcs.229.13. URL: http://dx.doi.org/10.4204/EPTCS.229.13.

[3] R. Alur, A. Radhakrishna, and A. Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer." In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by A. Legay and T. Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18. URL: https://doi.org/10.1007/978-3-662-54577-5_18.

[4] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. "FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, 218–228. ISBN: 9781450334686. DOI: 10.1145/2737924.2737952. URL: https://doi.org/10.1145/2737924.2737952.

[5] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[6] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. "Principles of Programming with Complex Objects and Collection Types." In: *Selected Papers of the Fourth International Conference on Database Theory*. ICDT '92. Berlin, Germany: Elsevier Science Publishers B. V., 1995, 3–48.

[7] L. Caires and B. Toninho. "Refinement Kinds: Type-Safe Programming with Practical Type-Level Computation." In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360557. URL: https://doi.org/10.1145/3360557.

[8] P. J. Denning. "The Field of Programmers Myth." In: *Commun. ACM* 47.7 (July 2004), 15–20. ISSN: 0001-0782. DOI: 10.1145/1005817.1005836. URL: https://doi.org/10.1145/1005817.1005836.

[9] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. "RobustFill: Neural Program Learning under Noisy I/O." In: *CoRR* abs/1703.07469 (2017). arXiv: 1703.07469. URL: http://arxiv.org/abs/1703.07469.

[10] E. W. Dijkstra. "The Humble Programmer." In: *Commun. ACM* 15.10 (Oct. 1972), 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: https://doi.org/10.1145/355604.361591.

[11] E. W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Commun. ACM* 18.8 (Aug. 1975), 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975.

[12] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376442. URL: https://doi.org/10.1145/3313831.3376442.

[13] A. Fariha and A. Meliou. "Example-Driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity." In: *Proc. VLDB Endow.* 12.11 (July 2019), 1262–1275. ISSN: 2150-8097. DOI: 10.14778/3342263.3342266. URL: https://doi.org/10.14778/3342263.3342266.

[14] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. "Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples." In: *SIGPLAN Not.* 52.6 (June 2017), 422–436. ISSN: 0362-1340. DOI: 10.1145/3140587.3062351. URL: https://doi.org/10.1145/3140587.3062351.

[15] T. Freeman and F. Pfenning. "Refinement Types for ML." In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, 268–277. ISBN: 0897914287. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[16] S. Gulwani. "FlashExtract: A Framework for Data Extraction by Examples." In: *PLDI '14, June 09 - 11 2014, Edinburgh, United Kingdom*. 2014. URL: https://www.microsoft.com/en-us/research/publication/flashextract-framework-data-extraction-examples/.

[17] S. Gulwani. "Programming by Examples (and its Applications in Data Wrangling)." In: *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016. URL: https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/.

[18] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*. Vol. 4. NOW, 2017, pp. 1–119. URL: https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[19] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. "Software Engineering and Formal Methods." In: *Commun. ACM* 51.9 (Sept. 2008), 54–59. ISSN: 0001-0782. DOI: 10.1145/1378727.1378742. URL: https://doi.org/10.1145/1378727.1378742.

[20] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. "Oracle-Guided Component-Based Program Synthesis." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, 215–224. ISBN: 9781605587196. DOI: 10.1145/1806799.1806833. URL: https://doi.org/10.1145/1806799.1806833.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. "GenProg: A Generic Method for Automatic Software Repair." In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72.

[23] W. Lee, K. Heo, R. Alur, and M. Naik. "Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models." In: *SIGPLAN Not.* 53.4 (June 2018), 436–449. ISSN: 0362-1340. DOI: 10.1145/3296979.3192410. URL: https://doi.org/10.1145/3296979.3192410.

[24] Z. Manna and R. Waldinger. "A Deductive Approach to Program Synthesis." In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), 90–121. ISSN: 0164-0925. DOI: 10.1145/357084.357090. URL: https://doi.org/10.1145/357084.357090.

[25] Z. Manna and R. J. Waldinger. "Toward Automatic Program Synthesis." In: *Commun. ACM* 14.3 (Mar. 1971), 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568. URL: https://doi.org/10.1145/362566.362568.

[26] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, A. Polozov, R. Singh, B. Zorn, and S. Gulwani. "User Interaction Models for Disambiguation in Programming by Example." In: *28th ACM User Interface Software and Technology Symposium (UIST 2015)*. ACM – Association for Computing Machinery, 2015. URL: https://www.microsoft.com/en-us/research/publication/user-interaction-models-for-disambiguation-in-programming-by-example/.

[27] P.-M. Osera and S. Zdancewic. "Type-and-Example-Directed Program Synthesis." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, 619–630. ISBN: 9781450334686. DOI: 10.1145/2737924.2738007. URL: https://doi.org/10.1145/2737924.2738007.

[28] H. Peleg and N. Polikarpova. "Perfect is the Enemy of Good: Best-Effort Program Synthesis." In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by R. Hirshfeld and T. Pape. LIPIcs. to appear. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 2:1–2:30. DOI: 10.4230/LIPIcs.ECOOP.2020.2. URL: http://cseweb.ucsd.edu/~hpeleg/ecoop2020.pdf.

[29] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. "Program Synthesis from Polymorphic Refinement Types." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, 522–538. ISBN: 9781450342612. DOI: 10.1145/2908080.2908093. URL: https://doi.org/10.1145/2908080.2908093.

[30] O. Polozov and S. Gulwani. "FlashMeta: A Framework for Inductive Program Synthesis." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, 107–126. ISBN: 9781450336895. DOI: 10.1145/2814270.2814310. URL: https://doi.org/10.1145/2814270.2814310.

[31] M. Samak, D. Kim, and M. C. Rinard. "Synthesizing Replacement Classes." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371120. URL: https://doi.org/10.1145/3371120.

[32] E. Schkufza, R. Sharma, and A. Aiken. "Stochastic Superoptimization." In: *SIGPLAN Not.* 48.4 (Mar. 2013), 305–316. ISSN: 0362-1340. DOI: 10.1145/2499368.2451150. URL: https://doi.org/10.1145/2499368.2451150.

[33] S. Shapiro. "Splitting the difference: the historical necessity of synthesis in software engineering." In: *IEEE Annals of the History of Computing* 19.1 (1997), pp. 20–54.

[34] R. Singh and S. Gulwani. "Predicting a Correct Program in Programming by Example." In: *Computer Aided Verification*. Ed. by D. Kroening and C. S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 398–414. ISBN: 978-3-319-21690-4.

[35] A. Solar-Lezama. "Program sketching." In: *International Journal on Software Tools for Technology Transfer* 15.5-6 (2013), pp. 475–495. DOI: 10.1007/s10009-012-0249-7. URL: https://doi.org/10.1007/s10009-012-0249-7.

[36] S. Srivastava, S. Gulwani, and J. S. Foster. "From Program Verification to Program Synthesis." In: *SIGPLAN Not.* 45.1 (Jan. 2010), 313–326. ISSN: 0362-1340. DOI: 10.1145/1707801.1706337. URL: https://doi.org/10.1145/1707801.1706337.

[37] 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 and Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[38] M. E. P. Valdez. "A Gift from Pandora's Box: The Software Crisis." Order No: GAXD–82988. Doctoral dissertation. GBR, 1988.

[39] C. Wang, A. Cheung, and R. Bodik. "Synthesizing Highly Expressive SQL Queries from Input-Output Examples." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, 452–466. ISBN: 9781450349888. DOI: 10.1145/3062341.3062365. URL: https://doi.org/10.1145/3062341.3062365.

[40] Y. Wang, J. Dong, R. Shah, and I. Dillig. "Synthesizing Database Programs for Schema Refactoring." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, 286–300. ISBN: 9781450367127. DOI: 10.1145/3314221.3314588. URL: https://doi.org/10.1145/3314221.3314588.

[41] S. Zhang and Y. Sun. "Automatically Synthesizing SQL Queries from Input-Output Examples." In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. Silicon Valley, CA, USA: IEEE Press, 2013, 224–234. ISBN: 9781479902156. DOI: 10.1109/ASE.2013.6693082. URL: https://doi.org/10.1109/ASE.2013.6693082.